University of Zurich[UZH]

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

Linux on Tiny Microcontrollers

Filip Dombos
Baden, Switzerland
Student ID: 14-939-664

Supervisor: Eryk Schiller
Date of Submission: July 27, 2022

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

ifi

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 ist die Kurzfassung... [36]

HELLO CITERS[11]

This is the summary for the english language

Acknowledgments

First and foremost, I would like to thank my research supervisor, Dr. Eryk Schiller, without whom this thesis would not have been possible. I would like to thank, Dr. Eryk Schiller for his guidance, patience, for every meeting we held, and his high-level insights on the topic at hand. Furthermore, I want to extend my gratitude to Prof. Dr. Brukhard Stiller for providing me with the opportunity to write my bachelors thesis at the Communication Systems Group (CSG) in the University of Zurich.

Thanks also goes to STM Technical Marketing Manager Marco Sanfilippo and STM as a company, for providing us with feedback, consultation, and with two STM32L4 development boards, on which we performed some tests.

Contents

Chapter 1

Introduction

1.1 Motivation

In the previous two decades, the development and use of sensing- and connectivity-enabling electronic gadgets has steadily increased, in some areas substituting conventional physical devices [42]. With a central focus on interconnectedness, the appropriately named, Internet of-Things (IoT) refers to the billions physical gadgets connected to the internet of throughout the world, all gathering, and more importantly, sharing massive amounts of data. IoT has become an integral part of the lives of billions of people worldwide, not only due to the sheer number of connected devices and potential use cases [20] but also due to the diversity and variety of IoT solutions [30]. Many people believe that IoT is the essential development of the twenty-first century, because it affects almost every industry, from healthcare to transportation. However, as the world around us becomes increasingly linked, protecting these resource-constrained devices has become critical.

Linux runs on some microcontrollers, such as the Raspberry Pi (RPI) family. For example, RPI 3B is a small computer equipped with computing essentials, e.g., the Advanced RISC Machines (ARM) Cortex central processing unit (CPU), Random Access Memory (RAM), but not necessarily of the latest generation, having minor energy requirements. As an example, instead of a hard drive, an RPI is equipped with a flash memory card, on which an Operating System (OS) may be installed. It also offers Universal Serial Bus (USB) connectors, a video output, and a Wireless Fidelity (Wi-Fi) adapter. As RPI is a small computer, a regular general-purpose OS such as Linux-based Ubuntu for the ARM architecture can also be supported. However, Raspberry Pi OS, previously known as Raspbian, is a typical distribution of choice for the Raspberry Pi device family. Linux is an excellent success on Raspberry Pi as it simplifies the development of microcontroller

applications, as a regular operating can be used with which users are already familiar. Currently, a new generation of microcontrollers is being introduced into the market. For example, the ESP32 device family is based on the dual-core Reduced Instruction Set Computer (RISC)-based Tensilica LX6 processor with a maximum frequency of 240 MHz, 8 MB PSRAM, and 4 MB flash seems to be a great choice to run Linux on those devices as well. Linux was first developed for the Complex instruction Set Computer (CISC)-based Intel 386 (i386) architecture in 1991. Back then, the typical CPU clock speed of the i386 system was between 12 MHz to 40 MHz, while the typical computer was equipped with several megabytes of RAM (e.g., 4 MB). As ESP32 already exceeds the specification of early i386 systems, it seems to be that porting Linux for those devices shall be possible. 8 mega byte (MB) pseudo static RAM (PSRAM) on ESP32-WROVER-IE shall be satisfactory to run the kernel, uclibc, and essential binaries.

The cheapest development board for ESP32-WROVER-IE costs around 10 CHF. However, when one does not need a development board, an ESP32-WROVER-IE costs 3 CHF. Regular RPI 0 devices, which already contain an ARM Cortex CPU, cost 22-24 CHF, while an RPI 3 costs around 38 CHF. The cost reduction from RPI 3 to Linux-capable ARM-based RPI 0 is already 42%, and the further cost reduction from RPI (Zero development board) to ESP32-WROVERIE would be another 54%. Running Linux on regular ESP32-WROVER-IE (i.e., not with a development board) would mean a cost reduction of 92% in comparison to an RPI 3 device. This is a massive incentive to port a Linux-based distribution towards the new family of devices. The migration of Linux on ESP32 should be possible as there is already a Linux kernel project supporting the kernel execution on the Tensilica LX6 processor family. Furthermore, there are emulators of the RISC-V platform for Tensilica LX6 processors, allowing for executing the code compiled for RISC-V architectures displaying poor performance [10].

RISC-V is another open standard instruction set architecture (ISA) that was first released in 2010 and is based on RISC (Reduced Instruction Set Computer) principles. A layered security method that employs a Trusted Execution Environment (TEE) provided in the RISC-V architecture is a gamechanger in the IoT industry. Unlike most other ISA designs, RISC-V is available under open-source that does not require license fees. RISC-V hardware is available from several companies. Opensource operating systems with RISC-V support are available, and many major software toolchains support the instruction set. Furthermore, there is a Linux kernel available for the RISC-V processor family. As an example, the Espressif ESP32-C3 is a single-core, 32-bit, RISC-V-based MCU with 400KB of SRAM and a 160 MHz clock speed. It includes 2.4 GHz Wi-Fi and Bluetooth 5 (LE) with built-in long-range capability.

1.2 Description of Work

The focus of this thesis is to lay a foundation for porting the open source software (OSS) Linux kernel to tiny microcontrollers, to condense a pratical guide for future studies, which people can use, and upon which they can build. Due to to sheer amount of IoT devices paired with I/O periferals such as sensors, and the endless configurations of the Linux kernel, the combinations appear infinite. To set formal footing in the standardization process of microcontroller units (MCU) with the Linux kernel, this thesis explores the vast amount of tools used for embedding and evaluating the Linux kernel on MCUs.

1.3 Thesis Outline

Chapter 2

Related Work

2.1 The Internet-of-Things

There was already a market for microcontrollers at the time Intel introduced the 4004 as the first single-chip microprocessor. By the end of 1971, Texas Instruments began marketing the TMS1802, a modern calculator designed for use in cash registers, watches, and measurement devices. But the first MCU's that gained widespread use were Intel's next generation 8-bit controllers, such as the Intel 8048 and Intel 8051 operating on the MCS-51 instruction set architecture (ISA), most notably used in Desktop peripherals such as the keyboards. While these MCU's build the foundation of IoT, the edge hardware, the first primitive IoT device, a toaster that can be turned on and off remotely, was introduced in 1990 [40].

The phrase "Internet of Things", was initially coined by Kevin Ashton in 1999. Ashton made the initial proposal for the Internet of Things (IoT), which he defined as a network of radio-frequency identification (RFID)-enabled, interoperable, linked items. IoT can include billions of intelligent, communicative "things", enabling connections between people and these things at any time, anywhere, with anything, and with anybody, preferably via any path/network and any service. Thus envisioning a system in which omnipresent and ubiquitous devices will link the Internet to every physical object [44, 48].

The quantity and diversity of IoT devices and solutions have multiplied due to the markets quick development. According to IoT reports, there were between 6.1 billion and 8.4 billion IoT devices in use in 2017, in 2020 growing to 20.4 billion, and by 2025 it is anticipated that there will be 75 billion IoT devices [33, 35]. Although others report fewer devices [21], the same upwards trend is captured. This discrepancy is indicative of the

diverse manufacturers, countless forms, and the enormous amount of devices, that make it challenging to identify a precise quantity. Despite the lack of definitive numbers, it clearly shows that IoT has become more relevant and omnipresent, with growth that isn't showing any signs of slowing.

With Governments heavily investing into initiatives such as the UK's Future Internet Initiatives, the European Research Cluster on IoT, the National IoT Plan of China's Ministry of Industry and Information Technology, the Italian National Project of Netergit, and Japan's u-Strategy [44]. Uses of IoT are numerous, including medical, industrial, and consumer. Examples include .... and automatic irrigation systems for farms [45].

2.2 IoT Architecture

But IoT does not only encompasss the tiny edge devices. It includes sensing, computing, networking, and cloud. Yet there is no concensus regarding IoT architecture, a multitude of models have been proposed, the most basic of which being the three-layer architecture [43].

1. The physical layer, which contains sensors for perceiving and gathering environmental data, is the perception layer. It detects certain physical factors or other intelligent things in the surrounding area.
2. The network layer is in charge of establishing connections with other intelligent objects, network components, and servers. Its capabilities are also employed for processing and transferring sensor data. Notably IPv6.....
3. The application layer delivers application-specific services to the user is the responsibility. It describes a variety of uses for the IoT, including smart homes, smart cities, and smart health.

[SHOW FIGGURE HERE]

2.3 The lack of standardization

[I had a really good quote here, but I can't find it anymore... something about the consequence of rapid development, and the need to standardize] As a result of the rapid

development of IoT, the industry has concentrated on creating and delivering the appropriate kinds of hardware. In the current model, the majority of IoT solution providers have been building all components of the stack, from the hardware devices to the relevant cloud services, or as they call it "IoT solutions" [30].

As interoperability is ensured by standardization, it improves the effective integration and information exchange between distributed systems. But manufacturers are using own standards which inevitably has the effect that devices can not talk to each other. [28] conclude that the lack of standardization negatively impacts the IoT industry.

The requirement for a standard model to carry out typical IoT backend functions, such as processing, storing, and firmware upgrades, is growing in importance as the industry develops. Different IoT solutions are expected to cooperate with shared backend services in this new architecture, which will provide levels of interoperability, portability, and management that are almost unattainable with the current generation of IoT systems [30].

"The Internet of Things Might Never Speak a Common Language" [38].

2.4 IoT Security Issues

The term IT-Security was defined as follows. Information integrity, availability and confidentiality [47]. In a paper by Schiller et al. [42] security is defined as message confindentiality. This is exclusively the case if only the sender and receiver are aware of the existence of the message and only they can verify its validity.

The security strategies and procedures that have been suggested are mostly based on traditional network security procedures. However, given the variety of the devices and protocols including the quantityÂ of nodes in the network, implementing security methods in an IoT system is more difficult than with a typical network [34].

2.5 Operating Systems on IoT

✓

Linux is a monolithic kernel..... elaborate on this. [37] have analysed different OS's for IoT edge devices. By compiling a list of [37] goes as far as stating that "Linux will never run on these chips", by chips refering to ARM Cortex-M series. [41]

2.6 Summary

With huge potential and growth to satisfy the rapid demand

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Realtime
Linux	∼1MB	∼1MB	✓	✓	✓	✗	○	○
Contiki	<2kB	<30kB	○	✗	○	✓	○	○
Tiny OS	<1kB	<4kB	✗	✗	○	✓	✗	✗
RIOT	∼1.5kB	∼5kB	✓	✓	✓	✓	✓	✓

Table 2.1: Key characteristics of TinyOS, Contiki, RIOT, and Linux

Chapter 3

Overview of Hardware & Technologies

Due to the explorative nature and the width of this thesis, we have to introduce a variety of different IoT ISA's, System-on-a-Chip manufactureres and their relevant products, open source projects with the aim to aid in the process of embedding, toolchains, and technologies. This chapter serves as an overview of the various devices and tools that were discovered, and may help further research which aim to achieve similar goals.

3.1 Micro Computing

Von Neumann's Architecture states that a modern computer requires a couple of core components to be able to run [29]. Figure 3.1 shows a visual representation.

- A **Processor Unit**, or a **core processing unit (CPU)** can further be divided into two subcomponents, the Controll Unit (CU) and the Arithmetic Logic Unit (ALU). This component performs operations and instructions on the data stored in the main memory unit and on the I/O devices.
- The **Main Memory Unit**, or simply memory, stores data and the operations that need to be performed on this data.
- The **Input/Output (I/O) Devices**, can be anything that allows us to interact with the computer, such as a mouse or keyboard, or simply just an LED light that gets turned on.

We can apply the definition of Von Neumann's Architecutre to further categorize two IoT edge devices:

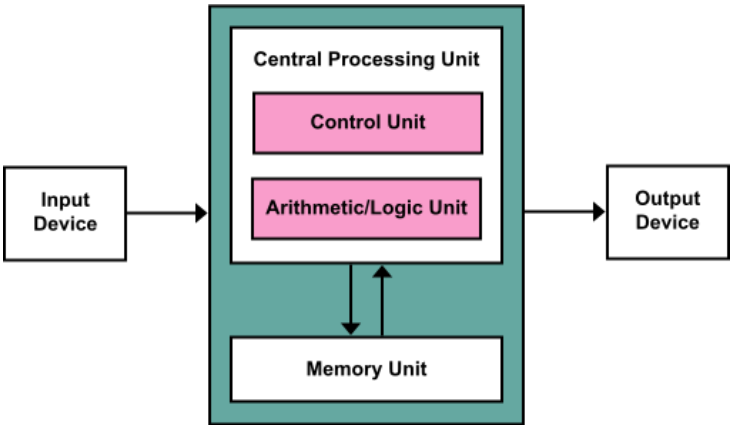

Figure 3.1: Von Neumann Architecture [**?**]

- A **Micro controller unit (MCU)** is an integrated, fully capable, self sufficient computer on a sincle chip. It can run a ”bare metal interface”, meaning that it doesn't require to run an OS. Without which, it can run a single thread, or a controll loop, forever. As the title of this thesis implies, MCUs are the main focus of this thesis.
- A **Micro processor unit (MPU)** requires support from surrounding chips that enable various functionalities like memory, interfaces, and I/O and cannot act as a stand alone computer. The MPU, according to the Von Neumann's Architecture, is just the processor unit.

While the above terms, MCU and MPU, are oftentimes used interchangably, we want to point out the differences of these two families. It is far easier to run an OS such as the Linux Kernel [18] on MPU enabled devices, such as the Raspberry Pi 4, because it is not limited to the capabilites on the chip, and can easily be extended with, i.e. more RAM [17]. A MCU is limited to the design and capabilities of the chip. Hence, a MPU runs with far higher processing capability and much larger applications, while MCUs are for lightweight computing where the OS, if one decides to use one, is integrated on-chip. But because some MCUs have straightforward software drivers for more complex peripherals and more MPUs are available that have integrated peripherals on-chip, the gap between MCUs and MPUs is becoming less evident [39, 46].

3.2 Processors

When compiling software for a target platform, one must be aware of the different instruction set architectures (ISA) of the target CPUs. Listed below are the most noteworthy processor families that we came across.

- **ARM Cortex-M-Series**, are 32-bit RISC processors, designed for low-cost, low-power, and usually embedded in MCUs and other IoT devices [2].

- **ARM Cortex-A-Series**, are 32-bit or 64-bit RISC processors, in contrast to the Cortex-M-Series, these processors have higher energy consumption that are built for more complex tasks such as supporting an OS [1].

- **Tensilica Xtensa**

As description of the ARM Cortex-M processors seems to fit our premise perfectly, the low-cost and MCU aspect, we will focus on these types of CPUs

3.3 Market Analysis

[MAYBE MOVE THIS CHAMPTER TO THE IMPLEMENTATION?]

To find a suitable MCU that could support Linux, a market analysis had to be performed. The most relevant criteria where having sizable RAM preferably more than 1MB, more than 40MHz CPU clock rate, availability in the region, and a large community because this can simplify development due to the availability of online ressources. Furthermore with the homogenious nature of IoT, directing this thesis towards a smaller target audience would inevitably decrease its value.

With the insights of the market analysis, see table 3.1, and the realization that the required MCU on-chip RAM might not suffice for running Linux, a multitude of sales representatives of hardware manufactureres primarily of STM, were contacted. Among others, Digi-Key Electronics, Anatec AG, Avnet Silica Rothrist and Mouser Electronics. Questions concerning extensibility of RAM were posed with the goal of gaining expert insight. For the purposes of this thesis, more RAM was required.

[MENTIONE: I arranged the boards from STM to get some support.]

THE PICTURE ——>[8] figure 3.2

[ADD PICTURE OF THE ESP32 WROOWIE]

Manufacturer	Chip/Dev Board	Classification	CPU	ISA	bit	Clock rate	RAM	Flash
Raspberry Pi	Raspberry Pi 4	MPU	ARM	Cortex-A72	64	1.5GHz	2-8GB (SDRAM)	asdf
Raspberry Pi	Zero 2 W	MPU	ARM	Cortex-A53	64	1GHz	512MB (SDRAM)	asdf
Raspberry Pi	Pico	MCU	ARM	Cortex-M0+	32	133 MHz	264kB (SRAM)	2MB
Espressif	ESP32	MCU	Tensilica	Xtensa	32	240 MHz	520kB (SRAM)	adsf
Espressif	ESP32-S2	MCU	Tensilica	Xtensa	32	240 MHz	320kB (SRAM)	asdf
Espressif	ESP32-C3	MCU	RISC-V	RISC-V	32	160 MHz	400kB (SRAM)	asdf
Espressif	ESP32-S3	MCU	Tensilica	Xtensa	32	240 MHz	512kB (SRAM)	asdf
STMicroelectronics	STM32F0	MCU	ARM	Cortex-M0	32	48MHz	4-32kB	asdf
STMicroelectronics	STM32F1	MCU	ARM	Cortex-M3	32	72MHz	4-94kB	adsf
STMicroelectronics	STM32F3	MCU	ARM	Cortex-M4	32	72MHz	16-80kB (SRAM)	asdf
STMicroelectronics	STM32F4	MCU	ARM	Cortex-M4	32	180MHz	256kB (SRAM)	asdf
STMicroelectronics	STM32G0	MCU	ARM	Cortex-M0+	32	64MHz	144KB (SRAM)	asdf
STMicroelectronics	STM32L4	MCU	ARM	Cortex-M4	32	adsf	adf	adf
STMicroelectronics	STM32G4	MCU	ARM	Cortex-M4	32	170 MHz	128-512KB (CCM-SRAM)	asdf
Arduino	Due A000062	MCU	ARM	Cortex-M3	32	84MHz	96KB (SRAM)	asdf

Table 3.1: Market analysis of available IoT edge devices

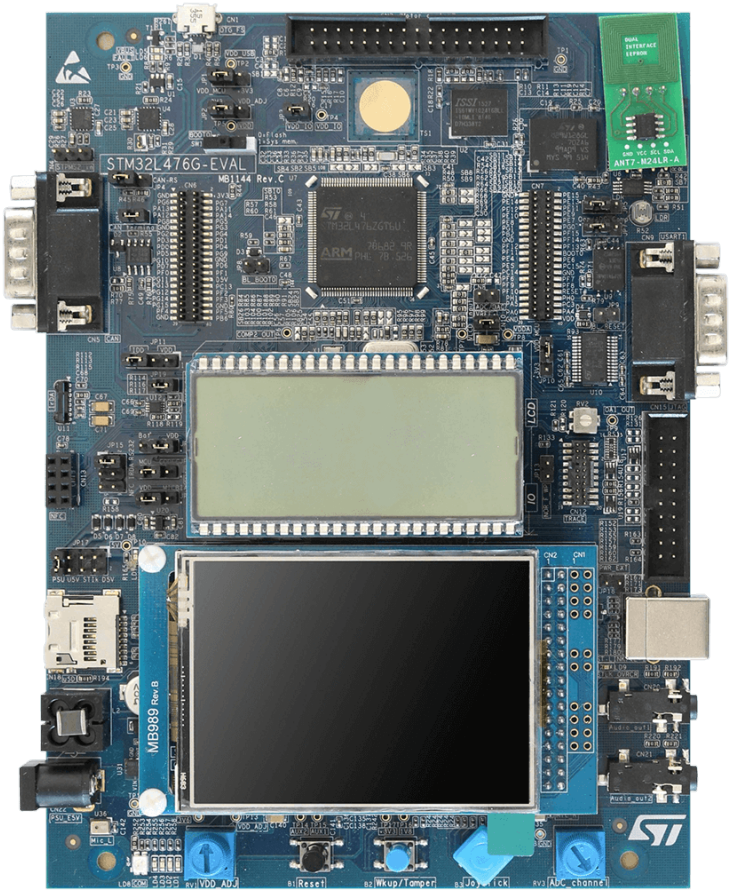

Figure 3.2: STM32L476G-Eval development board [8]

3.4 Toolchains, Cross-Compilation & Libraries

For compiled languages, the appropriately name toolchain combines multiple steps into a single pipeline, to produce binary, or machine code. Each step in the process is a piece of software that fullfils its role or function, and hands work over to the next tool in the chain. Interpreted languages such as Python do not require compilation or toolchains, the interpreter executes, or interprets, the code directly. Since this thesis focuses on code than runs closely to hardware, interpreted languages are not concidered. When compiling code a multitude of factors need to be concidered when choosing a toolchain, because they are specifically designed to work excusivley in the right circumstances. Most importantly, whether the host platform is the same as the target platform. The host is the platfrom on which the compilation takes place, and the target on which the binaries will evenually end up running on. If host and target platfrom are not the same, then we speak of cross-compilation. Another important aspect of cross-compilation is the fact that, especially in our case, the target platform may lack computing power and memory, thus compiling could be very time inefficient or simply be impossible. If compilation in mentioned, cross-compilation is implied, since the goal of this thesis is not to evaluate binary code compiled for the host system, but for the target system, being the MCU edge devices. Broadly simplified toolchains work as follows.

As mentioned, the first tep in a toolchain is source code compilation. Depending on the programming language the source code was writen in, compiler vary. For different compiled languages such as C and C++ different compilers are required. In our case, C is of primary concern because the linux kernel is mostly (98.4%) written in C [23]. From each source file that was compiled in the first step result Assembly code. Assembly code, marked with ending `.s`, is a set of simplified instructions and basic operations, in other words its a human readable abstraction of maschine code, i.e. bits. Nowdays Assembler programming is only utilized in situations when extremely efficient management of processor activities is required, but it is still an intermediate-product in compilation. Assebly code needs to be assembled outputing object code. Finaly the object code files need to be linked with required libraries to form an, either statically or dynamically linked, executable, see section 3.4.1. After successful cross-compilation follows only execution on target device, which might seem like the easiest step, this is further discussed in section 5.5.

For the goals of this tesis, we will be compiling, mostly, C code on host platform operating on `x64` architecture for target platforms on `Armv7E-M` architecutre. Hence cross-compilation will take place with the `gcc-arm-none-eabi` toolchain [6].

[INCLUDE GRAPHIC FOR COMPILATION]

3.4.1 Dynamically & Statically Linked Libraries

A Library is a collection of precompiled and reusable components that hold functionality for common processes. The most common example for such a library is C's `stdio.h` which hold, among others, function `fprintf`, which simply prints characters to the console. As previously stated there are two main methods of linking such functions with the executables, we distinguish between statically linked libraries (SLL) or static libraries, and dynamically linked libraries (DLL) or shared libraries. SLL is the simplest form, as when linking, the contents of the library, specifically the required functions, are included in the executable file. On a small scale this doesn't pose any problems, yet with more exectuables that are loaded into memory, each of these executables contain their own SLL. This can lead to the RAM (or ROM) being occupied by the same function multiple times. Especially when RAM is limited, as is in our case, the redundancy of the same function is not desired. DLL on the other hand requires only one instance of the functionality, all the executables that require this specific function, can access the read-only segment of the library, therefore it can be shared. This process of sharing libraries is aided by a hardware solution, the MMU, see section 3.5.

Once again, for our purposes dynamically linked executables are required, since we want to save as much space as possible, thus only linking the required functions and not the entire library.

3.4.2 Buildroot

Buildroot is a tool that makes cross-compiling a complete Linux system for an embedded devices easier and more automated. It runs primarily on Linux systems. Through the use of this facilitated toolchain, it creates a self compressing version of the Linux kernel `zImage`, a root filesystem, a U-boot bootloader, and root file system and an SD card image file `sdcard.img` [3].

3.4.3 Yocto Project & OpenWrt

Equivalently to Buildroot, the Yocto Project and OpenWrt are tools used for Linux cross-compilation for embedded systems. OpenWrt has a focus on Netoworking, the Yocto

Project does not currently support MMU-less builds. [14, 27]. These tools are not used in this thesis but fill a similar role as Buildroot and are mentioned for the sake of thoroughness.

3.5 Memory Management Unit

The Memory Management Unit (MMU) is hardware that is positioned between the processor and physical memory. If present, memory references from the software, through the processor, are passed through the MMU, which in turn maps these references to the actual memory, where the data being called actually resides. In more techtical terms, the reference points to a virtual memory addresses that the MMU can translated into the physical memory addresses. Hence, the program running on the CPU can doesn't need to know the physical memory address. This can simplify addressing in complicated systems. Furthermore, the MMU facilitates DLL implementation. Linux's memory management system is very complicated and has grown over time, offering a growing number of features, such as `nommu` which means MMU-less devices, often MCUs [12, 13]. While the implementation of DLL, for devices that don't contain a MMU device appears to be possible, it once again is very complicated [19]. There exits Linux variations that are tailored for MMU-less devices, see section 3.6.1

3.6 The Linux Kernel

The Linux Kernel (Linux) was initially created, by Linus Torwalds, in 1991 for i368 based PCs. After its initial appearance it quickly gained traction among developers, and was licenced under GNU General Public License (GPL) as free OSS [9]. At current time, Linux supports all kinds of different target architectures, and dominates that IoT market [41].

3.6.1 μClinux

The open source nature of Linux made it possible to fork the source code and modify it according to ones needs. One such project is the μClinux, which was specifically created to target MMU-les microcontrollers. Its hardware dependent, such as physical memory, and independent code, such as virtual memory, are distinct. Using the given instructions, the hardware-specific portion may be altered for a number of CPUs, hence the OS is modifiable. The system supports both user-space and kernel-space, and switching between the

two may be done using system calls. It is possible to develop in a multi-threading environment using POSIX thread libraries. Neither a virtual memory model nor an memory protection unit exist, but functions can be used to dynamically allocate memory, hence DLL is possible. It features a complete TCP/IP stack that may be swapped out for a lighter stack like uIP or lwIP [31]. But, in comparison to other IoT edge device OSs, as seen in section 2.5, μClinux has a far larger footprint than other IoT OSes [32]. μClinux was eventually discontinued as a standalone fork and was reintroduced into the mainline Linux kernel. With the official emailing list gone quiet and its webpage only visible through web archives, and the last official update publish in May of 2016 [26]. Other entities appear to have forked and maintained it further down the line, such as emcraft [4, 7], with last comits on December 2017, one year later. The last remaining verifiable remnants appear to be pointing towards a "small C library for developing embedded Linux systems" called uclibc-ng [25], which could prove useful.

3.7 U-Boot

"Das U-Boot" is an open source boot loader used predominantly in embedded devices. It's main focus is to load the OS kernel into main memory. It supports a wide variety of IoT development boards [22]. Yet again, as is common in the IoT ecosystem, there is a multitude of U-Boot forks, the original and the one that appears to be maintained and updated most frequently is by denx, while the previously mentioned emcraft has their own [5].

When not otherwise mentioned, when U-Boot is mentioned we are refering to U-Boot maintained by denx [24].

3.8 QEMU

Primarily a general-purpose machine virtualizer and emulator, QEMU has a variety of applications. In this thesis we will use it to emulate a system, thus creating a virtual replica of a MCU, including the CPU, memory, and simulated peripherals, in order to run a compiled version of Linux. The CPU may operate in this mode entirely emulated or in conjunction with a hypervisor like KVM, Xen, Hax, or Hypervisor. Equivalently to the cross-compilation process that was dicussed in section 3.4, the "user mode emulation," allows QEMU to run programs that were built for the target CPU, on our host CPU [15].

Chapter 4

Standarizing the IoT OSs with Linux

In this chapter we propose a way towards standardizing the heterogenious IoT edge device ecosystem with Linux. While not an easy task and coming with many stepping stones on the way, aswell as drawbacks, there certainly are a multitude of benefits that such a consensus would bring.

4.1 Proposal

As the layer between the user and the hardware, an OS provides an interface with which the former can input data, perform calculations and view the output, it works as a standarized layer. An Interface can be implemented as a graphical user interface (GUI), or a command line interface (CLI). With GUI's having much highter memory requirements, the CLI, such as Linux's Bourne Shell `sh` should provide a lightweight fit for the IoT edge device ecosystem. When it comes to OS choice in IoT, the traditional approach is to choose an real-time operating system (RTOS). With their lightweight While this solution solves many problems, such as concurrency and multi-threading. With increasing computing capabilites, MCUs have surpased the capabilities of the first PCs running the first operating systems such as UNIX and Linux. To aid the standardization process of IoT this thesis proposes the use of the Linux kernel as a primary OS on edge devices. With projects such as uClinux and Buildroot, with its support for MMU-less devices, this appears feasible.

[Include Graphic that Shows: MCU -> Linux Kernel -> Applications]

4.2 The benefits and drawbacks of Linux on MCUs

Benefits

With a monolithic kernel architecture, as is the case the Linux, module interaction costs are lower. Furthermore, performance is improved since, unlike in the case of microkernel, control is not transmitted between the kernel and user space [32].

Having standardized the OS layer, applications become much more portable and thus it provides the comunity with much flexibility. Shown in section **??**, any programm correctly compiled will be able to run on the target system. Furthermore, very interesting use cases arise such as the use of container technology, or just namespaces, increasing portability even further.

With the help of the open-source comunity entry barriers are lowered by a significant amount, thus future development of IoT edge device related applications are significantly easier.

In contrast to using MPUs to operate Linux, energy efficieny is much higher, thus fitting the role of IoT better.

Drawbacks

One might argue that for slightly higher price, development board powered by a Cortex A-* MPU, such as the RPI family, are available. These boards could fill the same purpose and have higher computing power, with more RAM and ROM. When operating Linux on such small memory, compared to the capabilites of RPI for example, the OS could take up a large proportions of the memory, thus limiting the available space for the applications that need to run on the MCU.

JuiceVM printk() of 1 character for 5 seconds.

Chapter 5

Implementation

In this section the general approach is described, aswell as trial and error. One must keep in mind that steps have been boiled down to the most notable. Also all the different tools that were used where expanded as time passed, certainly not all packages and software are required for all steps but where helpfull in building, compiling, flashing and auxillary tasks.

5.1 Work Environment

To provide a portable environment where source code, different tools and toolchains can be placed, a docker container was created. The main advantages of Docker are that tools such as Buildroot 3.4.2, qemu, toolchains, etc are available on Linux. With the additional benefit of the possibilty to automate large parts of the compilation processes through shell scripts, and potentially out sourcing expensiv compilation, in the absence of capable local hardware, to the cloud, Docker seems to be a good fit.

To build a Docker container from an image the Dockerfile is specified in [.....]. The official Ubuntu 20.04 long time support (LTS) baseimage is used. This provides a solid OS with many tools that facilitate aquiering packages with the included packet manager `apt-get`.

A lot of different To install debendencies the installscript.sh is run automatically when building the image. This script includes a variety of different packages that will be required. A set of basic tools enables working inside the container on a CLI, a collection of the most common and broadly used untilies are shown in listing 5.1. We use `git` and `wget` to aquire repositories and files from the internet, a CLI text editor for `.config` files, and

Listing 5.1: Installing basic utility

```
apt-get install -y git \
wget \
bc \
nano \
curl \
cpio \
unzip \
rsync
```

compressing and uncompressing utility. Any scripts that are displayed, unless specifically stated otherwise, work with the `bash` shell.

To be able to cross-compile source code to architectures specific binaries for the target device we require further packages, all of which are available on Ubuntu's package manager `apt-get`.

5.2 Sample Code

To establish a baseline with less complex code compared to the massive source code repepository of the Linux kernel, and all its forks, that can fully be understood, a collection of sample code was found [16]. These three code snippets, in combination with QEMU as an emulator for target platform, will provide a controlled environment, in which it is easier to verify if compilation was successful and correct, if the binaries works on target platform, or if it was correctly linked. By having such a fall back option it saves time and power, and provides a solid foundation of the complex processes that occur during cross-compilation.

`notmain.c`, as seen in listing 5.3, writen in the C programming language represents the kernel, which contains the basic loop, with the task of printing the numbers 0 to 7 onto some universal asynchronous receiver-transmitter (UART). Theoretically, if the UART is connected to some console, the output would look something like this:

```
01234567
```

To compile this sample code for target platfrom the commands shown in listing 5.6 are issued through the command line shell `bash`. Initially starting with three files `flash.s`, `notmain.c` and `flash.ld`, we are now presented with 5 additional files. `flash.o`, `notmain.o`, `notmain.elf`, `notmain.list` and most importantly `notmain.bin`, the "sample

Listing 5.2: Installing toolchains and dependencies

```
apt-get install -y gcc \
binutils-arm-none-eabi \
make \
gcc-arm-none-eabi \
gcc-arm-linux-gnueabihf \
gcc-arm-linux-gnueabi \
libncurses-dev \
libncurses5-dev \
flex \
bison \
openssl \
libssl-dev \
dkms \
perl \
libelf-dev \
libudev-dev \
libpci-dev \
libiberty-dev \
autoconf \
lzop
```

Listing 5.3: notmain.c

```
void PUT32 ( unsigned int, unsigned int );
#define UART0BASE 0x4000C000
int notmain ( void )
{
    unsigned int rx;
    for(rx=0;rx<8;rx++)
    {
        PUT32(UART0BASE+0x00,0x30+(rx&7));
    }
    return(0);
}
```

Listing 5.4: flash.s

```
1  .thumb
2  .thumb_func
3  .global _start
4  _start:
5  stacktop: .word 0x20001000
6  .word reset
7  .word hang
8
9  .thumb_func
10 reset:
11     bl notmain
12     b hang
13
14 .thumb_func
15 hang:   b .
16
17 .thumb_func
18 .globl PUT32
19 PUT32:
20     str r1,[r0]
21     bx lr
```

Listing 5.5: flash.ld

```
ENTRY(_start)

MEMORY
{
    rom : ORIGIN = 0x00000000, LENGTH = 0x1000
    ram : ORIGIN = 0x20000000, LENGTH = 0x1000
}

SECTIONS
{
    .text : { *(.text*) } > rom
    .rodata : { *(.rodata*) } > rom
    .bss : { *(.bss*) } > ram
}
```

Listing 5.6: Compiling sample code with gcc-arm-none-eabi toolchain

```
1 arm-none-eabi-as --warn --fatal-warnings -mcpu=cortex-m4 flash.s -o flash
2 arm-none-eabi-gcc -Wall -O2 -ffreestanding -mcpu=cortex-m4 -mthumb -c not
3 arm-none-eabi-ld -nostdlib -nostartfiles -T flash.ld flash.o notmain.o -o
4 arm-none-eabi-objdump -D notmain.elf > notmain.list
5 arm-none-eabi-objcopy -O binary notmain.elf notmain.bin
```

kernel". From listing 5.6 we can also see the flag `-mcpu=cortex-m4`, symbolizing that we are compiling for target platfrom ARM Cortex-M4.

5.3 Compilation

5.3.1 U-Boot

As a first step, we will compile U-Boot for QEMU, so that we can continue evaluating, without needing to flash onto an MCU every time. If not previously done so, we need to clone the U-Boot repository, and navigate inside the freshly cloned folder. Using our previously established toolchain, `gcc-arm-none-eabi`, we call the commands, shown in listing 5.13, inside of the current folder. At line 2 of listing 5.13, `qemu_arm_defconfig`, is mentioned after having declared the toolchain. This config is simply a pre-set configuration file that is provided by U-Boot, for specific usecases. After compilation has completed `u-boot.bin` is the output of our compilation.

Listing 5.7: Compiling U-Boot for QEMU

```
1 make mrproper
2 make ARCH=arm CROSS_COMPILE=arm-none-eabi- qemu_arm_defconfig
3 make ARCH=arm CROSS_COMPILE=arm-none-eabi-
```

Listing 5.8: Compiling U-Boot for QEMU

```
1 make stm32f469_disco_sd_defconfig
2 make
```

5.3.2 Mainline Linux kernel

[I did this but its pretty pointless, idk. the explorative nature of this thesis had me running against a lot of walls and IDK how to show that. I mean with buildroot, that

was found lateron in the thesis, things work easier, but this work here seems to be for nothing.]

5.3.3 μClinux

[I did this but its pretty pointless, idk. the explorative nature of this thesis had me running against a lot of walls and IDK how to show that. I mean with buildroot, that was found lateron in the thesis, things work easier, but this work here seems to be for nothing.]

5.3.4 Buildroot

We need to download and navigate to the buildroot directory. In the documentation of Buildroot, under 2.1. "Mandatory packages", a list of required dependencies for the compilation process are shown, we have installed some of these packages in section 5.1, and others come with the Ubuntu 20.04 LTS distribution. We are presented with the build root directory shown in listing 5.9. `configs` holds all the pre-set configuration files that can be used to compile Linux, by navigating to the directory we see all the configs that can be used, and lateron modified. Sadly STM32L476G-Eval is not included, but other configurations for STM are available, such as `stm32f429_disco_xip_defconfig` for the STM32F429-Discovery development board. `make stm32f429_disco_xip_defconfig` will copy the code from the specified configuration into the `.config` file, which is not visible in the thee unless command `ls -a` is issued.

Listing 5.9: Buildroot directory

```
buildroot/
|
|-- arch
|-- board
|-- boot
|-- configs
|-- docs
|-- linux
|-- package
|-- support
|-- system
|-- toolchain
```

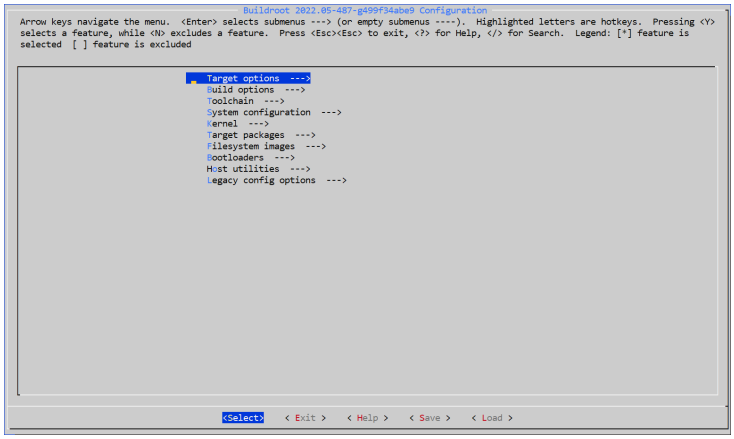

Figure 5.1: `make menuconfig` inside Buildroot directory

```
|-- utils
```

Seen in 5.1, by typing `make menuconfig` will open a visual menu inside the console that can help us visually navigate through the `.config` file that was previously set. Note that for this visual editor the `libncurses5-dev` package is required and was installed in section 5.1.

With `make`, the compilation process starts. Depending on the local maschine that is performing the process, this process can be very time and resource intensive. Once complete, the outputs will appear in a new folder inside the buildroot directory under `output`, the contents are shown in listing 5.10.

Listing 5.10: Buildroot output directory

```
output/
|
|-- build
|-- host
|-- images
|-- staging
|-- traget
```

Listing 5.11: Compiling

```
1 make stm32f469_disco_sd_defconfig
2 make
```

5.4 Proof of Concept

asdfasdf

Listing 5.12: Installing QEMU in the container

```
printf 'y\n8\n7\n' | apt-get install -y qemu-system-arm
```

asdfasdf

Listing 5.13: Running U-Boot with QEMU

```
1 qemu-system-arm -machine virt -nographic -bios u-boot.bin
```

5.4.1 Setting up μClinux

[maybe irrelevant at this point]

Since the original μClinux is not maintained anymore, or rather included in the mainline kernel, and then discontinued. The task of finding an entitiy that has maintained a μClinux fork was a daunting task. Emcraft Furthermore, scipts that extend this container with additional functionality such as μClinux and compatible toolchains [7, 5], and Buildroot are provided in init-uClinux.sh and init-Buildroot.sh respectively.

5.4.2 Setting up Buildroot

[maybe irrelevant at this point]

5.5 Flashing binaries

5.6 JuiceVM

JuiceVM printk() of 1 character for 5 seconds.

Chapter 6

Evaluation

[I AM NOT SURE WHAT TO EVALUATE]

6.1 QEMU as an evaulation tool

6.2 JuiceVM

6.3 U-Boot

6.4 Mainline Linux kernel

6.5 uClinux

6.6 Buildroot

Chapter 7

Summary and Conclusions

This thesis summarizes IoT hardware and its limitations, explores the different OS solutions, shows proof of concept methodologies using QEMU, and proposes a standardizes OS for IoT edge devices, the Linux kernel, with a plethora of tools that facilitate cross compilation and boot loading. possible tools used to succesfully standardize MCU's on an OS level. It also shows the challenges that are faced, when trying to homogenize such a heterogeneous and interconnected field such as IoT.

This thesis provides a modular refined toolchain container for diverse uses for embedded Linux, with defined structure, that can be used to cross-compile Linux kernel source code and evaluate its correctness.

QEMU is not the most reliable.

7.1 Future Work

Bibliography

[1] "Arm Cortex-A series processors," last-accessed: Jul 14, 2022. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-a/

[2] "Arm Cortex-M series processors," last-accessed: Jul 14, 2022. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m/

[3] "Buildroot Documentation," last-accessed: Jul 15, 2022. [Online]. Available: https://buildroot.org/downloads/manual/manual.html

[4] "EmCraft systems," last-accessed: Jul 15, 2022. [Online]. Available: https://emcraft.com/

[5] "Emcraftsystems/u-boot," last-accessed: Jul 18, 2022. [Online]. Available: https://github.com/EmcraftSystems/u-boot

[6] "GNU Arm Embedded Toolchain Downloads," last-accessed: Jul 15, 2022. [Online]. Available: https://developer.arm.com/downloads/-/gnu-rm

[7] "https://github.com/emcraftsystems/linux-emcraft," last-accessed: Jul 18, 2022. [Online]. Available: https://www.segger.com/evaluate-our-software/st-microelectronics/st-stm32l476g-eval/#gallery

[8] "index : kernel/git/torvalds/linux.git," last-accessed: Jul 18, 2022. [Online]. Available: https://www.segger.com/evaluate-our-software/st-microelectronics/st-stm32l476g-eval/#gallery

[9] "index : kernel/git/torvalds/linux.git," last-accessed: Jul 14, 2022. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/LICENSES/preferred/GPL-2.0

[10] "Juice VM a Small RISC-V Virtual Machine," last accessed: Jan 31, 2022. [Online]. Available: https://github.com/juiceRv/JuiceVm

[11] "The linux kernel cpu architectures," last accessed: Jan 31, 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/arch.html

[12] "Memory Management: Concepts overview," last-accessed: Jul 15, 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html

[13] "Memory Management: No-MMU memory mapping support," last-accessed: Jul 15, 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/mm/nommu-mmap.html

[14] "OpenWRT," last-accessed: Jul 15, 2022. [Online]. Available: https://openwrt.org/

[15] "QEMU," last-accessed: Jul 14, 2022. [Online]. Available: https://www.qemu.org/docs/master/about/index.html

[16] "qemu-arm with cortex-m4 on linux," last-accessed: Jul 14, 2022. [Online]. Available: https://stackoverflow.com/questions/67591930/qemu-arm-with-cortex-m4-on-linux

[17] "Raspberry Pi 4," last-accessed: Jul 14, 2022. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[18] "Raspberry Pi OS," last-accessed: Jul 14, 2022. [Online]. Available: https://www.raspberrypi.com/software/

[19] "Shared Libraries without an MMU," last-accessed: Jul 15, 2022. [Online]. Available: http://xflat.sourceforge.net/NoMMUSharedLibs.html

[20] "State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally," last-accessed: Jan 31, 2022. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[21] "State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally," last-accessed: Jan 31, 2022. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[22] "The U-Boot Documentation," last-accessed: Jul 14, 2022. [Online]. Available: https://u-boot.readthedocs.io/en/latest/index.html

[23] "torvalds/linuxt," last-accessed: Jul 15, 2022. [Online]. Available: https://github.com/torvalds/linux

[24] "u-boot," last-accessed: Jul 14, 2022. [Online]. Available: https://github.com/u-boot/u-boot

[25] "uClibc-ng - Embedded C library," last-accessed: Jul 20, 2022. [Online]. Available: https://uclibc-ng.org/

[26] "uclinux.org," last-accessed: Jul 15, 2022. [Online]. Available: https://web.archive.org/web/20181113230737/http://www.uclinux.org/

[27] "Yocto Project: Release 2.1," last-accessed: Jul 15, 2022. [Online]. Available: https://docs.yoctoproject.org/migration-guides/migration-2.1.html?highlight=mmu

[28] S. A. Al-Qaseemi, H. A. Almulhim, M. F. Almulhim, and S. R. Chaudhry, "Iot architecture challenges and issues: Lack of standardization," in *2016 Future technologies conference (FTC)*. IEEE, 2016, pp. 731–738.

[29] I. Arikpo, F. Ogban, and I. Eteng, "Von neumann architecture and modern computers," *Global Journal of Mathematical Sciences*, vol. 6, no. 2, pp. 97–103, 2007.

[30] A. Banafa, "Iot standardization and implementation challenges," *IEEE internet of things newsletter*, pp. 1–10, 2016.

[31] A. Dunkels, "Full {TCP/IP} for 8-bit architectures," in *First International Conference on Mobile Systems, Applications, and Services (MobiSys2003)*, 2003.

[32] P. Gaur and M. P. Tahiliani, "Operating systems for iot devices: A critical survey," in *2015 IEEE region 10 symposium*. IEEE, 2015, pp. 33–36.

[33] D. Georgiev, "Internet of things statistics, facts & predictions [2022's update]," last-accessed: July 09, 2022. [Online]. Available: https://review42.com/resources/internet-of-things-stats/

[34] W. H. Hassan *et al.*, "Current research on internet of things (iot) security: A survey," *Computer networks*, vol. 148, pp. 283–294, 2019.

[35] G. D. Maayan, "The IoT Rundown For 2020: Stats, Risks, and Solutions," last-accessed: July 09, 2022. [Online]. Available: https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx?Page=2

[36] A. Maier, A. Sharp, and Y. Vagapov, "Comparative Analysis and practical implementation of the ESP32 microcontroller module for the internet of things," in *2017 Internet Technologies and Applications (ITA)*, 2017, pp. 143–148.

[37] A. Milinković, S. Milinković, and L. Lazić, "Choosing the right RTOS for IoT platform," *Infoteh Jahorina*, vol. 14, pp. 504–9, 2015.

[38] J. Newman, "Why the internet of things might never speak a common language," *App Economy, Fast Company*, 2016.

[39] Z. Peterson, "Using an MCU vs. MPU in Embedded Systems Design," last-accessed: Jul 14, 2022. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[40] J. Romkey, "Toast of the iot: The 1990 interop internet toaster," *IEEE Consumer Electronics Magazine*, vol. 6, no. 1, pp. 116–119, 2017.

[41] 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)*. IEEE, 2017, pp. 369–375.

[42] E. Schiller, A. Aidoo, J. Fuhrer, J. Stahl, M. ZiÃ¶rjen, and B. Stiller, "Landscape of IoT security," *Computer Science Review*, vol. 44, p. 100467, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013722000120

[43] P. Sethi and S. R. Sarangi, "Internet of things: architectures, protocols, and applications," *Journal of Electrical and Computer Engineering*, vol. 2017, 2017.

[44] D. Shin, "A socio-technical framework for internet-of-things design: A human-centered design for the internet of things," *Telematics and Informatics*, vol. 31, no. 4, pp. 519–531, 2014.

[45] P. H. Tarange, R. G. Mevekari, and P. A. Shinde, "Web based automatic irrigation system using wireless sensor network and embedded linux board," in *2015 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2015]*. IEEE, 2015, pp. 1–5.

[46] S. Thornton, "Microcontrollers vs. Microprocessors: What's the difference?" last-accessed: Jul 14, 2022. [Online]. Available: https://www.microcontrollertips.com/microcontrollers-vs-microprocessors-whats-difference/

[47] V. L. Voydock and S. T. Kent, "Security mechanisms in high-level network protocols," *ACM Computing Surveys (CSUR)*, vol. 15, no. 2, pp. 135–171, 1983.

[48] P. Wang, R. Valerdi, S. Zhou, and L. Li, "Introduction: Advances in iot research and applications," *Information Systems Frontiers*, vol. 17, no. 2, pp. 239–241, 2015.

Abbreviations

IoT	Internet-of-Thingsi
RPI	Raspberry Pi
ARM	Advanced RISC Machines
CPU	Core processing unit
RAM	Random access memory
OS	Operating System
USB	Universal Serial Bus
Wi-Fi	Wireless Fidelity
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
PSRAM	Pseudo Static RAM
ISA	Instruction Set Architecture
TEE	Trusted Execution Environment
RFID	Network of Radio-Frequency Identification
I/O	Input/Output
SoC	System on a Chip
SoM	System on a Module
IoT	Internet of Things
PCB	Print-Circuit board
RAM	Random Access Memory
SRAM	
ULP	Ultra Low Power
OSS	Open Source Software
CPU	Core Processing Unit
CU	Controll Unit
ALU	Arithmetic Logic Unit
MCU	Microcontroller unit
MPU	Micro-processing unit
DLL	Dynamically Linked Libraries

SLL	Statically Linked Libraries
GUI	Graphical User Interface
CLI	Command Line Interface
RTOS	Real-Time Operating System
UART	Universal Asynchronous Receiver-Transmitter

Glossary

Authentication

Authorization Authorization is the decision whether an entity is allowed to perform a particular action or not, e.g. whether a user is allowed to attach to a network or not.

Accounting

List of Figures

List of Tables

Appendix A

Installation Guidelines

Appendix B

Contents of the CD