



University of
Zurich^{UZH}

Linux on Tiny Microcontrollers

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

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

Abstract

Während das Internet der Dinge (Internet of Things, IoT) in Bezug auf die Anzahl der Geräte und die Anwendungsfälle ein schnelles Wachstum verzeichnet, mangelt es in diesem Bereich stark an Standardisierung. Mit heterogenen Edge-Geräten, sogenannten Mikrocontrollern (MCUs), und einer Vielzahl von Betriebssystemen (OS) zur Auswahl, leidet die Interoperabilität. Das Ziel dieser Arbeit ist es, den Kernel des Open-Source-Betriebssystems Linux auf kleine MCUs zu portieren. Auf diese Weise wird die Hardware von der Anwendungsschicht abstrahiert und somit die dringend benötigte Standardisierung im IoT-Ökosystem geschaffen. Dies wurde erreicht, indem die richtige Toolchain gefunden und der Linux- und μ Clinux-Kernel mit Hilfe von Tools wie Buildroot kompiliert wurde. Anschliessend wurden die kompilierten Distributionen mit QEMU getestet und auf das STM32L476G-Eval Board bzw. ESP-EYE portiert. Darüber hinaus wurde ein anderer Ansatz mit JuiceVM, einer virtuellen RISC-V-Maschine, auf der Linux läuft, erprobt.

As the Internet-of-Things (IoT) see rapid growth, in device numbers and use cases, standardization is very much lacking in the field. With heterogeneous edge devices, or so-called microcontrollers (MCUs), and a variety of operating systems (OS) to choose from, interoperability is suffering. The goal of this thesis is to port the kernel of the open-source operating system Linux onto tiny MCUs. By doing so abstracting the hardware from the application layer, and therefore providing much-needed standardization in the IoT ecosystem. This was achieved by finding the correct toolchain, and compiling the Linux and μ Clinux kernel with the help of tools such as Buildroot. Subsequently, the compiled distributions were tested with QEMU and ported to STM32L476G-Eval board and ESP-EYE respectively. Additionally, a different approach with JuiceVM, a RISC-V virtual machine, running Linux was explored.

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

Abstract	i
Acknowledgments	iii
1 Introduction	3
1.1 Motivation	3
1.2 Description of Work	5
1.3 Thesis Outline	5
2 Related Work	7
2.1 The Internet-of-Things	7
2.2 IoT Architecture	8
2.3 The lack of standardization	9
2.4 IoT Security Issues	10
2.5 Operating Systems on IoT	10
3 Overview of Hardware & Technologies	13
3.1 Micro Computing	13
3.2 Processors	14
3.3 Market Analysis	15
3.4 Toolchains, Cross-Compilation & Libraries	17

3.4.1	Dynamically & Statically Linked Libraries	19
3.4.2	Buildroot	19
3.4.3	Yocto Project & OpenWrt	20
3.5	Memory Management Unit	20
3.6	The Linux Kernel	20
3.6.1	μ Clinux	21
3.6.2	Device Trees	21
3.7	U-Boot	21
3.8	QEMU	22
3.9	JuiceVM	22
3.10	STM Tools	22
3.11	Miscelanous Tools	23
4	Standarizing the IoT OSs with Linux	25
4.1	Proposal	25
4.2	The benefits and drawbacks of Linux on MCUs	26
5	Implementation	29
5.1	Work Environment - Docker container	29
5.1.1	Downloading and setting up μ Clinux	30
5.2	Sample Code	32
5.3	FreeRTOS on STM32L4	32

<i>CONTENTS</i>	1
6 Evaluation	37
6.1 Compilation	37
6.1.1 Sample Code	37
6.1.2 U-Boot	38
6.1.3 Mainline Linux kernel	38
6.1.4 μ Clinux	40
6.1.5 Buildroot	40
6.2 Emulating with QEMU	42
6.2.1 Sample Code	42
6.2.2 U-Boot	43
6.2.3 Buildroot	44
6.3 Flashing binaries	45
6.3.1 FreeRTOS on STM32L4	45
6.3.2 JuiceVM	46
6.3.3 Buildroot	48
7 Summary, Conclusions & Future Work	53
Abbreviations	61
Glossary	63
List of Figures	63
List of Tables	66
A Installation Guidelines	69
B Contents of the CD	71

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 [50]. 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 [21] but also due to the diversity and variety of IoT solutions [33]. 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 operation 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 megabyte (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.

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 and to provide a Linux distribution for future studies, which people can use, and upon which they can build. As the first step, a market analysis must be performed to find suitable MCU boards that can support Linux. Then correct toolchains and libraries need to be found to successfully compile Linux targeted at these tiny edge devices. Having compiled the Linux source code, correctness needs to be verified using QEMU, and as the last step flashing the code onto the chosen device.

1.3 Thesis Outline

This thesis is segmented into seven chapters. Related Work in Section 2 provides a description of IoT architectures, security, operating systems, and the lack of standardization. In Section 3 we establish what devices this thesis considers, perform a market analysis, and introduce various tools and projects that were used within the thesis. Section 4 proposes a way towards standardization, and outlines the goal. In Section 5 we set up an environment with the tools discussed in Section 3, and sample code, that helps us to evaluate, is introduced and explained. Finally, in Section 6 we evaluate the tools and projects, with a conclusion following in Section 7.

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 MCUs 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 keyboards. While these MCUs build the foundation of the Internet of Things (IoT), the edge hardware, the first primitive IoT device, a toaster that can be turned on and off remotely, was introduced in 1990 [48].

The phrase "Internet of Things", was initially coined by Kevin Ashton in 1999. Ashton made the initial proposal for the 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 envisions a system in which omnipresent and ubiquitous devices will link the Internet to every physical object [52, 56].

The quantity and diversity of IoT devices and solutions have multiplied due to the market's 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 [38, 44]. 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.

Uses of IoT are numerous, including medical, industrial, and consumer. With Governments heavily investing in 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 [52]. Concrete examples include food safety through tracking of production and supply [39], warning systems for floods [36] and automatic irrigation systems for farms [53].

2.2 IoT Architecture

While the edge devices are in the center of IoT, it can not be reduced to just them. A stable infrastructure is required to, effectively collect and handle data, efficiently organize storage and transportation, ensure connectivity and security, and above all provide a foundation to process ever larger quantities of data [43]. This includes sensing, computing, networking, and the cloud. As every thing becomes interconnected, as IoT envisions, this architecture needs to be able to support the growing number of nodes. With 32-bit addressing, IPv4 is at its limits and can not support more than 4.5 billion devices, new solutions such as IPv6 need to be implemented to identify each device. Although a multitude of models describing such architectures have been proposed, there is no consensus regarding which one will prevail as the standard. The most basic model is the three-layer architecture [51].

1. The **Perception layer**, is the bottom of the pyramid where sensors reside, gathering environmental data and transmitting them to the next, or to other devices in the same layer. Omnipresence and ubiquity, are keywords used in IoT, and it is in the perception layer that these traits are gained. Devices figuratively located in this layer are still required to perform basic computation, yet need to be power efficient, low cost, and relatively small.
2. The **Network layer** is in charge of establishing secure connections with other intelligent objects, network components, and servers. Its capabilities are also employed for processing and transferring sensor data.

3. The **Application layer** delivers application-specific services to the user. It has two fundamental functions, state tracking, and remote control. The status of the sensors and microcontroller can all be monitored via state tracking, and as the name suggests, remote control is in charge of controlling the sensors or MCUs [57].

While this model is extremely basic, it captures the very essence of IoT, and shall suffice for the topics in this thesis, where we mostly focus on devices, MCUs, from the perception layer. Note that microcontrollers are by no means limited to the first layer, and could be deployed in the network layer.

2.3 The lack of standardization

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 individually, from the hardware devices, development environments, and tools, to the relevant cloud services [33]. Additionally, there are many various IoT protocols to choose from, including Wi-Fi, Bluetooth, 6LoWPAN, Zigbee, etc. for communication networks; EPC, uCode, IPv6, and URIs for identification; and MQTT, CoAP, AMQP, Websocket, and Node, etc. for application data protocol. Since there is no established standard, an IoT developer chooses protocols depending on his needs and domain knowledge and designs end-to-end IoT solutions. In an open market, such systems become vendor-specific, which is undesirable [42]. This inevitably has the effect that devices can not talk to each other. [30] conclude that the lack of standardization negatively impacts the IoT industry. [42] lists the advantages of standardization and the disadvantages with the absence thereof. [46] even goes as far as stating that "The Internet of Things Might Never Speak a Common Language".

As interoperability is ensured by standardization, it improves the effective integration and information exchange between distributed systems. The requirement for a standard model to carry out typical IoT backend functions, such as processing, storing, and firmware upgrades, is growing in importance. Different IoT solutions are expected to cooperate with shared backend services in this new architecture, which will provide levels of compatibility, portability, and management that are almost unattainable with the current generation of IoT systems [33].

2.4 IoT Security Issues

The term IT-Security was defined as follows. Information integrity, availability and confidentiality [55]. In a paper by Schiller et al. [50] security is defined as message confidentiality. This is exclusively the case if, and only if, the sender and receiver are aware of the existence of the message, and only they can verify its validity. First and foremost, compromised IoT systems have the potential to damage consumers physically as well as compromise their privacy when sensors, actuators, or other linked devices are utilized maliciously. This is, even more, the case in applications in the medical field. Secondly, an attack's effects extend beyond a single device or network due to the strong interconnectedness of IoT devices. The IoT network is only as secure as its weakest device, hence the proverb "A chain is only as strong as its weakest link" is entirely appropriate in this context.

The security strategies and procedures suggested are mostly based on traditional network security procedures. However, given the variety of the devices and protocols including the number of nodes in the network, implementing security methods in an IoT system is more difficult than with a typical network [40].

2.5 Operating Systems on IoT

[49] compile quantitative survey results of OS usage in IoT. Of which Linux takes up more than 70% of IoT Devices, while the majority of other OSs are used less than 10%. As established in Section 2.2, low-end devices don't represent the entirety of IoT, but a portion that resides at the lowest layer. Therefore, this overwhelming majority does not reflect the status quo for perception layer devices, rather gateway devices on the network layer. The low representation of other OSs can be explained by the various choices that developers can make when choosing an OS, once again underlining the heterogeneous nature of IoT.

[37] classify OSs into three different architectures, monolithic, layered, and modular microkernel. Furthermore, they specify advantages of these architectures, the monolithic kernel having a smaller footprint on memory with improved module interaction, the modular kernel not requiring to restart of the system when a single module fails, and the layered architecture sitting in between the two. Theoretically, the lower memory print for monolithic kernels is true, in practice, this is hardly the case for the Linux kernel due to its sheer size.

[45] analyzed different OSs for IoT edge devices and provide some factors that are crucial for choosing such an operating system. They go as far as stating that "Linux will never run on these chips", by chips referring to ARM Cortex-M powered MCUs. This claim is based on the premise that the onboard memory of such chips is too low to support the OS. As seen in Table 2.1, they show key characteristics of the mentioned OSs and Linux has by far the largest RAM and ROM footprint.

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 [32]

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 ISAs, System-on-a-Chip manufacturers and their relevant products, and open source projects with the aim to aid in the process of embedding, toolchains, and technologies. This chapter serves as an introductory overview of the various devices and tools that were discovered and may help further research aiming 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 [31]. Figure 3.1 shows a visual representation.

- A **Processor Unit**, or a **core processing unit (CPU)** can further be divided into two subcomponents, the Control 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 Architecture to further categorize two IoT edge devices:

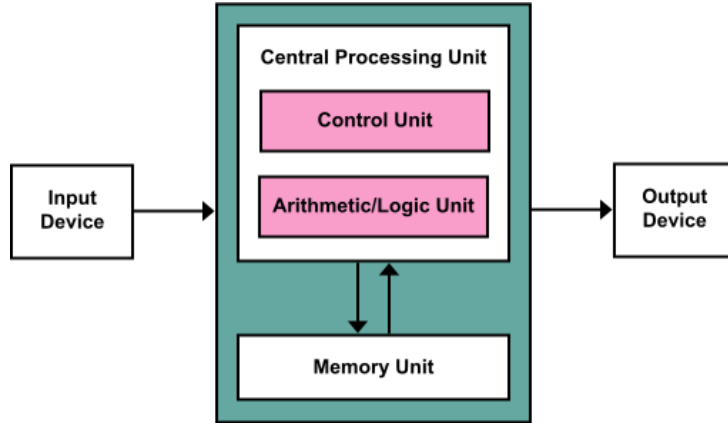


Figure 3.1: Von Neumann Architecture [8]

- A **Micro controller unit (MCU)** is an integrated, fully capable, self-sufficient computer on a single chip. It can run a "bare metal interface", meaning that it doesn't require running an OS. Without which, it can run a single thread, or a control 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 Von Neumann's Architecture, is just the processor unit.

While the above terms, MCU and MPU, are oftentimes used interchangeably, we want to point out the differences between these two families. It is far easier to run an OS, such as the Linux kernel [19], on MPU-enabled devices, such as the Raspberry Pi 4, because it is not limited to the capabilities on the chip, and can easily be extended with, i.e. more RAM [18]. An MCU is limited to the design and capabilities of the chip. Hence, an 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 [47, 54].

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**, are 32-bit, energy-efficient, high-performance processors that build on the modular RISC architecture [23]. These processors are found on MCUs.

As the description of the ARM Cortex-M and Tensilica Xtensa processors seems to fit our premise perfectly, low-cost, power-efficient and embedded in MCUs, we will focus on these types of CPUs.

3.3 Market Analysis

To find a suitable MCU that could support Linux, a market analysis had to be performed. The most relevant criteria were the following. A cheap MCU, having sizable RAM preferably more than 1MB, more than 40MHz CPU clock rate, availability in the region, and a large community. Having a sizable community can simplify development due to the availability of online resources. Furthermore with the heterogeneous nature of IoT, directing this thesis towards a smaller target audience would inevitably decrease its value. In Table 3.1 the results are shown. Due to the countless different boards and chips that are available on the market, we grouped the chips into families defined by the manufacturers, to capture a larger picture, the column "Chip / Dev Board" reflects this. The chips were categorized into the IoT families established in Section 3.1.

The Raspberry Pi modules were used as a reference, as running Linux on these boards is possible and fairly streamlined, through their own Linux distribution Raspbian [19]. Yet, with their more capable Cortex-A processors and much larger RAM, they did not fit the criteria for this thesis.

With the exception of ESP-EYE, 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 manufacturers primarily of STM, were contacted. Among others, Digi-Key Electronics, Anatec AG, Avnet Silica Rothrist, Mouser Electronics, and STM personally. Questions concerning configurability and extensibility of RAM were posed with the goal of gaining expert insight. They were also specifically

Manufacturer	Chip/Dev Board	Classification	CPU	bit	Clock rate	RAM
Raspberry Pi	Raspberry Pi 4 B	MPU	ARM Cortex-A72	64	1.5 GHz	2-8 GB
Raspberry Pi	Zero 2 W	MPU	ARM Cortex-A53	64	1 GHz	512 MB
Raspberry Pi	Pico	MCU	ARM Cortex-M0+	32	133 MHz	264 kB
Espressif	ESP32	MCU	Tensilica Xtensa	32	240 MHz	520 KB
Espressif	ESP32-S2	MCU	Tensilica Xtensa	32	240 MHz	320 KB
Espressif	ESP32-C3	MCU	RISC-V	32	160 MHz	400 KB
Espressif	ESP32-S3	MCU	Tensilica Xtensa	32	240 MHz	512 KB
Espressif	ESP-EYE	MCU	Tensilica Xtensa	32	240 MHz	8 MB
STMicroelectronics	STM32F0	MCU	ARM Cortex-M0	32	48 MHz	4-32 KB
STMicroelectronics	STM32F1	MCU	ARM Cortex-M3	32	24-72 MHz	4-80 KB
STMicroelectronics	STM32F3	MCU	ARM Cortex-M4	32	72 MHz	16-80 KB
STMicroelectronics	STM32F4	MCU	ARM Cortex-M4	32	84-180 MHz	32-384 KB
STMicroelectronics	STM32G0	MCU	ARM Cortex-M0+	32	64 MHz	144 KB
STMicroelectronics	STM32G4	MCU	ARM Cortex-M4	32	170 MHz	128 KB
STMicroelectronics	STM32L4	MCU	ARM Cortex-M4	32	80 MHz	40-320 KB

Table 3.1: Market analysis of available IoT edge devices

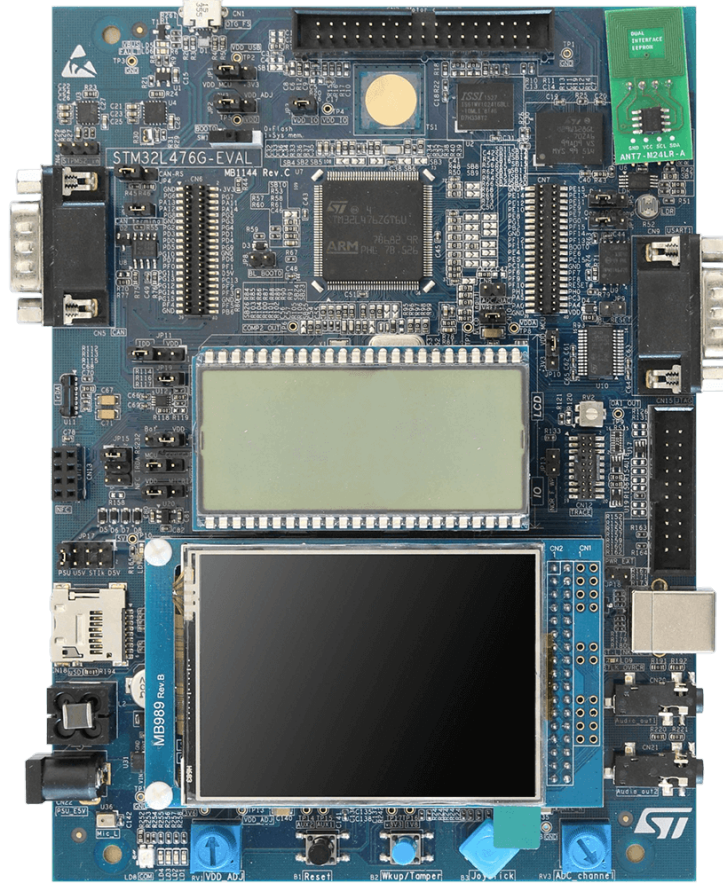


Figure 3.2: STM32L476G-Eval development board [11]

asked about boards such as the STM32F and STM32G series. STM personally replied with the suggestion that we should try using the STM32L4 series boards and was willing to supply us with two boards visible in Figure 3.2. This board provided a flexible memory controller (FMC) interface and would fit our proceedings.

The second device used in this thesis is the ESP-EYE seen in Figure 3.3. With its inbuilt 2MP camera and its goal to capture and process pictures as well as sound, in comparison to the other boards, it was naturally equipped with a relatively large amount of RAM.

3.4 Toolchains, Cross-Compilation & Libraries

For compiled languages, the appropriately named 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 fulfills 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 that

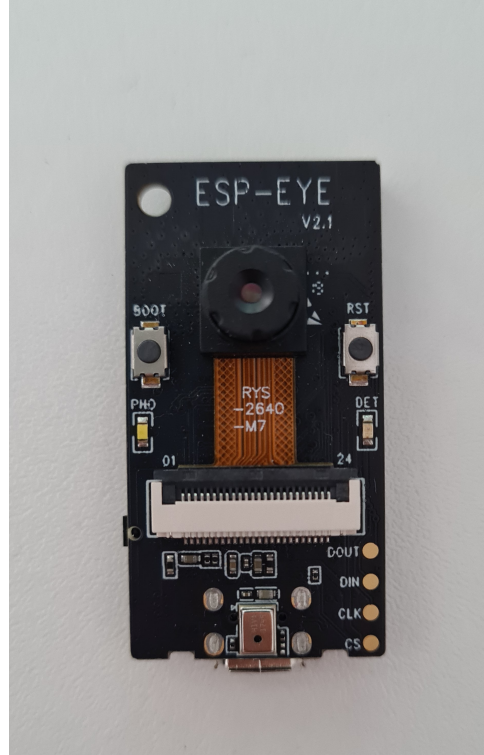


Figure 3.3: ESP-EYE

runs close to the hardware, interpreted languages are not considered. When compiling code a multitude of factors needs to be considered when choosing a toolchain, because they are specifically designed to work exclusively in the right circumstances. Most importantly, whether the host platform is the same as the target platform. The host is the platform on which the compilation takes place and the target on which the binaries will eventually end up running on. If the host and target platform 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 is 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, the MCU edge devices. Broadly simplified toolchains work as follows.

As mentioned, the first step in a toolchain is source code compilation. Depending on the programming language the source code was written in, the compilers 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 [25]. 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, it is a human-readable abstraction of machine code, i.e. bits. Nowadays

Assembler programming is only utilized in situations when extremely efficient management of processor activities is required, but it is still an intermediate product in the compilation. Assembly code needs to be assembled outputting object code. Finally, 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 6.3.

For the goals of this thesis, we will be compiling, mostly, C code on the host platform operating on x64 architecture for target platforms on Armv7E-M architecture. Hence cross-compilation will take place with the `gcc-arm-none-eabi` toolchain [7].

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 holds, 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 executables 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. The process of sharing libraries is aided by a hardware solution, the MMU, see section 3.5.

3.4.2 Buildroot

Buildroot is a tool that makes cross-compiling a complete Linux system for embedded devices easier and more automated. It runs primarily on Linux systems. Through the use of this facilitated toolchain, it creates a self-decompressing version of the Linux kernel `zImage`, a root filesystem, a U-boot bootloader, a 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 Networking, the Yocto Project does not currently support MMU-less builds. [15, 29]. 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 technical terms, the reference points to a virtual memory address that the MMU can translate into the physical memory address. 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 [13, 14]. While the implementation of DLL, for devices that don't contain an MMU device appears to be possible, it once again is very complicated [20]. There exist 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 Torvalds, in 1991 for i386 based PCs. After its initial appearance, it quickly gained traction among developers and was licensed under GNU General Public License (GPL) as free OSS [9]. At the current time, Linux supports all kinds of different target architectures and dominates that IoT market [49].

3.6.1 μ Clinux

The open-source nature of Linux made it possible to fork the source code and modify it according to one's needs. One such project is the μ Clinux, which was specifically created to target MMU-less microcontrollers. Its hardware-dependent, such as physical memory, and independent, such as virtual memory, code is 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 a memory protection unit exists, 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 [35]. But, in comparison to other IoT edge device OSs, as seen in Section 2.5, μ Clinux has a far larger footprint than other IoT OSes [37]. μ Clinux 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 published in May of 2016 [28]. Other entities appear to have forked and maintained it further down the line, such as emcraft [4, 6], with the last commits on December 2017, one year later. The last remaining verifiable remnants appear to be pointing toward a "small C library for developing embedded Linux systems" called uclibc-ng [27], which could prove useful.

3.6.2 Device Trees

The device tree is a data structure that describes the hardware, that is locally available, for the device we want to port Linux to. The device tree is the dynamic solution for the problem that the CPU must know its environment.

3.7 U-Boot

"Das U-Boot" is an open-source boot loader used predominantly in embedded devices. Its main focus is to load the OS kernel into the main memory. It supports a wide variety of IoT development boards [24]. 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 made available by denx, while the previously mentioned

emcraft has their own [5]. When not otherwise mentioned, when U-Boot is mentioned we are referring to U-Boot maintained by denx [26].

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 an 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 Kernel-based Virtual Machine (KVM), Xen, Hax, or Hypervisor. Equivalently to the cross-compilation process that was discussed in section 3.4, the "user mode emulation," allows QEMU to run programs that were built for the target CPU, on our host CPU [16].

3.9 JuiceVM

JuiceVM is a found on GitHub, that started in 2020, with only three visible contributors [10]. Self-described, it is a small RISC-V Virtual Machine (VM), capable of running on just a few hundred KB of RAM. Under its demonstration projects it shows that it can boot Linux, which is why this thesis found interest in the project. While it only provides precompiled binaries, and the source code not being visible to the public, it can still provide the required functionalities that we seek.

3.10 STM Tools

Specifically, two STM provided tools were used in this thesis, STM32CubeProgrammer and STM32CubeIDE 1.9.0. The former provides functionality to connect to the STM boards through the ST-LINK/V2 interface, enables easy flashing capabilities through a GUI, and information on the state RAM and ROM. The latter is an IDE designed for embedded programming that facilitates the creation of projects, the initialization of various libraries, configuration of the chip's I/O peripherals, compilation, and debugging. Additionally, it is possible to flash a connected STM device directly through the IDE.

3.11 Miscelanous Tools

Various different utilities and tools are required that help achieve certain goals, here the most noteworthy are listed. PuTTY is an open-source software developed for Windows OS that, among others, provides SSH and serial port connections. With this tool, a connection to the devices can be established and consoles viewed. Docker is a platform providing virtualization functionalities, it is used to build a Linux-based container, in which the compilations will be executed. By providing great portability it allows future researchers to use the built container. Disk Imager is a small windows utility that enables the flashing of SD cards with `.img` files. `esptool.exe` is a command line utility tool that will allow us to flash ESP32 chips with binaries, mainly the ESP-EYE.

Chapter 4

Standardizing the IoT OSs with Linux

In this chapter, we propose a way toward standardizing the heterogeneous IoT edge device ecosystem with Linux. While not an easy task and with many stepping stones on the way, 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. OSs can be seen as a standardized layer. An interface can be implemented such as a graphical user interface (GUI), or a Command Line Interface (CLI). With GUI's having much higher memory requirements, a CLI 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). While these OSs introduced in Section 2.5, are technically categorized as operating systems, they should rather be seen as frameworks when compared to Linux. Bare metal code can be written within these frameworks which in turn handles low-level implementations, such as threads and message passing [41]. With increasing computing capabilities, MCUs have surpassed 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 their support for MMU-less devices, this appears beneficial and feasible.

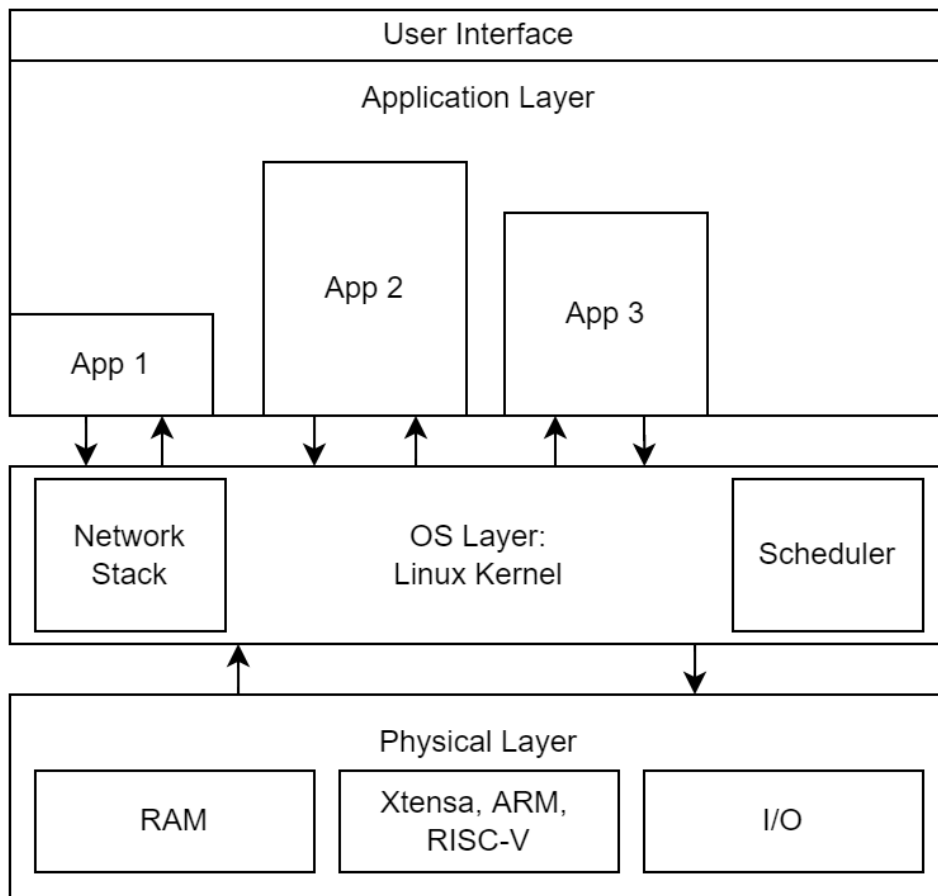


Figure 4.1: Proposed architecture of MCU IoT devices

Illustrated in Figure 4.1, the userspace and the underlying device drivers and hardware are clearly separated from one another. With this level of abstraction, a similar level of user-friendliness, portability, and versatility can be achieved as in laptops and desktops. Precedence for this is the RPI, which has gained widespread use in the IoT industry, partially due to easily supporting Linux.

4.2 The benefits and drawbacks of Linux on MCUs

Benefits

Linux facilitates memory management, protection, and dynamic memory allocation, in section 3.6.1 we established that this is possible without an MMU. RTOSs rely on static memory allocation, which can quickly become a problem as applications grow. Problems such as memory leaks and memory fragmentation can force the system to restart, diminishing its real-time aspect. Linux also provides a network stack that removes the

requirement to program network applications, and thus ensures interoperability. Furthermore, Linux uses a standardized filesystem, that can be used to reliably manage and store data. Broader language support is another benefit, enabling the use of the same programming languages and libraries across all devices. Applications become much more portable and provide the community with much flexibility. Very interesting use cases arise such as the use of container technology, increasing portability even further. With a large open source community at its back, that maintains and updates the source regularly for many architectures, the adoption of MCU IoT devices into the arsenal of Linux, should happen sooner rather than later [41].

Drawbacks

One might argue that for a slightly higher price, development boards powered by a Cortex-A MPUs, such as the RPI family, are available. These boards could fill the same purpose and have higher computing power, with more RAM and ROM, at the cost of energy efficiency, size, and price. When operating Linux on such small memory, compared to the capabilities of RPI for example, the OS takes up a large proportion of the memory, thus limiting the available space for the applications that need to run on the MCU. Another aspect that needs to be overcome is the larger upfront investment when porting Linux to specific IoT devices, yet with projects such as Buildroot, this is well on the way, if not present already.

Chapter 5

Implementation

In this section, the general setup of the environment is described. The final container is made available at [12]

5.1 Work Environment - Docker container

To provide a portable environment where source code, different tools, and toolchains can be placed, a docker container was created. The main advantages of containers are portability and ease of replication. Tools such as Buildroot and QEMU are all available on Linux. Furthermore, containers provide the additional capability to automate large parts of the compilation processes through shell scripts, and potentially out sourcing expensive compilation, in the absence of capable local hardware, to the cloud. By using a Linux OS to run most of the tasks for this thesis, it enforces the claims made in section 4.1, by demonstrating its functionalities and portability.

To build a Docker container, a Dockerfile serves as a template, which is then built into an image and run. The official Ubuntu 20.04 Long Time Support (LTS) base image lays the foundation. This provides a solid OS with many tools that facilitate acquiring packages with the included packet manager `apt-get`.

To install dependencies the `apt-get` command is run, followed by the package that we want to install. A set of basic tools are required for working on a CLI inside the container, a collection of the most common and broadly used utilities are shown in Listing 5.1. We use `git` and `wget` to acquire Git repositories and files from the internet, `nano` a CLI text editor for `.config` files, and compressing and decompressing utility, all of which are seen

Listing 5.1: Installing basic utility

```

1 | apt-get install -y git \
2 | wget \
3 | bc \
4 | nano \
5 | curl \
6 | cpio \
7 | unzip \
8 | rsync

```

in Listing 5.1. Any scripts that are displayed, unless specifically stated otherwise, are executed in the `bash` shell.

To be able to cross-compile source code to architecture-specific binaries, we require further packages, all of which are available on Ubuntu’s package manager `apt-get`. Among these are dependencies that the codebases of Linux, Buildroot and μ Clinux require. These tools are seen in Listing 5.2.

Lastly, we need to install `qemu` for ARM devices, this is seen in Listing 5.3. The `printf` statement allows us to answer prompted questions that are posed upon installation. The 8 answers question concerning location, which maps to Europe, the 7 to the city, in our case we chose Berlin.

Listing 5.3: Installing QEMU in the container

```

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

```

5.1.1 Downloading and setting up μ Clinux

Since the original μ Clinux is not maintained anymore, the task of finding an entity that has maintained a μ Clinux fork was a daunting task. In section 3.6.1 we established our choice, a distribution maintained by Emcraft. In Listing 5.4 all dependencies are downloaded for it.

Both U-Boot and Buildroot were cloned using `git`, the commands are shown in Listing 5.5.

Listing 5.2: Installing toolchains and dependencies

```

1 apt-get install -y gcc \
2 binutils-arm-none-eabi \
3 make \
4 gcc-arm-none-eabi \
5 gcc-arm-linux-gnueabi \
6 gcc-arm-linux-gnueabi \
7 libncurses-dev \
8 libncurses5-dev \
9 flex \
10 bison \
11 openssl \
12 libssl-dev \
13 dkms \
14 perl \
15 libelf-dev \
16 libudev-dev \
17 libpci-dev \
18 libiberty-dev \
19 autoconf \
20 lzop

```

Listing 5.4: Downloading μ Clinux into the container and setting up its special toolchain

```

1 wget https://sourcery.mentor.com\
2 /GNUToolchain/package6503/public/\
3 arm-uclinuxeabi/\
4 arm-2010q1-189-arm-uclinuxeabi-i686-pc-linux-gnu.tar.bz2
5 tar -xf arm-2010q1-189-arm-uclinuxeabi-i686-pc-linux-gnu.tar.bz2
6 rm -d arm-2010q1-189-arm-uclinuxeabi-i686-pc-linux-gnu.tar.bz2
7 export PATH=/workdir/arm-2010q1/bin:$PATH
8
9 dpkg --add-architecture i386
10 apt-get update && apt-get upgrade -y
11 apt-get install libc6:i386
12
13 #download emcraft uClinux
14 git clone https://github.com/EmcraftSystems/linux-emcraft.git
15 git clone https://github.com/EmcraftSystems/u-boot.git
16
17 #fixing filename so compiles work
18 cp /workdir/linux-emcraft/initramfs-list-min.stub \
19 /workdir/linux-emcraft/initramfs-list-min
20 cp /workdir/linux-emcraft/arch/arm/kernel/vmlinux.lds.S.good \
21 /workdir/linux-emcraft/arch/arm/kernel/vmlinux.lds.S

```

Listing 5.5: Cloning Buildroot and U-Boot

```

1 | git clone https://github.com/buildroot/buildroot.git
2 | git clone https://source.denx.de/u-boot/u-boot.git

```

Listing 5.6: notmain.c

```

1 | void PUT32 ( unsigned int , unsigned int );
2 | #define UART0BASE 0x4000C000
3 | int notmain ( void )
4 | {
5 |     unsigned int rx;
6 |     for (rx=0;rx<8;rx++)
7 |     {
8 |         PUT32(UART0BASE+0x00,0x30+(rx&7));
9 |     }
10 |    return (0);
11 | }

```

5.2 Sample Code

To establish a baseline with less complex code compared to the massive source code repository of the Linux kernel, that can fully be understood, a collection of sample code was found [17]. These three code snippets, in combination with QEMU as an emulator for the target platform, will provide a controlled environment, in which it is easier to verify if the compilation was successful and correct, and if the binaries work on the target platform, or if it was correctly linked. Having such a fallback option saves time and power, and provides a solid foundation for the complex processes that occur during cross-compilation.

`notmain.c`, as seen in Listing 5.6, written 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). Listing 5.8 and 5.7, are the corresponding linker and assembler file, respectively. Theoretically, the output, after compilation should look something like this:

```
01234567
```

5.3 FreeRTOS on STM32L4

A baseline for the STM32L476G-EVAL MCU shown in 3.2 was also established. We prepared sample code in the STM32QubeIDE mentioned in 3.10. With a few exceptions,

Listing 5.7: 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 5.8: 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
}

```

a guide was followed to reach this state [34]. In our case, the Light-Emitting Diode (LED) was connected to General-Purpose I/O (GPIO) pin B2. Seen in Listing 5.10 lines 8 and 27 had to be adapted to our specific board. The main goal of running this sample code on STM32L476G-EVAL MCU is to evaluate that the hardware is in working conditions and exclude any hardware-related problems.

The `main.c` function shown in Listing 5.9, initializes the configuration, peripherals and the kernel, starts the system clock, creates and adds two threads to the `Blink0*Handles` which in turn are added to the scheduler. The OS, or middleware, used in this example is FreeRTOS.

The main idea with these functions is that they compete for CPU time, both trying to toggle the LED located at GPIO pin B2. With different back-off times, it should give an interesting lighting pattern, with which we can confirm that both threads are working concurrently.

Listing 5.9: main.c, the main function

```

1  int main(void)
2  {
3      /* MCU Configuration ----- */
4
5      HAL_Init();
6
7      /* Configure the system clock */
8      SystemClock_Config();
9
10     /* Configure the peripherals common clocks */
11     PeriphCommonClock_Config();
12
13     /* Initialize all configured peripherals */
14     MX_GPIO_Init();
15     MX_ADC1_Init();
16     MX_CAN1_Init();
17     MX_COMP2_Init();
18     MX_DAC1_Init();
19     MX_FMC_Init();
20     MX_I2C1_Init();
21     MX_LPUART1_UART_Init();
22     MX_USART1_UART_Init();
23     MX_USART3_SMARTCARD_Init();
24     MX_OPAMP1_Init();
25     MX_SAI1_Init();
26     MX_SDMMC1_SD_Init();
27     MX_SPI2_Init();
28     /* Init scheduler */
29     osKernelInitialize();
30
31     /* Create the thread(s) */
32     /* creation of Blink01 */
33     Blink01Handle = osThreadNew(StartBlink01, NULL, &Blink01_attributes);
34
35     /* creation of Blink02 */
36     Blink02Handle = osThreadNew(StartBlink02, NULL, &Blink02_attributes);
37
38     /* Start scheduler */
39     osKernelStart();
40
41     /* We should never get here as control is now taken by the scheduler */
42     /* Infinite loop */
43     while (1)
44     {
45
46     }
47 }

```

Listing 5.10: `main.c` two blinking functions

```

1  void StartBlink01(void *argument)
2  {
3      /* init code for USB_HOST */
4      MX_USB_HOST_Init();
5      /* Infinite loop */
6      for (;;)
7      {
8          HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_2);
9          osDelay(500);
10     }
11     osThreadTerminate(NULL);
12 }
13
14 /* USER CODE BEGIN Header_StartBlink02 */
15 /**
16  * @brief Function implementing the Blink02 thread.
17  * @param argument: Not used
18  * @retval None
19  */
20 /* USER CODE END Header_StartBlink02 */
21 void StartBlink02(void *argument)
22 {
23     /* Infinite loop */
24     for (;;)
25     {
26         HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_2);
27         osDelay(600);
28     }
29     osThreadTerminate(NULL);
30 }

```

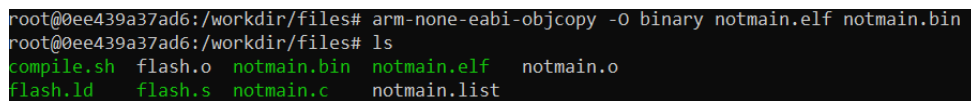

Chapter 6

Evaluation

With the setup explained in Section 5, we will now compile, evaluate with QEMU, and flash the binaries onto our boards.

6.1 Compilation

6.1.1 Sample Code



```
root@0ee439a37ad6:/workdir/files# arm-none-eabi-objcopy -O binary notmain.elf notmain.bin
root@0ee439a37ad6:/workdir/files# ls
compile.sh  flash.o  notmain.bin  notmain.elf  notmain.o
flash.ld    flash.s  notmain.c    notmain.list
```

Figure 6.1: Sample code directory after compilation

To compile this sample code for the target platform the commands shown in Listing 6.1 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 five additional files. `flash.o`, `notmain.o`, `notmain.elf`, `notmain.list` and most importantly `notmain.bin`, the "sample kernel", as seen in Figure 6.1. From Listing 6.1 we can also see the flag `-mcpu=cortex-m4`, symbolizing that we are compiling for target platform ARM Cortex-M4.

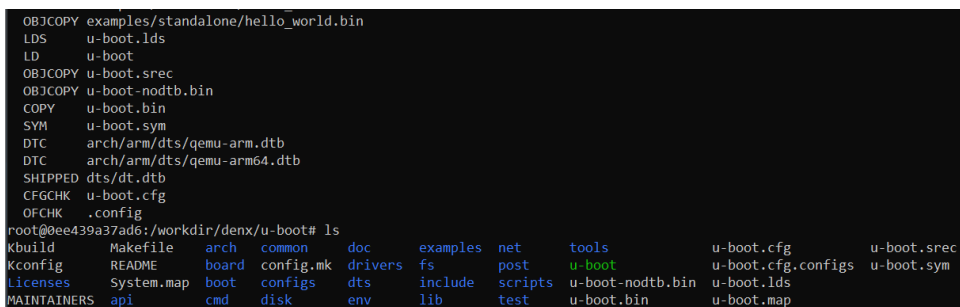
Listing 6.1: Compiling sample code with gcc-arm-none-eabi toolchain

```

1 arm-none-eabi-as -warn -fatal-warnings \
2 -mcpu=cortex-m4 flash.s -o flash.o
3 arm-none-eabi-gcc -Wall -O2 -ffreestanding \
4 -mcpu=cortex-m4 -mthumb -c notmain.c -o notmain.o
5 arm-none-eabi-ld -nostdlib -nostartfiles -T flash.ld flash.o notmain.o \
6 -o notmain.elf
7 arm-none-eabi-objdump -D notmain.elf > notmain.list
8 arm-none-eabi-objcopy -O binary notmain.elf notmain.bin

```

6.1.2 U-Boot



```

OBJCOPY examples/standalone/hello_world.bin
LDS      u-boot.lds
LD       u-boot
OBJCOPY  u-boot.srec
OBJCOPY  u-boot-nodtb.bin
COPY     u-boot.bin
SYM      u-boot.sym
DTC      arch/arm/dts/qemu-arm.dtb
DTC      arch/arm/dts/qemu-arm64.dtb
SHIPPED  dts/dt.dtb
CFGCHK   u-boot.cfg
OFCHK    .config
root@0ee439a37ad6:/workdir/denx/u-boot# ls
kbuild  Makefile  arch      common    doc        examples  net        tools      u-boot.cfg  u-boot.srec
Kconfig  README    board     config.mk drivers    fs         post       u-boot     u-boot.cfg.configs  u-boot.sym
licenses System.map boot      configs   dts        include    scripts    u-boot-nodtb.bin  u-boot.lds
MAINTAINERS api       cmd       disk      env         lib        test       u-boot.bin      u-boot.map

```

Figure 6.2: Executing Listing 6.2 with output files

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 navigate inside the folder. Using our previously established toolchain, **gcc-arm-none-eabi**, we call the commands, shown in Listing 6.2 inside of the current folder. At line 2 of Listing 6.2, **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 targets. After compilation is completed, taking about 30 seconds, **u-boot.bin** is the main output of our compilation, as seen in Figure 6.2.

Listing 6.2: 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-

```

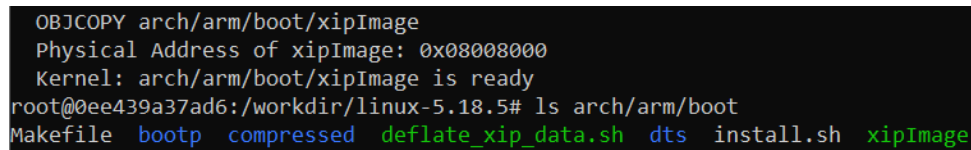
6.1.3 Mainline Linux kernel

A similar approach is taken within the directory of the Linux kernel source code. Firstly we make sure that previous configurations and compilations are deleted, this is achieved

with line 1 in Listing 6.3 make `mrproper`. On line 2 we set the `stm32_defconfig` as our config since we are trying to compile Linux for STM32 systems.

Listing 6.3: Compiling the Linux kernel

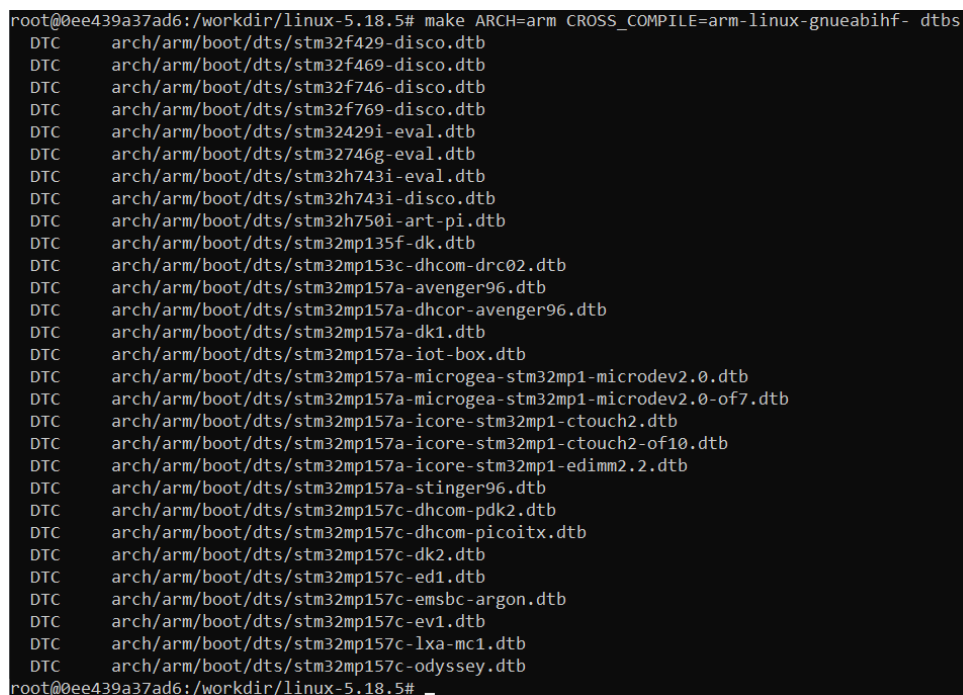
```
1 | make ARCH=arm CROSS_COMPILE=arm-none-eabi- mrproper
2 | make ARCH=arm stm32_defconfig
3 | make ARCH=arm CROSS_COMPILE=arm-none-eabi- xipImage
4 | make ARCH=arm CROSS_COMPILE=arm-none-eabi- dtbs
```



```
OBJCOPY arch/arm/boot/xipImage
Physical Address of xipImage: 0x08008000
Kernel: arch/arm/boot/xipImage is ready
root@0ee439a37ad6:/workdir/linux-5.18.5# ls arch/arm/boot
Makefile bootp compressed deflate_xip_data.sh dts install.sh xipImage
```

Figure 6.3: Directory after Linux compilation of Listing 6.3

After line 3 of Listing 6.3 we successfully compiled `xipImage` which is located in `./arch/arm/boot`.



```
root@0ee439a37ad6:/workdir/linux-5.18.5# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- dtbs
DTC      arch/arm/boot/dts/stm32f429-disco.dtb
DTC      arch/arm/boot/dts/stm32f469-disco.dtb
DTC      arch/arm/boot/dts/stm32f746-disco.dtb
DTC      arch/arm/boot/dts/stm32f769-disco.dtb
DTC      arch/arm/boot/dts/stm32429i-eval.dtb
DTC      arch/arm/boot/dts/stm32746g-eval.dtb
DTC      arch/arm/boot/dts/stm32h743i-eval.dtb
DTC      arch/arm/boot/dts/stm32h743i-disco.dtb
DTC      arch/arm/boot/dts/stm32h750i-art-pi.dtb
DTC      arch/arm/boot/dts/stm32mp135f-dk.dtb
DTC      arch/arm/boot/dts/stm32mp153c-dhcom-drc02.dtb
DTC      arch/arm/boot/dts/stm32mp157a-avenger96.dtb
DTC      arch/arm/boot/dts/stm32mp157a-dhcor-avenger96.dtb
DTC      arch/arm/boot/dts/stm32mp157a-dk1.dtb
DTC      arch/arm/boot/dts/stm32mp157a-iot-box.dtb
DTC      arch/arm/boot/dts/stm32mp157a-microgea-stm32mp1-microdev2.0.dtb
DTC      arch/arm/boot/dts/stm32mp157a-microgea-stm32mp1-microdev2.0-of7.dtb
DTC      arch/arm/boot/dts/stm32mp157a-icore-stm32mp1-ctouch2.dtb
DTC      arch/arm/boot/dts/stm32mp157a-icore-stm32mp1-ctouch2-of10.dtb
DTC      arch/arm/boot/dts/stm32mp157a-icore-stm32mp1-edimm2.2.dtb
DTC      arch/arm/boot/dts/stm32mp157a-stinger96.dtb
DTC      arch/arm/boot/dts/stm32mp157c-dhcom-pdk2.dtb
DTC      arch/arm/boot/dts/stm32mp157c-dhcom-picoitx.dtb
DTC      arch/arm/boot/dts/stm32mp157c-dk2.dtb
DTC      arch/arm/boot/dts/stm32mp157c-ed1.dtb
DTC      arch/arm/boot/dts/stm32mp157c-emsbc-argon.dtb
DTC      arch/arm/boot/dts/stm32mp157c-ev1.dtb
DTC      arch/arm/boot/dts/stm32mp157c-lxa-mc1.dtb
DTC      arch/arm/boot/dts/stm32mp157c-odyssey.dtb
root@0ee439a37ad6:/workdir/linux-5.18.5#
```

Figure 6.4: Output after running line 4 of Listing 6.3

In Figure 6.4 we can see all the device trees that are created upon the conclusion of line 4 of Listing 6.3. Unfortunately, our board, seen in Figure 3.2, is not present.

```

Kernel configured for XIP (CONFIG_XIP_KERNEL=y)
Only the xipImage target is available in this case
make[1]: *** [arch/arm/boot/Makefile:52: arch/arm/boot/Image] Error 1
make: *** [arch/arm/Makefile:318: Image] Error 2
root@0ee439a37ad6:/workdir/linux-5.18.5#

```

Figure 6.5: Error: using `zImage` as output instead of `xipImage` in Listing 6.3

In the error shown in Figure 6.5, we can see that if choosing the wrong output for our compilation, or a different output than the one defined in our `.config` file, we get an error.

6.1.4 μ Clinux

A similar approach is taken for μ Clinux. Under `arch/arm/configs`, we find the different configurations, the closest to our current board are `stm32f2_defconfig`, `stmp378x_defconfig` and `stmp37xx_defconfig`. The ones labeled with `stmp` are referring to boards with MPUs, as described in section 3.1.

Listing 6.4: Compiling μ Clinux

```

1 | make stm32f2_defconfig
2 | make CROSS_COMPILE=arm-uclinuxeabi- vmlinux

```

To prove that the toolchains and the setup work we can start compilation with commands shown in Listing 6.4. During the compilation process **warning**'s are shown, this could be due to the age of this μ Clinux distribution, yet it completes successfully. The output is a `vmlinux` file.

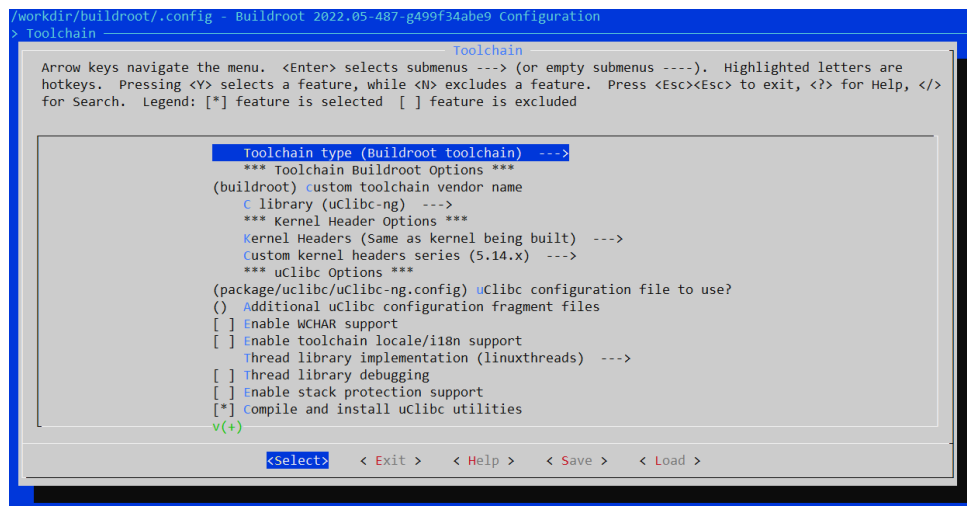
6.1.5 Buildroot

We have to 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. Equivalently to the previous steps, the `configs` directory 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 later on modified. Unfortunately STM32L476G-Eval is not included, but other configurations for STM are available, such as `stm32f429_disco_sd_defconfig` for the STM32F429-Discovery development board. `make stm32f469_disco_sd_defconfig` will copy the code

from the specified configuration into the `.config` file, which is not visible in the tree unless command `ls -a` is issued.

Listing 6.5: buildroot

```
1 | make stm32f469_disco_sd_defconfig
2 | make
```

Figure 6.6: Buildroot `make menuconfig` / Toolchain

To get further insight into what the Buildroot configuration has specified we can type command `make menuconfig` inside the directory, we are loaded into the visual configuration menu displayed in the console that can help us visually navigate through the `.config` file and configure our build further. Note that for this visual editor the `libncurses5-dev` package is required and was installed in section 5.1. This is shown in Figure 6.6. An interesting variation is to be observed, the library mentioned in section 3.6.1, `uClibc-ng`, is used.

With `make`, the compilation process starts. Depending on the local machine 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 Figure 6.7. `u-boot.bin` and `rootfs.ext2` were created automatically, `sdcard.img` can be flashed directly onto our SD-card. The usual `zImage` and device tree `stm32f469-disco.dtb` are also present. While compilation here required much more time than any of the previous compilations, Buildroot shows much promise because we are not required to configure and compile any other tools.

```
INFO: hdimage(sdcards.img): adding partition 'u-boot' (in MBR) from 'boot.vfat' ...
INFO: hdimage(sdcards.img): adding partition 'rootfs' (in MBR) from 'rootfs.ext2' ...
INFO: hdimage(sdcards.img): adding partition '[MBR]' ...
INFO: hdimage(sdcards.img): writing MBR
root@0ee439a37ad6:/workdir/buildroot# ls output/images
boot.vfat  extlinux  rootfs.ext2  sdcards.img  stm32f469-disco.dtb  u-boot.bin  zImage
```

Figure 6.7: The output/images directory after Buildroot compilation

Listing 6.6: Compiling commands used in Buildroot directory

```
1 | make stm32f469_disco_sd_defconfig
2 | make
```

```
ln -snf /workdir/buildroot/output/host/arm-buildroot-linux-uclibcgnueabi/sysroot
droot/output/staging
>>> Executing post-image script board/qemu/post-image.sh
root@0ee439a37ad6:/workdir/buildroot# ls output/images
rootfs.ext2  start-qemu.sh  versatile-pb.dtb  zImage
```

Figure 6.8: The output/images directory for QEMU

Equivalently to Listing 06.6, we can use `make qemu_arm_versatile_defconfig` instead of line 1 to compile a linux distribution specifically for QEMU. With this defconfig the output is shown in Figure 6.8. Most notably the `start-qemu.sh` script.

6.2 Emulating with QEMU

After having compiled the various source code in Section 6.1, the next step is to emulate using QEMU. To leave QEMU press CTRL+A and X.

6.2.1 Sample Code

```
root@0ee439a37ad6:/workdir/files# qemu-system-arm -M lm3s811evb -cpu cortex-m4 -m 8K -nographi
c -kernel notmain.bin
01234567
```

Figure 6.9: Executing Listing 6.7

Although in QEMUs documentation [22] it is stated that `netduinoplus2`, `netduino2` and `stm32vldiscovery` are supported and are the closest to our board, only `netduino2` is actually available. If we run `qemu` with this machine, flagged as the `-M netduino2`, we do not get an output. When using commands from Listing 6.7 we get the desired output seen in Figure 6.9.

Listing 6.7: Running sample code with QEMU

```

1 | qemu-system-arm -M lm3s811evb -cpu cortex-m4 \
2 | -m 8K -nographic -kernel notmain.bin

```

6.2.2 U-Boot

```

root@0ee439a37ad6:/workdir/denx/u-boot# qemu-system-arm -machine virt -nographic -bios u-boot.bin

U-Boot 2022.07-00737-g319d309b58 (Jul 24 2022 - 15:58:33 +0200)

DRAM: 128 MiB
Core: 47 devices, 14 uclasses, devicetree: board
Flash: 64 MiB
Loading Environment from Flash... *** Warning - bad CRC, using default environment

In: pl011@90000000
Out: pl011@90000000
Err: pl011@90000000
Net: eth0: virtio-net#32
Hit any key to stop autoboot: 0
starting USB...
No working controllers found
USB is stopped. Please issue 'usb start' first.
scanning bus for devices...

Device 0: unknown device

Device 0: unknown device

Device 0: unknown device
starting USB...
No working controllers found
BOOTP broadcast 1
DHCP client bound to address 10.0.2.15 (1 ms)
Using virtio-net#32 device
TFTP from server 10.0.2.2; our IP address is 10.0.2.15
Filename 'boot.scr.uimg'.
Load address: 0x40200000
Loading: *
TFTP error: 'Access violation' (2)
Not retrying...
BOOTP broadcast 1
DHCP client bound to address 10.0.2.15 (0 ms)
Using virtio-net#32 device
TFTP from server 10.0.2.2; our IP address is 10.0.2.15
Filename 'boot.scr.uimg'.
Load address: 0x40400000
Loading: *
TFTP error: 'Access violation' (2)
Not retrying...
=>

```

Figure 6.10: Running U-boot with QEMU

By running the command seen in Listing 6.8, we successfully loaded the bootloader with QEMU, as shown in Figure 6.10. We are presented with the CLI of U-boot.

Listing 6.8: Running U-Boot with QEMU

```

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

```

6.2.3 Buildroot

```

Kernel memory protection not selected by kernel config.
Run /sbin/init as init process
EXT4-fs (sda): warning: mounting unchecked fs, running e2fsck is recommended
EXT4-fs (sda): re-mounted. Opts: (null). Quota mode: disabled.
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Initializing random number generator: OK
Saving random seed: random: dd: uninitialized urandom read (512 bytes read)
OK
Starting network: 8139cp 0000:00:0c:0 eth0: link up, 100Mbps, full-duplex, lpa 0x05E1
udhcpd: started, v1.35.0
random: mktemp: uninitialized urandom read (6 bytes read)
udhcpd: broadcasting discover
udhcpd: broadcasting select for 10.0.2.15, server 10.0.2.2
udhcpd: lease of 10.0.2.15 obtained from 10.0.2.2, lease time 86400
deleting routers
random: mktemp: uninitialized urandom read (6 bytes read)
adding dns 10.0.2.3
OK

Welcome to Buildroot
buildroot login: root
# echo "Hello World!"
Hello World!
#

```

Figure 6.11: Starting Buildroot Linux with QEMU

Seen in Figure 6.11, by activating the `start-qemu.sh` script previously compiled in Section 6.1.5 a prompt ask us to provide login credentials, upon entering `root`, the system grants us access and we are loaded in Linux's typical `sh` shell. This exhausts the requirements of this thesis, as it provides a Linux distribution running on MCUs, emulated with QEMU.

```

root@0ee439a37ad6:/workdir/buildroot# qemu-system-arm -M netduino2 -m 2M -kernel output/images/zImage -dtb output/
images/stm32f469-disco.dtb -drive file=output/images/rootfs.ext2,if=scsi -append "root=/dev/sda console=ttyAMA0,11
5200" -nographic
WARNING: Image format was not specified for 'output/images/rootfs.ext2' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restr
icted.
Specify the 'raw' format explicitly to remove the restrictions.
qemu-system-arm: Could not load kernel 'output/images/zImage'
root@0ee439a37ad6:/workdir/buildroot#

```

Figure 6.12: Error: Starting Buildroot `stm32f469_disco_sd_defconfig` configuration

Alternatively trying to start the distribution compiled with the `stm32f469_disco_sd_defconfig` configuration, compiled in Section 6.1.5, when running the command seen in Listing 6.9 in the Buildroot directory, the kernel could not be loaded. The results are shown in Figure 6.12. Because we do not have the exact board.

Listing 6.9: Running Buildroot Linux with QEMU

```

1 | qemu-system-arm -M netduino2 \

```



```

2 |—kernel output/images/zImage \
3 |—dtb output/images/stm32f469—disco.dtb \
4 |—drive file=output/images/rootfs.ext2 ,if=scsi \
5 |—append "root=/dev/sda_console=ttyAMA0,115200" \
6 |—nographic

```

6.3 Flashing binaries

In this Section, we are flashing compiled binaries onto our devices and SD card, namely STM32L476G-Eval 3.2 and ESP-EYE DevKit 3.3.

6.3.1 FreeRTOS on STM32L4

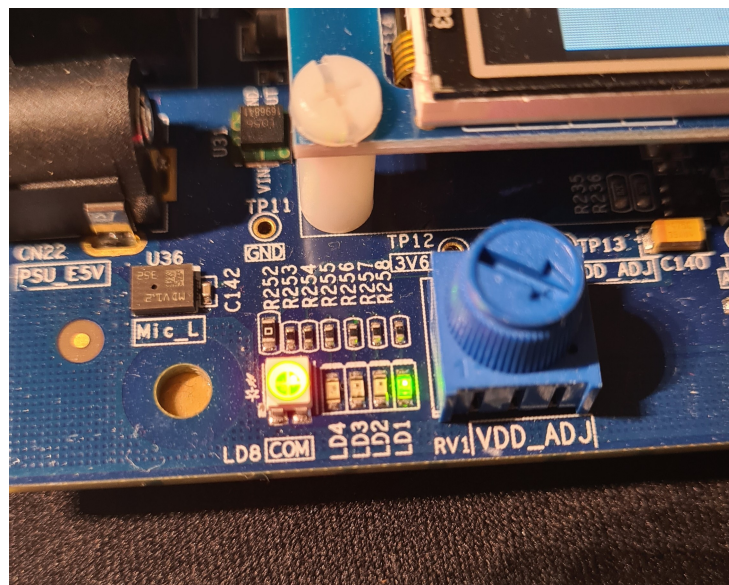


Figure 6.13: Blinking LD1

Since STM32CubeIDE was used, which includes the required toolchains and libraries, the compilation process is facilitated and was not worth mentioning in the previous chapter. By running the compiled binaries as an "STM32 Cortex-M C/C++ Application" we can observe that the system behaves correctly, and the LD1 flashes in a periodic manner described best by the `sin()` function. As the two threads compete to toggle the LD, with different `osDelays`, 600 and 500, respectively. In Figure 6.13, we can see that the light is on, as blinking is hard to capture in a picture.

6.3.2 JuiceVM

Listing 6.10: Flashing JuiceVM to ESP-EYE

```

1 |.\esptool.exe -chip esp32 -port COM7 erase_flash
2 |.\esptool.exe -chip esp32 -port COM7 ^
3 |--baud 460800 write_flash -z 0x1000 ^
4 |.\juicevm-risc-v-vm-for-esp32-wrover-linux-demo.bin

```

To flash the binaries provided by JuiceVM, namely the `juicevm-risc-v-vm-for-esp32-wrover-linux-demo.bin`, onto the ESP-EYE DevKit, we require the previously mentioned `esptool.exe`. By executing commands seen in Listing 6.10 inside the directory where the binary file is located. Note that this command was executed on the host OS, which in this case was Windows 10.

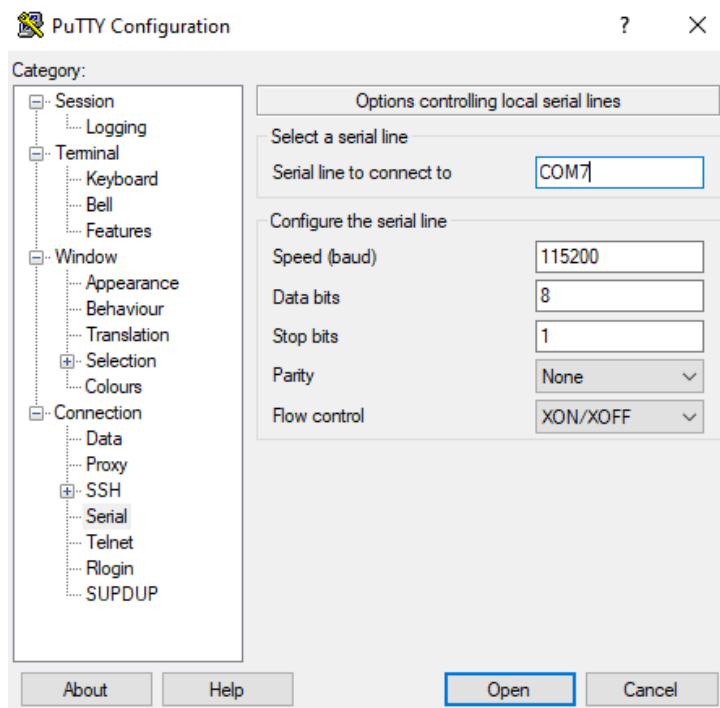


Figure 6.14: PuTTY configuration to connect to display JuiceVMs console

After flashing has been completed a connection to the console through the serial console using PuTTY is possible. It was not possible to connect to the serial port beforehand because it was occupied with flashing the files. To achieve this baud rate is set to 115200, data bits to 8, stop bits to 1, party to none, and flow control to XON/XOFF, this is shown in Figure 6.14. If a wrong baud rate is chosen, for example, the one specified in Listing 6.10, an output is visible but not coherent.

Figure 6.15: PuTTY: JuiceVM console after more than 10 hours

```
file: /mnt/ssd_prj/risc-v_sim/sim/test/opensbi/opensbi-master/platform/juice/juiceRv/platform.c func: platform_console_init line:72
OpenSBI v0.9
      _____
     |   _   _   |
    |  | | | |  |
    |  |_| |_|  |
    |_____|_|_||
[ ] [ ] 13356K/6144K available (180K kernel code, 134K rwdta, 328K rodata, 100K init, 212K bss, 2748K reserved, 0K cma-reserved)
[ ] 0.000000 f:drivers/tty/hvc/hvc_riscv_sbi.c func:hvc_sbi_init line:51
[ ] 0.000000 f:drivers/tty/hvc/hvc_riscv_sbi.c func:hvc_sbi_init line:52
[ ] 0.000000 f:include/linux/fs.h func:get_write_access_line:2851 inode: ffffffff805b7c00
[ ] 0.000000 Warning: unable to open an initial console.
[ ] 0.000000 Freeing unused kernel memory: 100K
[ ] 0.000000 This architecture does not have kernel memory protection.
[ ] 0.000000 Run /init as init process
[ ] 0.000000 f:fs/exec.c func:_do_execveat_common line:1970
[ ] 0.000000 f:fs/exec.c func:_do_execve file line:1731
[ ] 0.000000 f:fs/exec.c func:_do_execve file line:1737
[ ] 0.000000 f:fs/exec.c func:_do_execve file line:1737
```

Figure 6.16: PuTTY: JuiceVM console after 32 hours

After close to 32 hours, seen in Figure 6.16, it was decided to quit the proceedings due to the unusably long boot process. At this point the `printf()` function took 12 seconds to print a character to the console. Interestingly we can see the line **Freeing unused kernel memory: 100K**, which was making space in the main memory for other objects. The reason for such a long boot process is not known, and requires further investigation.

6.3.3 Buildroot



Figure 6.17: Flashing SD card with `sdcard.img`

The files used in this Section resulted from the compilation process seen in Section 6.1.5 with the `stm32f469_disco_sd_defconfig` configuration. While this is not the board we are working with the CPU architecture is the same. Firstly, we need to flash our `sdcard.img` onto the 4GB SD card provided with the STM32L476G-Eval board, shown in 6.17.

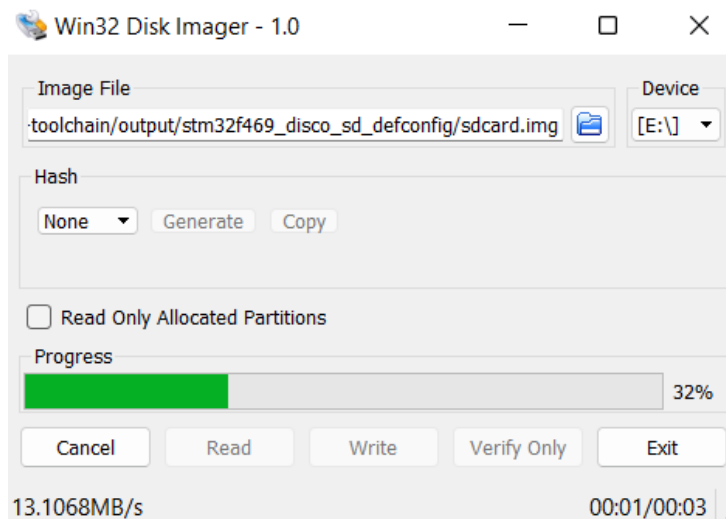


Figure 6.18: Flashing SD card with `sdcard.img` using Disk Imager

To achieve this the Disk Imager tool is used, as shown in Figure 6.18. The process takes less than 3 seconds and is completed successfully. The contents of the SD card are shown in Figure 6.19.

extlinux	22/06/2022 10:20	File folder	
stm32f469-disco.dtb	22/06/2022 10:20	DTB File	20 KB
zImage	22/06/2022 10:20	File	1.701 KB

Figure 6.19: Contents of SD card after flashing

Inside the `extlinux` folder a single file is found, the `extlinux.conf` file. These files are created automatically and are a result of flashing the SD card.

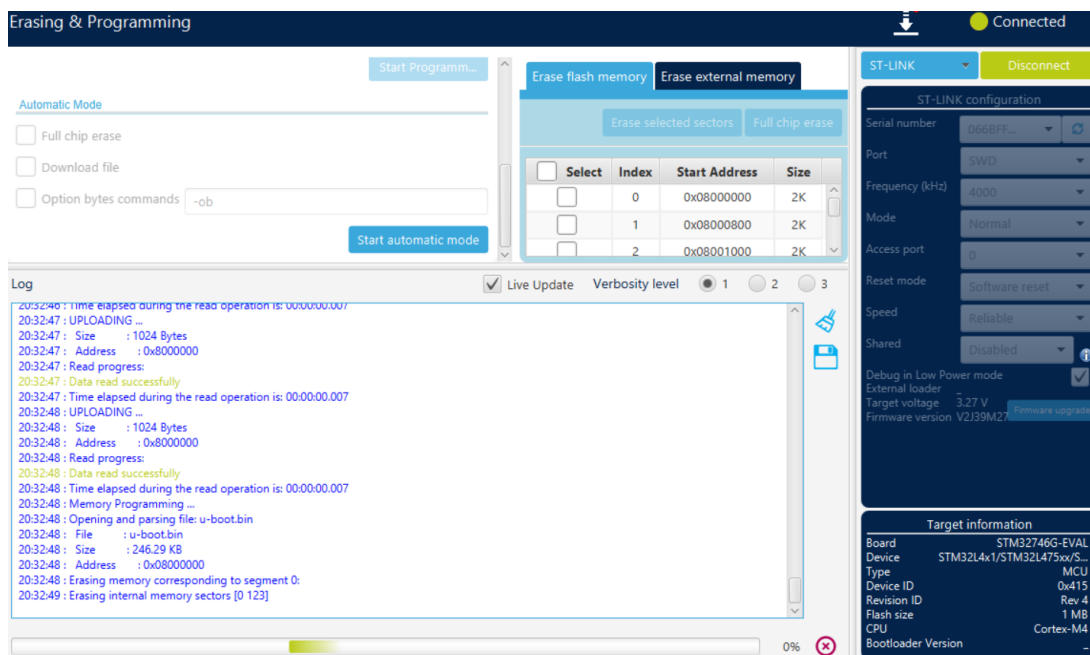


Figure 6.20: Flashing U-Boot onto internal ROM of STM32L476G-Eval

Shown in Figure 6.20, using the STM32CubeProgrammer tool, we flash `u-boot.bin` directly to the MCU. We can observe that the file has a size of 246.29 KB and the starting physical address where it was flashed, namely `0x08000000`. In the bottom right, under "Target Information", our board is identified as the one specified earlier.

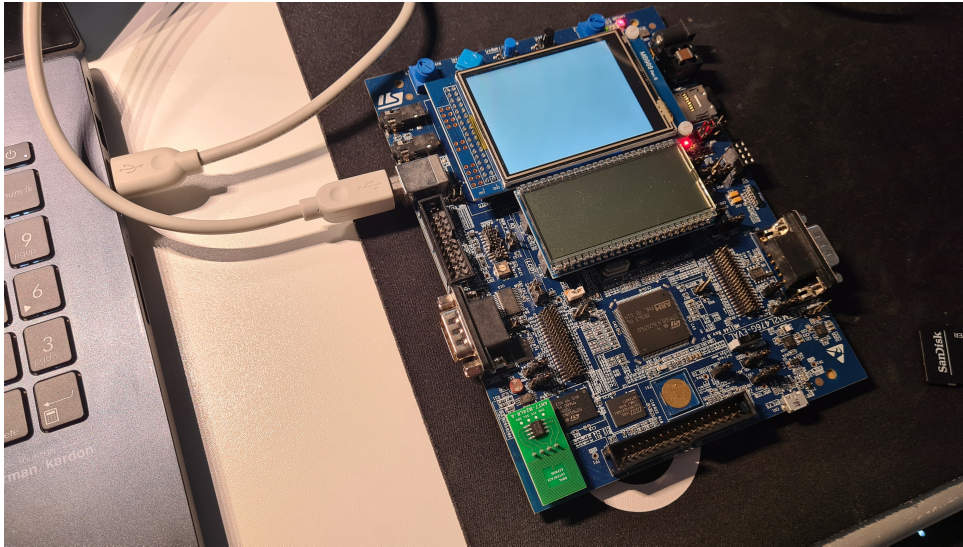


Figure 6.21: Laptop connected to STM32L476G-Eval through USB Type-B

In Figure 6.21 the connection to the board is visible. The ST-Link/V2-1 interface is connected to the PC through a USB Type-B cable, acting as a power source and data exchange simultaneously. While flashing is in progress, the red light in the top right corner flashed rapidly.

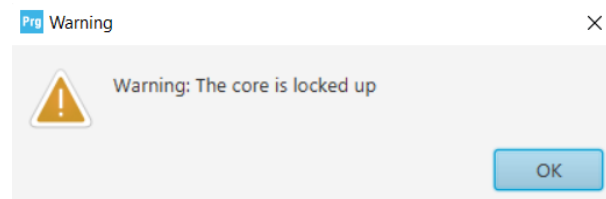


Figure 6.22: Flashing U-Boot: Warning the core is locked up

Upon completion, we get a warning dialog box, seen in Figure 6.22. The cause of such an error can have various origins, and further investigation is required.



Figure 6.23: No console output in PuTTY

Lastly, when connecting to the serial port COM4 using PuTTY, for which we used the same configuration as shown in Figure 6.14, we can see no output as seen in Figure 6.23.

A similar output as seen in Figure 6.11, was hoped for. Although this was to be expected since the compilation was performed for STM32F469 and not STM32L476, while both employ a Cortex-M4 CPU, this common ground was not enough.

Chapter 7

Summary, Conclusions & Future Work

In this thesis, different Linux distributions for IoT edge devices were compiled, with a proposed standardized architecture in the foreground. To achieve this a market analysis of IoT hardware was performed, a variety of tools and open source projects such as Buildroot and JuiceVM were explored and introduced, and Linux and U-Boot binaries were emulated on QEMU. Furthermore, it was attempted to flash Linux binaries onto an STM32L476G-Eval development board which was not successful due to lacking community support for the board, and running a RISC-V emulation on ESP-EYE, which was technically successful but unusable in practice.

The diverse IoT ecosystem and the lack of standardization paired with a manufacturer-driven industry, are the primary justifications for this thesis. The variety of IoT network protocols and OSs, in this these referred to as frameworks rather than OSs, only amplify this heterogeneity. We propose standardization of the OS layer for IoT edge devices, such that an abstraction of the diverse hardware is possible. This would have the effect that board-specific implementations, such as bare metal code, are not required, but that the user can choose from diverse solutions that all work upon the OS layer, therefore the application layer would gain portability.

The process of this thesis was to evaluate possible MCU candidates, that displayed promising qualities, such as sufficient RAM, a CPU with more than 40MHz clock speed while considering energy efficiency and cost. STM in particular showed desired qualities such as a healthy community and the relevant ARM Cortex-M architectures. At this point, contact with STM employees was established and they generously supplied us with two STM32L476G-Eval development boards. Only then did we explore the means to find the right implementation for the desired goal, Linux on MCUs. This thesis concludes that this bottom-up approach is fundamentally flawed, it is not necessarily the hardware that

enables the use of Linux, but the community that adapts the kernel to the physical layer. By searching for compatible hardware first, work previously done by the open source community is omitted. The best possible approach to solve a given use case is to compile a list of MCUs that are supported by Linux and tools such as Buildroot, not to choose a board and realize that it is not supported. In such a case, the upfront investment, of porting Linux to a specific board is not worth it. Yet, the explorative nature of this thesis brought forward a different perspective, and different tools such as Buildroot were discovered and evaluated.

Buildroot proved to be an invaluable tool for embedded Linux, as it contains all necessary components such as libraries like the `uClibc` and U-Boot. Supporting multiple architectures and subsequently multiple boards, even more so than the mainline Linux kernel. It should further be evaluated with the right boards available. As we performed tests on the STM32L476G-Eval development board, which did not have a dedicated configuration for compilation, we had to adapt configurations of similar boards, which ultimately did not work because the contrast was too large, as is common in IoT.

While the success of this thesis was evaluated on QEMU in section 6.2.3, it does not support all the different boards that are available on the market, understandably so. As STM boards were not present but recommendations that similar boards could be emulated instead and that the architecture is similar, yet it did not work as intended. Another aspect that QEMU can not achieve is to verify performance on the target architecture since the underlying hardware is much more capable.

JuiceVM was successfully run, but in practice, it is not a viable or usable solution due to its long boot times. Yet it is a very interesting implementation, and potentially opens up the door to further studies of virtualization on tiny edge devices. Applying such concepts, such as container technology, to tiny IoT devices, if further enhances the portability and versatility of such devices.

Lastly, we provide a modular toolchain container, containing all tools required for embedded Linux, that can be used to cross-compile Linux or μ Clinux kernel, and evaluate their correctness with QEMU.

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] “EmcraftSystems/uclinux,” last-accessed: Jul 18, 2022. [Online]. Available: <https://github.com/EmcraftSystems/linux-emcraft>
- [7] “GNU Arm Embedded Toolchain Downloads,” last-accessed: Jul 15, 2022. [Online]. Available: <https://developer.arm.com/downloads/-/gnu-rm>
- [8] “Graphic of Von Neumann Architecture,” last-accessed: Jul 18, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Von_Neumann_architecture#/media/File:Von_Neumann_Architecture.svg
- [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] “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>
- [12] “MCU toolchains,” last-accessed: Jul 27, 2022. [Online]. Available: <https://github.com/LagShaggy/mcu-toolchain>
- [13] “Memory Management: Concepts overview,” last-accessed: Jul 15, 2022. [Online]. Available: <https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html>
- [14] “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>
- [15] “OpenWRT,” last-accessed: Jul 15, 2022. [Online]. Available: <https://openwrt.org/>
- [16] “QEMU,” last-accessed: Jul 14, 2022. [Online]. Available: <https://www.qemu.org/docs/master/about/index.html>
- [17] “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>
- [18] “Raspberry Pi 4,” last-accessed: Jul 14, 2022. [Online]. Available: <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>
- [19] “Raspberry Pi OS,” last-accessed: Jul 14, 2022. [Online]. Available: <https://www.raspberrypi.com/software/>
- [20] “Shared Libraries without an MMU,” last-accessed: Jul 15, 2022. [Online]. Available: <http://xflat.sourceforge.net/NoMMUSharedLibs.html>
- [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] “STMicroelectronics STM32 boards (netduino2, netduinoplus2, stm32vldiscovery),” last-accessed: Jul 20, 2022. [Online]. Available: <https://www.qemu.org/docs/master/system/arm/stm32.html>
- [23] “Tensilica Controllers and Extensible Processors,” last-accessed: Jul 26, 2022. [Online]. Available: https://www.cadence.com/en_US/home/tools/ip/tensilica-ip/tensilica-xtensa-controllers-and-extensible-processors.html

- [24] “The U-Boot Documentation,” last-accessed: Jul 14, 2022. [Online]. Available: <https://u-boot.readthedocs.io/en/latest/index.html>
- [25] “torvalds/linux,” last-accessed: Jul 15, 2022. [Online]. Available: <https://github.com/torvalds/linux>
- [26] “u-boot,” last-accessed: Jul 14, 2022. [Online]. Available: <https://github.com/u-boot/u-boot>
- [27] “uClibc-ng - Embedded C library,” last-accessed: Jul 20, 2022. [Online]. Available: <https://uclibc-ng.org/>
- [28] “uclinux.org,” last-accessed: Jul 15, 2022. [Online]. Available: <https://web.archive.org/web/20181113230737/http://www.uclinux.org/>
- [29] “Yocto Project: Release 2.1,” last-accessed: Jul 15, 2022. [Online]. Available: <https://docs.yoctoproject.org/migration-guides/migration-2.1.html?highlight=mmu>
- [30] 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.
- [31] 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.
- [32] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, “RIOT OS: Towards an OS for the Internet of Things,” in *2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*. IEEE, 2013, pp. 79–80.
- [33] A. Banafa, “IoT standardization and implementation challenges,” *IEEE internet of things newsletter*, pp. 1–10, 2016.
- [34] Digi-Key, “Getting Started With STM32 and Nucleo Part 3: FreeRTOS - How To Run Multiple Threads w/ CMSIS-RTOS,” last-accessed: Jul 14, 2022. [Online]. Available: https://www.youtube.com/watch?v=OPrcpbKNSjU&ab_channel=Digi-Key
- [35] A. Dunkels, “Full {TCP/IP} for 8-Bit Architectures,” in *First International Conference on Mobile Systems, Applications, and Services (MobiSys2003)*, 2003.
- [36] S. Fang, L. Xu, Y. Zhu, Y. Liu, Z. Liu, H. Pei, J. Yan, and H. Zhang, “An integrated information system for snowmelt flood early-warning based on internet of things,” *Information Systems Frontiers*, vol. 17, no. 2, pp. 321–335, 2015.

- [37] 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.
- [38] 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/>
- [39] W. Han, Y. Gu, W. Wang, Y. Zhang, Y. Yin, J. Wang, and L.-R. Zheng, "The design of an electronic pedigree system for food safety," *Information Systems Frontiers*, vol. 17, no. 2, pp. 275–287, 2015.
- [40] W. H. Hassan *et al.*, "Current research on Internet of Things (IoT) security: A survey," *Computer networks*, vol. 148, pp. 283–294, 2019.
- [41] Jay Carlson, "So you want to build an embedded Linux system?" last-accessed: Jul 20, 2022. [Online]. Available: <https://jaycarlson.net/embedded-linux/#>
- [42] V. P. Kaffle, Y. Fukushima, and H. Harai, "Internet of things standardization in ITU and prospective networking technologies," *IEEE Communications Magazine*, vol. 54, no. 9, pp. 43–49, 2016.
- [43] R. Krishnamoorthy, K. Krishnan, B. Chokkalingam, S. Padmanaban, Z. Leonowicz, J. B. Holm-Nielsen, and M. Mitolo, "Systematic approach for state-of-the-art architectures and system-on-chip selection for heterogeneous IoT applications," *IEEE Access*, vol. 9, pp. 25 594–25 622, 2021.
- [44] 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>
- [45] A. Milinković, S. Milinković, and L. Lazić, "Choosing the right RTOS for IoT platform," *Infoteh Jahorina*, vol. 14, pp. 504–9, 2015.
- [46] J. Newman, "Why the Internet of things might never speak A common language," *App Economy, Fast Company*, 2016.
- [47] 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/>
- [48] J. Romkey, "Toast of the IoT: The 1990 Interop Internet Toaster," *IEEE Consumer Electronics Magazine*, vol. 6, no. 1, pp. 116–119, 2017.

- [49] 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.
- [50] E. Schiller, A. Aidoo, J. Fuhrer, J. Stahl, M. Ziörjen, 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>
- [51] P. Sethi and S. R. Sarangi, “Internet of things: architectures, protocols, and applications,” *Journal of Electrical and Computer Engineering*, vol. 2017, 2017.
- [52] 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.
- [53] 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.
- [54] 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/>
- [55] 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.
- [56] 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.
- [57] Z. Wu, K. Qiu, and J. Zhang, “A smart microcontroller architecture for the Internet of Things,” *Sensors*, vol. 20, no. 7, p. 1821, 2020.

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
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
PCB	Print-Circuit Board
RAM	Random Access Memory
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
FMC	Flexible Memory Controller
VM	Virtual Maschine

KVM	Kernel-based Virtual Machine
GUI	Graphical User Interface
CLI	Command Line Interface
RTOS	Real-Time Operating System
UART	Universal Asynchronous Receiver-Transmitter
LTS	Long Time Support
LED	Light-Emitting Diode
GPIO	General-Purpose I/O

Glossary

Edge Devices An edge device refers to perception layer devices, oftentimes powered by an MCU. They are the sensors that reside at the bottom of the chain of IoT, thus called the "edge" device.

Board A board refers to a PCB upon which an MCU and peripherals are mounted.

List of Figures

3.1	Von Neumann Architecture [8]	14
3.2	STM32L476G-Eval development board [11]	17
3.3	ESP-EYE	18
4.1	Proposed architecture of MCU IoT devices	26
6.1	Sample code directory after compilation	37
6.2	Executing Listing 6.2 with output files	38
6.3	Directory after Linux compilation of Listing 6.3	39
6.4	Output after running line 4 of Listing 6.3	39
6.5	Error: using <code>zImage</code> as output instead of <code>xipImage</code> in Listing 6.3	40
6.6	Buildroot <code>make menuconfig</code> / Toolchain	41
6.7	The <code>output/images</code> directory after Buildroot compilation	42
6.8	The <code>output/images</code> directory for QEMU	42
6.9	Executing Listing 6.7	42
6.10	Running U-boot with QEMU	43
6.11	Starting Buildroot Linux with QEMU	44
6.12	Error: Starting Buildroot <code>stm32f469_disco_sd_defconfig</code> configuration	44
6.13	Blinking LD1	45
6.14	PuTTY configuration to connect to display JuiceVMs console	46

6.15 PuTTY: JuiceVM console after more than 10 hours	47
6.16 PuTTY: JuiceVM console after 32 hours	47
6.17 Flashing SD card with <code>sdcard.img</code>	48
6.18 Flashing SD card with <code>sdcard.img</code> using Disk Imager	48
6.19 Contents of SD card after flashing	49
6.20 Flashing U-Boot onto internal ROM of STM32L476G-Eval	49
6.21 Laptop connected to STM32L476G-Eval through USB Type-B	50
6.22 Flashing U-Boot: Warning the core is locked up	50
6.23 No console output in PuTTY	50

List of Tables

2.1	Key characteristics of TinyOS, Contiki, RIOT, and Linux [32]	12
3.1	Market analysis of available IoT edge devices	16

Appendix A

Installation Guidelines

Since a Docker container was used in the thesis, the installation of the toolchains container is very simple.

1. Download and install Docker.
2. Download and install Git.
3. Clone the repository provided here [12].
4. Inside the directory run command: `docker build -t mcu-toolchain:latest .`
5. Then run `docker run -it mcu-toolchain:latest.`

You have successfully installed the container, and are currently inside it.

Appendix B

Contents of the CD

- STM32CubeIDE project containing source code based on FreeRTOS toggling the LED.
- Docker container
- Installation files for all utilities used.
- Thesis with latex files and the graphics.
- Final presentation
- JuiceVM folder