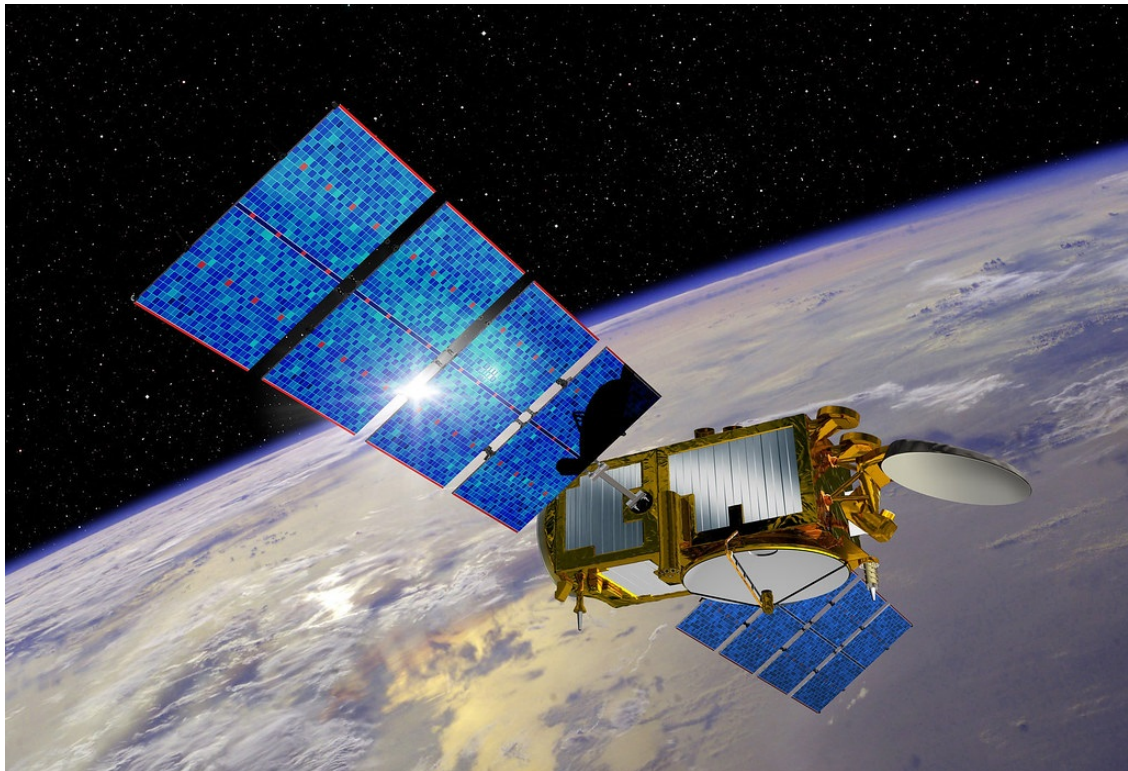




CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG



Building a Linux Distribution for Space Computers

Master's Thesis in Computer Systems and Networks &
High-Performance Computer Systems

LINA LAGERQUIST SERGEL & GUSTAV PETTERSSON

MASTER'S THESIS 2021

Building a Linux Distribution for Space Computers

Lina Lagerquist Sergel & Gustav Pettersson



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Building a Linux Distribution for Space Computers
Lina Lagerquist Sergel & Gustav Pettersson

© Lina Lagerquist Sergel & Gustav Pettersson, 2021.

Chalmers Supervisor: Roger Johansson, Computer Science and Engineering
Company Supervisor: Peter Spjuth & Eric Fornstedt, RUAG Space
Examiner: Johan Karlsson, Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Satellite in space - Work is marked as being in the public domain via CC.

Typeset in L^AT_EX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Building a Linux Distribution for Space Computers
Lina Lagerquist Sergel & Gustav Pettersson,
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Computer platforms used in mission- or safety-critical applications are often designed to support execution of hard real-time tasks. Linux has traditionally not been used as a real-time operating system for these applications but has become a more viable option within recent years.

In this thesis, we explore how embedded Linux distributions can be built to be used as a real-time operating system for space computers. We compare how the build systems Yocto Project and Buildroot can generate the components required to boot Linux on two reference design boards provided by RUAG Space. By using the benchmark tools `cyclicttest` and `lat_ctx` we evaluate the latency of response to an interrupt and context switching latency. This is done for two builds on one of the reference design boards. Further, we provide pointers on how an implementation can be made on RUAG Space's custom space computer, and discuss what components should be included in such a build.

Based on our benchmark results, we conclude that a Linux kernel equipped with the PREEMPT_RT patch has an improved deterministic behaviour. The patched kernel compared to a basic version of the kernel has a reduced maximum latency when reacting to an interrupt. The patched kernel also has a more consistent increasing context switching latency based on the number of involved processes. Regarding the build systems we can, through our observations, conclude that Buildroot is more user-friendly than Yocto. However, we perceive that Buildroot can be more limiting because of its reduced complexity and available packages.

Keywords: Linux, PREEMPT_RT, cyclicttest, rt-tests, LMBench, Yocto Project, Buildroot, RTOS

Acknowledgements

We would like to thank RUAG Space for offering us the opportunity to finalize our studies within an area we find very interesting. We appreciate that they provided us with an office to work in and the necessary equipment that made it possible for us to complete this thesis. A special thanks to our supervisors at RUAG Space, Eric Fornstedt and Peter Spjuth, that always were available to provide us a helping hand when we needed one. We also would like to thank Maria Palmqvist and Viktor Fägerlind for the rewarding conversations making it possible for the project to progress.

Moreover, we would like to thank our supervisor at the Computer Science and Engineering department, Roger Johansson, for helping us shape this thesis and giving us valuable advice throughout. Thanks to our examiner, Johan Karlsson, for the extensive amount of feedback on our writing, and for assuring that the thesis reached its expectations.

Finally, we would like to give our thanks to our peer reviewers, Åke Axeland and Omar Oueidat, for enjoyable conversations and for sharing valuable feedback that has been very useful.

Lina Lagerquist Sergel & Gustav Pettersson, Gothenburg, June 2021

Contents

List of Figures	xi
List of Tables	xiii
Glossary	xv
1 Introduction	1
1.1 Purpose & Scope	1
1.2 Structure	2
2 Background	3
2.1 Space Computers	3
2.1.1 RUAG's On-Board Computers	3
2.2 Operating Systems	4
2.3 Real-Time Operating Systems	5
2.3.1 Performance Metrics	6
2.4 The boot process	7
2.4.1 Das U-Boot	7
2.5 Linux	8
2.5.1 Linux as an RTOS	8
2.5.2 Build Systems	9
2.5.3 Components	10
2.6 Tools	11
2.6.1 UART	11
2.6.2 Screen	12
2.6.3 Trivial File Transport Protocol	12
2.6.4 LMBench	12
2.6.5 Rt-tests	12
3 RDB Deployment and Benchmarking	15
3.1 Host Environment Setup	15
3.1.1 Choice of Build System	15
3.1.2 Yocto Setup	16
3.2 Target Environment Setup	16
3.2.1 Attempts to build and deploy Linux on P2020RDB	18

3.2.2	Moving ahead with LS1046A	19
3.3	Benchmarks	20
3.3.1	<code>cyclictest</code>	20
3.3.2	<code>lat_ctx</code>	21
3.3.3	Results	21
4	Pointers for Implementation on the cOBC	27
4.1	Requirements to boot on the cOBC	27
4.2	Boot wrappers	28
4.3	Build for P2020/cOBC with Buildroot	29
5	Discussion	33
6	Conclusion	39
6.1	Future Work	40
	Bibliography	41
A	Appendix A	I
B	Appendix B	III

List of Figures

2.1	Abstract view of the components included in a desktop computer system.	4
2.2	Illustration a context switch.	6
2.3	An abstract view of how the boot process happens in multiple stages.	7
2.4	Illustration of a root file system with the root directory and some of its sub-directories.	10
2.5	An example of how a DTS is structured.	11
3.1	Results from <code>cyclicttest</code> <i>without</i> background load for the vanilla and the PREEMPT_RT patched kernels.	22
3.2	Results from <code>cyclicttest</code> <i>with</i> background load for the vanilla and the PREEMPT_RT patched kernels	23
3.3	Plot based on results from <code>lat_ctx</code> for the vanilla kernel.	24
3.4	Plot based on results from <code>lat_ctx</code> for the patched kernel.	25
4.1	The menuconfig interface where it is possible for a user to choose Buildroot configuration.	29

List of Tables

3.1	Minimum, average and maximum latency from <code>cyclictest</code> on an idle system.	22
3.2	Minimum, average and maximum latency from <code>cyclictest</code> on a system with a background load.	23

Glossary

BSP Board Support Package

libc C Library

cOBC constellation On-Board Computer

COTS Commercial-off-the-shelf

DTB Device Tree Blob

DTS Device Tree Source

FDT Flattened Device Tree

FIT Flattened Image Tree

glibc GNU C Library

LSDKYOCTOUG Layerscape Software Development Kit User Guide for Yocto

OS Operating System

RDB Reference Design Board

rootfs Root File System

RTOS Real-Time Operating System

SMP Symmetric Multiprocessing

TFTP Trivial File Transfer Protocol

U-Boot Das U-Boot

UART Universal Asynchronous Receiver/Transmitter

1

Introduction

When launching computers into space there are many design options to take into consideration. One cannot simply take a standard off-the-shelf computer, plug it into a satellite and launch it into space. Just like with cars or aircraft, it is not desirable that the computers sent into space fail or experience lag. Computer failures can have devastating consequences in this kind of systems. This is where Real-Time Systems comes in, where not only the logical correctness of the system is important, but also at what time the results are generated.

The term "real-time" can be distinguished as soft and hard real-time [1]. Examples of soft real-time could be applications on your desktop that are controlling the video or sound where minor inaccuracies are acceptable. Hard real-time could for example be applications that run on safety-critical platforms, for instance in cars or planes where there is no room for hiccups.

Platforms that execute critical tasks traditionally uses a Real-Time Operating System (RTOS) [2]. A prime example of an RTOS is VxWorks, which has been used extensively for space application where timely delivery of results can be paramount for mission success [3]. Linux is today used in various types of systems but has not been a natural choice as an RTOS, since the focus is mainly on desktop and server use. This is changing since there is an interest in the market, partly because of the lucrative aspects of Linux. It has for instance been widely used by SpaceX for their on-board computers on their satellites and rockets [4, 5].

The work reported in this thesis was carried out in collaboration with RUAG Space (from now on referred to as RUAG) which is a leading supplier of products for the space industry in Europe [6]. We had access to their guidance and hardware throughout our thesis project.

1.1 Purpose & Scope

The goal of this thesis project was to study the feasibility to use Linux as a RTOS for space computers. This was done by investigating if, and how an implementation could be made on two Reference Design Boards (RDB) provided by RUAG. The research questions addressed in this thesis are:

1. Is Linux suitable for a platform that must ensure hard real-time requirements?
2. What components are suitable to use when building a Linux distribution for space computers?
3. Does the PREEMPT_RT patch for Linux improve the deterministic behaviour for interrupt latency and task switching latency?

There are many open source tools available for building a Linux distribution for a given hardware platform. We conducted two tests to evaluate the real-time properties of the Linux kernel in this thesis. These tests are limited to factors that affect the performance of the operating system. This thesis does not provide a finished product, but rather documentation and pointers to how further development can be made.

1.2 Structure

The remainder of the thesis is organized as follows. The gathered background theory based on an initial literature study is presented in Chapter 2. In Chapter 3, we describe several attempts to produce a Linux distribution for the two RDBs provided by RUAG. In the same chapter, we present results of the benchmark tests conducted on one of the RDBs called LS1046ARDB. In Chapter 4, we provide guidelines for how a custom-built Linux distribution can be implemented on one of RUAG's space computers. The contents of this report is discussed in Chapter 5 and lastly, we conclude by reflecting upon our work in Chapter 6.

2

Background

This chapter describes fundamental concepts and background information, which serves as a foundation for the rest of thesis. First, we give a short introduction to space computers, Operating Systems (OS) and RTOS. We then discuss Linux and its use as a RTOS. We also discuss how Linux distributions can be built for embedded platforms. Finally, we provide an overview of the Linux tools we selected for our investigations.

2.1 Space Computers

Computers installed in mobile objects such as spacecrafts are called on-board computers [7]. An example of why these are not too similar to normal desktop computers is because of the unique aspect that computers in space are susceptible to errors caused by cosmic radiation [8]. This could traditionally be dealt with by using radiation-hardened components [9]. An alternative solution has been applied by SpaceX which rather uses Commercial-Off-The-Shelf (COTS) technology. COTS is merchandise that is available for sale and not specifically tailored for the finished product. This technology brings advantages such as reduced development time, simplified supply-chain, and cost efficiency [10]. Using COTS, SpaceX focuses on the creation of a radiation-tolerant design for a system such as redundancy in computation [5].

These On-Board Computers do not run a typical OS such as Windows 10, since they are not intended for plugging in a monitor and provide a friendly user experience. It may vary, but the focus rather lies in OSs that are reliable, deterministic, and secure. These are typically referred to as RTOS, which we discuss in Section 2.3.

2.1.1 RUAG's On-Board Computers

Two of RUAG's on-board computers in development are the constellation On-Board Computer (cOBC), and the Lynx. Both of these are based on COTS technology. The focus in this thesis is two RDBs, the P2020RDB and LS1046ARDB, which uses the same processor as the cOBC and Lynx respectively. An RDB can be used to quickly evaluate and demonstrate a new design.

Both of these processors are produced by NXP Semiconductors (formerly Freescale). The cOBC and P2020RDB uses a PowerPC processor with two e500 cores, namely a QorIQ P2020 [11]. The Lynx and LS1046ARDB both uses a LS1046A processor based on the ARM architecture, with four Cortex-A72 cores [12].

2.2 Operating Systems

In a desktop computer system, the OS is placed between the computer hardware and the application programs users have access to, see Figure 2.1. Such OS provides users with several services and controls program execution to prevent errors and improper use of the computer.

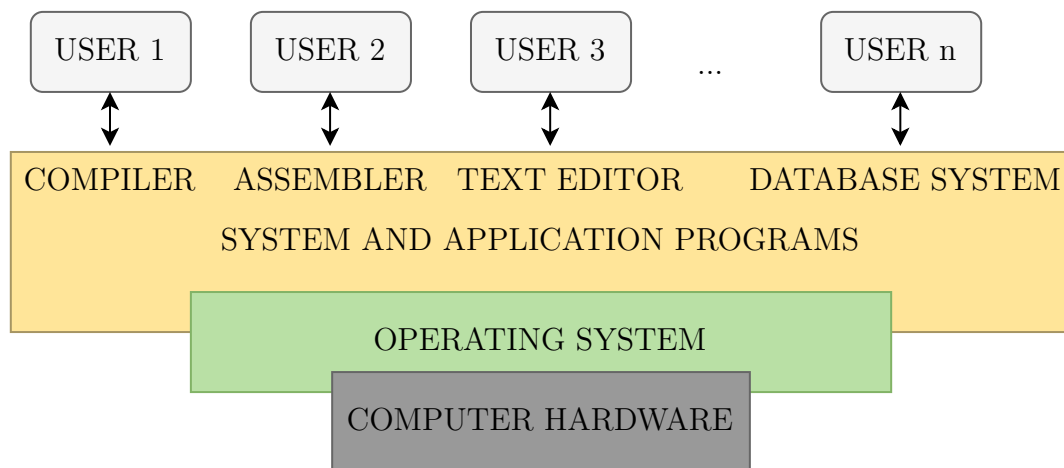


Figure 2.1: Abstract view of the components included in a desktop computer system.

An OS manages devices connected to the computer and takes actions for every event a device may generate, also called an interrupt. For example, pressing a key on a keyboard generates an interrupt which the OS will evaluate and schedule so that the character shows up on the screen. The core component of an OS is called the *kernel* and its job is to schedule and control the execution of system and user programs. A computer program such as a text editor or web browser request services from the kernel via *system calls* [13].

A computer can operate in two different modes: *user mode* and *kernel mode*. In the former, the system is simply running a user program. In kernel mode, a privileged program is being executed where the process has unrestricted access to system resources like hardware, memory, etc.

The main services provided by an OS are [14]:

- allowing a user to interact with the system
- allowing a user to access files and directories

- allowing programs to run
- allowing communication with I/O devices

A *task* can be explained as set of program instructions loaded into memory that an OS controls and a *thread* as a flow of a task being executed on a processor.

An important component of the kernel is the *scheduler* whose job is to assign and decide how tasks should run on the processor. The scheduler also makes sure that the utilization of the processor is maximized [15]. A scheduler can use different types of scheduling algorithms to rank a task's need for CPU time. An important class of scheduling algorithms are those that assign priorities to tasks. An example of such an algorithm is priority-based scheduling. This is where tasks are assigned a priority based on some property, and the tasks with the highest priority should be scheduled first. Hence, the scheduler ensures that tasks always preempt running tasks with a lower priority [15]. In real-time systems where deadlines have to be met, a higher priority could, for example, be given to tasks with the shortest deadline.

In embedded systems, embedded operating systems are used as they are designed to be more compact, reliable, and resource efficient [16]. An embedded OS generally does not load and execute a variety of individual programs at a user's command as a standard desktop OS would. Instead, it is usually designed for a single purpose to cover specific tasks. The commands to the kernel are coming from applications rather than from users.

2.3 Real-Time Operating Systems

RTOS are designed to ensure timely delivery of result from tasks with strict timing requirements. [17]. Technically, we use real-time applications on our traditional desktops in our everyday life in the form of video and sound [1]. In those cases, inaccuracies or delays are acceptable since they will usually go unnoticed to the user, and these could be classified as tasks with soft real-time requirements. The OS we refer to as RTOS are usually intended for applications with strict requirements on precise delivery of computational results. This could be where the running tasks are safety-critical such as sensors in cars or airplanes where a delay and imprecision could have devastating consequences. These tasks are classified as hard real-time tasks. The general goal for hardening a real-time system is to minimize response times and removing possible unbounded latencies [18]. The term "response time" can be explained as the elapsed time between an interrupt and the response to that interrupt and "latency" describes some type of delay.

There is no actual formal definition of how "hard" or "soft" a system is. It is rather a spectrum where the placement of the hardness depends on the importance of meeting a deadline. In the end, real-time is about timely delivery of results rather than fast execution time and high performance [19].

One example of a commercial RTOS is VxWorks, which has been used extensively

for space application, where timing can be paramount for mission success [3]. Wind River, which is the distributor of VxWorks, themselves mentions that VxWorks is “...ideal for hard real-time embedded applications.” [20].

2.3.1 Performance Metrics

An area of interest when measuring an RTOS’s performance is the timely delivery of results. Two factors that affect this is the context switching latency and the latency of response to an interrupt.

Context switching is the process of switching between two independent tasks with the same priority [21]. Context switching consists of the following actions:

- Save the context of the task being suspended.
- Select a new task.
- Prepare a new task to be executed.

Even though it is impossible to avoid the context switching latency in its entirety, it is desired that it becomes as minimal as possible since the CPU is not doing any useful work during this time [22].

The calculation of a context switch is straightforward. As illustrated in Figure 2.2, at the end of the suspended task *task1*, the time is denoted as t_1 , and the beginning of the new task *task2*, the time is t_2 . The time for the context switch becomes $t_2 - t_1$.

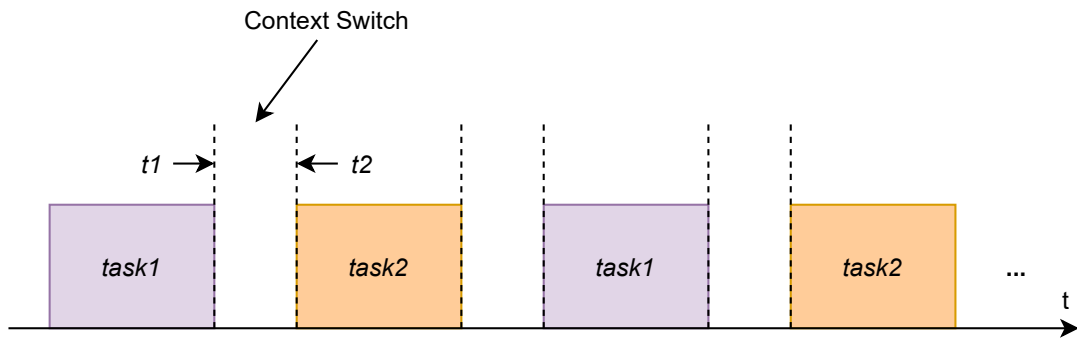


Figure 2.2: Illustration a context switch.

To measure the latency for a response to an interrupt, the application `cyclictest` can be used, which is one of the most frequently used benchmarks for evaluating the performance of real-time systems [23]. It measures latencies by running one master thread which starts a specified number of threads with a given priority, called measuring threads [23]. A measuring thread, in turn, will be woken up periodically with a predefined interval of a timer. The difference between the programmed wake-up time and the true wake-up time is calculated and the maximum, minimum,

and average latencies will be printed out. In short, a group of periodic measuring threads will measure the difference between the expected wake-up time and their actual wake-up time. A few factors that may increase this latency can be interrupt handling, the invocation of the scheduler code, and the time it takes to start the program. Delays can also be caused by higher priority programs that are currently running [24]. `cyclictest` is described more thoroughly in Section 2.6.5.

2.4 The boot process

The boot process can briefly be described as all the processes and tasks that are carried out to make a computer ready to be used when powered on. The process varies from device to device but is generally quite similar. One of the key components in the boot process is the *bootloader*.

When an embedded platform is powered on, the OS is normally not loaded into the RAM straight away [25]. Instead, the CPU starts to execute an initial piece of code which is the bootloader. A bootloader can initialize necessary hardware, find and load another program into the memory from non-volatile storage and then execute it. This next program to be run can in turn be another bootloader since it is not uncommon to use a bootloader to load other bootloaders in multiple stages. Such bootloaders are referred to as multi-stage bootloaders. Finally, the OS code will be loaded and executed.

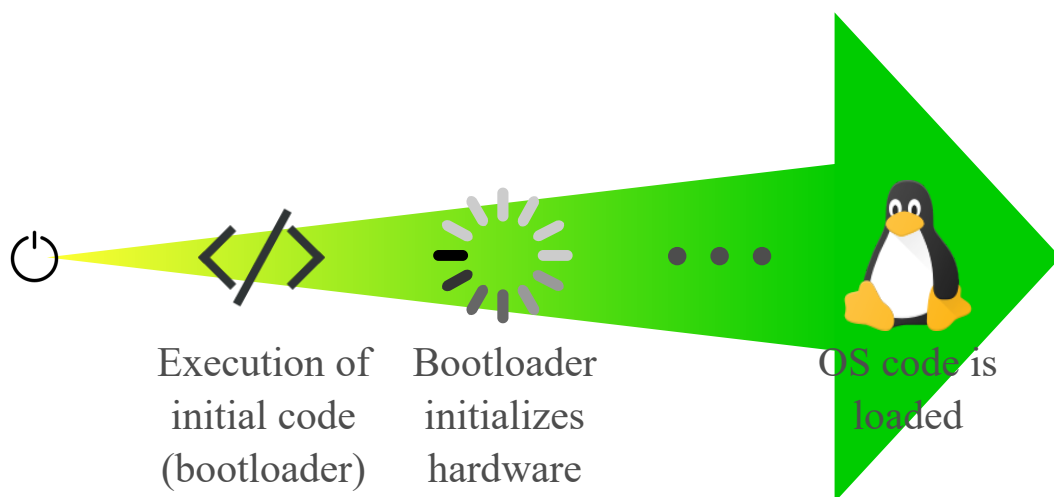


Figure 2.3: An abstract view of how the boot process happens in multiple stages.

2.4.1 Das U-Boot

The previously described boot process corresponds to the process of one of the most common embedded bootloaders, Das U-Boot (U-Boot), which was used throughout this thesis project [25]. It supports different CPU architectures, amongst them are

ARM and PowerPC. U-Boot is an open-source bootloader and can boot a kernel over a file transfer protocol or from an external device such as an SD card. U-Boot also provides an interface (much like a shell) where it is possible to set hardware configurations and necessary environment variables [26].

2.5 Linux

Linux is an open-source OS with the Linux kernel being the at lowest level, and all Linux distributions (distros) are based on this kernel [27]. A large ecosystem is built around the kernel with software components, and a big community is actively maintaining and contributing to further development of this ecosystem. One of the most important aspects of Linux is that it is open-source and free, meaning that it is freely available for anyone to use, distribute and build upon.

This leads to some of the aspects why using Linux is popular. It does not require any licensing to use, it is constantly maintained and got a large supply of tools. Since it is developed by an open source community, there is a lot of help and support that can be found online through a search or by asking for help in forums.

This section introduces how Linux can be applied as an RTOS, as well as the build systems that can be used to produce the components for a functional distro.

2.5.1 Linux as an RTOS

Linux has mostly throughout its time been developed as a general-purpose OS rather than an RTOS. These two objectives are conflicting by nature since the prior focuses on maximizing the throughput, while the other aims to enforce an upper-bound on the execution time of a task [1]. Linux compared to a traditional RTOS, has a fairly complex kernel and it has therefore been a challenge to achieve the determinism that is desired on real-time platforms. However, efforts to achieve determinism have been in the works with the PREEMPT_RT patch for the kernel [28].

PREEMPT_RT Patch

The PREEMPT_RT patch is one of the bases of improving the real-time aspects of the Linux kernel. It gives the option to reconfigure the kernel, so that a larger portion of the kernel code becomes preemptable and unbounded latencies are removed [18]. The patch consists of two additional preemption levels apart from the ones available in the standard kernel [28]. These two additional levels expand the parts of the kernel that can be preempted, minimizing possible unbounded latencies in the system.

Linux with the PREEMPT_RT patch has been used by SpaceX for their on-board computers [4].

2.5.2 Build Systems

There are several tools that support the development of a custom embedded Linux distribution. These tools are commonly referred to as *build systems*. We have considered two popular build systems called Yocto Project (Yocto) and Buildroot in our thesis project.

Yocto Project

Yocto is an open-source project that consists of a reference distribution called Poky and a set of build tools [29]. These tools can be used by a user to manage patches and packages and then produce an image to be deployed on the targeted embedded computer. Yocto has a large ecosystem of users developing packages and contributing to the project. Many silicon vendors contribute to Yocto to provide easy implementation of their Board Support Packages (BSP). A BSP contains software that is essential for running Linux on a specific computer board, including device drivers and the bootloader [30].

Yocto uses a build tool called BitBake that can be used to build a bootable image for a target embedded computer. BitBake acts as a task scheduler that builds up a dependency tree and then uses *recipes* and configuration files to build all the required and specified components. A recipe file (.bb) is similar to a package; it consists of a collection of metadata in a file with information such as a source, descriptions, and instructions on how to install a particular software. The recipes could contain anything from a test tool to a compiler, depending on what should be included in a build. The configuration files (.conf) contains data such as target computer options and distribution configuration options.

Yocto uses a layer model to simplify the development and to ease working with multiple embedded computers. This model makes it possible to enable/disable particular layers in the build. Silicon vendors may create their own layers to provide support for their hardware via Yocto, with components such as BSPs.

Buildroot

Buildroot is another open-source embedded Linux build system which is designed for small to medium-sized embedded systems. It uses fewer software components compared to Yocto, which makes it more straightforward and easier to understand. However, what it can achieve is limited due to its reduced complexity.

Buildroot relies on the Makefile language and uses a configuration interface in which it is possible to manage all configurations for a specified target system. Buildroot is automated in the way that it downloads and builds the necessary packages. It then extracts the source codes, compiles and installs the selected components, and takes care of any dependencies [31].

2.5.3 Components

In this thesis, three of the generated components from the build systems were deployed onto RUAG's hardware. These components were the *root file system*, *device tree* and *kernel image*.

Root File System

A root file system (rootfs) consists of a hierarchy of file directories that are crucial for the system to operate. In Linux, the *root directory*, denoted `/`, is the highest directory of that hierarchy. It contains a series of sub-directories which, in turn, contains further subdirectories. An example of a rootfs structure can be seen in Figure 2.4. The content of the rootfs varies depending on the computer.

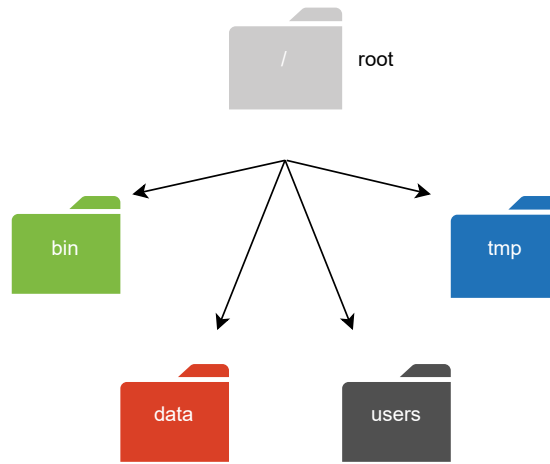


Figure 2.4: Illustration of a root file system with the root directory and some of its sub-directories.

Device Tree

A device tree is a data structure built up by properties and nodes [32]. Properties are key-value pairs, and nodes can contain both properties and child nodes. The task of a device tree is to describe the hardware layout on a particular embedded device by specifying the memory layout, pin assignments, address bindings, etc [33]. This is utilized by the kernel as a map of what hardware is available and how it should manage these different components on the device.

Two different notions are used when talking about device trees and these are *Device Tree Source (DTS)* and *Device Tree Blob (DTB)*. A DTB can also be referred to as a *Flattened Device Tree (FDT)*, although we use the term DTB in this thesis. The DTS and DTB contain the same information but are represented differently. The DTS is a human-readable file that can be converted into a DTB which is a binary object usable by the kernel.

```

/ {
    model = "fsl,P2020RDB";
    compatible = "fsl,P2020RDB";

    aliases {
        ethernet0 = &enet0;
        ethernet1 = &enet1;
        ethernet2 = &enet2;
        serial0 = &serial0;
        serial1 = &serial1;
        pci0 = &pci0;
        pci1 = &pci1;
    };

    memory {
        device_type = "memory";
    };

    lbc: localbus@ffe05000 {
        reg = <0 0xffe05000 0 0x1000>;

        /* NOR and NAND Flashes */
        ranges = <0x0 0x0 0x0 0xef000000 0x01000000
                  0x1 0x0 0x0 0xffa00000 0x00040000
                  0x2 0x0 0x0 0xffb00000 0x00020000>;

        nor@0,0 {
            #address-cells = <1>;
            #size-cells = <1>;
            compatible = "cfi-flash";
            reg = <0x0 0x0 0x1000000>;
            bank-width = <2>;
            device-width = <1>;
        };
    };
};

```

Figure 2.5: An example of how a DTS is structured.

Kernel Image

A kernel image is a binary representation of the kernel. The image can be wrapped with different headers depending on the chosen bootloader [34]. If for example U-Boot is used, the kernel image is called *uImage* and is wrapped with a U-Boot header, making it possible for U-Boot to extract it. At the end of the boot process, the computer begins to execute the kernel image from a predefined state.

2.6 Tools

In this section, the tools used in this thesis are presented.

2.6.1 UART

A UART is a hardware circuit that enables a computer to communicate with other hardware devices via a serial communication link [35]. A UART can be configured with parameters such as baud rate (the rate at which information is being transferred in a communication channel), stop bit (signaling the end of data transmission),

parity (detection if a frame has changed), and bits per byte.

2.6.2 Screen

We used the utility program `screen` for displaying the console of the target systems on the host computer [36]. It can be used to configure a channel connected to UART and through the window see messages from the target board.

2.6.3 Trivial File Transport Protocol

To transfer files from the host computer to the target board, we used a protocol called *Trivial File Transfer Protocol (TFTP)* [37]. This requires that a TFTP server is set up on the host computer. It is then possible to transfer the desired file(s) by accessing the TFTP server via the network or by using an Ethernet crossover cable between the two machines directly. However, there are risks to using TFTP since it does not use any security during the file transfer.

2.6.4 LMBench

LMBench is a suite of benchmark programs that measures system performance on UNIX-related OS [38]. The test called `lat_ctx` measures context switching latency and does so by creating a specified number of processes and, by UNIX pipes, connects them in a ring [39]. Each process will read a token from its pipe and then write it to the next process. The generated numbers of this benchmark can be somewhat inaccurate and vary by about 10 to 15 % for each run.

2.6.5 Rt-tests

It is possible to test and measure a kernel's real-time behavior with the programs included in the `rt-tests` test suite. Two of its programs used in this thesis are `cyclicttest` and `hackbench`.

`cyclicttest`

`cyclicttest` is extensively used in Linux PREEMPT_RT analyses [1]. The test measures the sum of all latencies occurring when the system reacts to an input event. The results of the measurements are expressed in microseconds. In [40], the author states that short tests may not be able to reflect the actual load of a system as infrequent latencies may not even be measured. Therefore, tests should preferably run for a longer time, from a day to a week, depending the usage of the system.

Several parameters need to be considered while setting up a system for `cyclicttest`, and the parameters used in this thesis are:

- **Number of measuring threads** This parameter specifies the number of measuring threads with `--threads (-t)`. It is in general a good idea to have one measuring thread running on each CPU of the target system [41].

- **Thread wake-up interval** The option `--interval (-i)` decides the expected execution period of the measuring threads [42]. When using several measuring threads, `--distance (-d)` is used to specify if the threads should have different wake-up times. For most test situations, it is recommended to run all the measuring threads simultaneously, and in that case `-d` should be set to 0.
- **Thread real-time priority** `--priority (-p)` decides the measuring threads real-time priority [43]. This option should always be specified. The priority must be set so that it is *lower* than whatever is producing the latency. This to avoid a thread's timer interrupting an ongoing measured latency of interest. Moreover, the priority must be set *higher* than that of tasks whose latencies should *not* be measured. On a Symmetric Multiprocessing (SMP) system, `--smp (-S)` can be used to give all measuring threads the same priority.
- **Test duration** By default, the duration is set to infinity and can be stopped manually [44]. The duration and number of iterations can be specified with `--duration (-D)` and `--loops (-l)` respectively.
- **Prevent memory page out** Generally, `--mlockall (-m)` should always be set to prevent pages from being paged out of memory [45].
- **Use `clock_nanosleep()`** `--nanosleep (-n)` will make the measuring threads use the function `clock_nanosleep()` to sleep until their next intended execution time [46]. `clock_nanosleep()` shortens the threads wake-up latency rather than the default interval timer. In general, when running `cyclictest`, this option should always be used.
- **Histograms** With `--histofall (-H)`, a summary of all latencies during a run will be presented at the end of the test [23].

To set these parameters accordingly, it is necessary to have good knowledge about the platform. However, it is important to consider that the result of `cyclictest` may not reflect the exact maximum latencies an application would experience.

While running `cyclictest`, a load as similar as possible to the real-time application for which the system is intended should be executed [47].

hackbench

hackbench is a stress test intended for the kernel [48]. Besides this, it also stresses parts of the memory subsystem by first creating and then destroying threads. The main goal with **hackbench** is to help identify a system's bottlenecks and can be executed simultaneously with `cyclictest`. However, it is not possible to generate the same load as a real-time application would since **hackbench** does not test the communication between devices. A few examples of how to run the program can be found in the manual page [48].

3

RDB Deployment and Benchmarking

The first two sections in this chapter provide the necessary information needed to set up the host and target environment correctly. We then describe a series of attempts to build a Linux distribution for two RDBs provided by RUAG. In the final section we present the results of two benchmarks tests conducted on one of the RDBs, the LS1046ARDB.

3.1 Host Environment Setup

We conducted our work using a desktop computer provided by RUAG, which initially was running the Ubuntu 20.04 distribution of Linux. However, due to compatibility issues in several build environments, we downgraded to Ubuntu 18.04 early in the project.

A TFTP server was set up on the host computer so that files could be transferred via Ethernet onto the target boards. To this end, an additional network card was installed on the host computer, which hosted the TFTP server. The network card was assigned with the IP address 192.168.0.3. It was practical to have the additional network card to be able to access the internet while also being connected to the target board via a Crossover Ethernet Cable.

The software application `screen` was used on the desktop as a console emulator for the target board. The `screen` application was used to communicate with the target system using a serial link connected via USB from the desktop and an adapter to an RJ45-connector on the computer board.

3.1.1 Choice of Build System

There are several approaches to build a kernel and a distribution for Linux. In this chapter, we describe our attempts to build Linux distros using Yocto. We decided to use Yocto because it appeared to have a wide range of advantages. For example, it is widely adopted across the industry, supports several processor architectures, and the output can easily be changed for separate embedded computers [49].

3.1.2 Yocto Setup

Yocto comes in many forms and sizes, and different variants operate slightly differently from each other and a problem may have many solutions. This section will briefly describe how we set up Yocto using the official manual.

First, Yocto needs to be fetched which normally is done using git, which is a free open-source version control software. This is done with the following command:

```
$ git clone git://git.yoctoproject.org/poky
```

The fetched items are a combination of a script to set up the build environment, other build tools, and layers which contain recipes and configuration files. The next step is to set up the build environment, which is done as follows:

```
$ source oe-init-build-env
```

This creates a `build` folder in the source directory. From the source directory, run:

```
$ bitbake <TARGET>
```

where `<TARGET>` is the recipe that should be built. There are various recipes to choose from (it is also possible to make a custom one) depending on what the desired output is. A minimal build can for example be produced by running `bitbake core-image-minimal`, and will be based on the recipe called `core-image-minimal.bb`. BitBake takes the configurations in that file and produces an output that will be placed in the directory `/build/tmp/depoy/images/<MACHINE>`.

An important variable that resides in `build/conf/local.conf` is `MACHINE`. This variable should be set to the name of the target board for which the build is intended. This ensures that the correct BSP for the target platform is included in the build.

A customized or downloaded layer can be added by running:

```
$ bitbake-layers add-layer ../<LAYER>
```

Different releases of Yocto have different code names, of which the three latest are Gatesgarth, Dunfell & Zeus (newest first). We have used all three releases in different attempts of building a Linux distro.

3.2 Target Environment Setup

To get started, we wanted to build a plain Linux distro that could be implemented on the two RDBs. Conveniently enough, both RDBs had previously been set up with the U-Boot bootloader from the factory and could therefore be used immediately.

By running:

```
$ screen /dev/ttyUSB0 115200
```

in the host terminal, we were able to access the target console. Before we could connect the target board to the TFTP server and begin transferring files from the host, a few environmental variables had to be set in U-Boot for the target board. The environment variables *serverip*, *ipaddr*, and *gatewayip* were set accordingly:

```
=> setenv serverip 192.168.0.3
=> setenv ipaddr 192.168.0.1
=> setenv gatewayip 192.168.0.3
```

Once this was done, a ping was sent in U-Boot from the target board to the host to make sure it was reachable and we received a response that the host was alive. We then used the `tftp` command to transfer the necessary files onto the target board. Two arguments were required for the `tftp` command: *address* and *filename*. These arguments, which we describe in Sections 3.2.1 and 3.2.2, vary depending on what files are being used in the boot process. For more details on how the TFTP server was set up and used, see Appendix A. The following commands represent a generalization of how the kernel image, rootfs, and DTB were loaded:

```
=> tftp <0xaddr> <uImage>.bin
=> tftp <0xaddr> <root_file_system>.rootfs.ext2.gz.u-boot
=> tftp <0xaddr> <device_tree_blob>.dtb
```

Another way of loading the required components onto the target board is with a Flattened Image Tree (FIT). The file transfer looks slightly different as the kernel and the DTB are embedded into a single file. The generalization of the file transfer of a FIT-image is:

```
=> tftp <0xaddr> <FIT-image>.bin
```

When all necessary files had been loaded into the board's memory, another environment variable, *bootargs*, needed to be set. With *bootargs*, information can be passed directly to the Linux kernel such as what device is to be used as the rootfs while booting and what serial port to take over after U-Boot. The variable was set differently depending on what build system we used and will be specified in Section 3.2.1 and 3.2.2. However, it will have a similar syntax to:

```
=> setenv bootargs '<passed boot arguments>'
```

Finally, to start the boot process run:

```
=> bootm <address>
```

The initial attempts for getting the desired software to be deployed onto an RDB were carried out on the P2020RDB. Unfortunately, we encountered several problems

trying to implement the PREEMPT_RT patch with Yocto for the P2020RDB. Since we were unable to resolve these problems, we could only produce a working image using Yocto for the LS1046ARDB.

3.2.1 Attempts to build and deploy Linux on P2020RDB

Vanilla Yocto With meta-freescale

NXP have their own BSP layer "meta-freescale" which was downloaded and applied to Yocto. The layer is available for download from the OpenEmbedded Layer Index.¹ It includes build configuration for their boards. Simply by changing the MACHINE variable to "p2020rdb", BitBake will build according to that board's requirement and configuration. The build was performed as follows:

```
$ bitbake core-image-minimal
```

A resulting kernel image, rootfs, and DTB were transferred via TFTP onto the board in U-Boot. However, U-boot was not able to boot the image successfully; the kernel did not start when U-boot tried to hand over control to kernel.

Linux SDK for QorIQ Processors

Our next attempt was to use "Linux SDK for QorIQ Processors" which is a five year old release of Yocto with some additional layers.² The build environment is slightly modified since it is set up for a specific machine in the source folder, rather than a general one.

```
$ . ./setup-env -m p2020rdb
```

This creates a folder called build_p2020rdb with the MACHINE variable already specified as "p2020rdb".

However, after many attempts to build with different versions of Ubuntu on separate host computers, errors were encountered related to the build process. This approach was abandoned because of these problems.

Layerscape Software Development Kit User Guide for Yocto

Our next approach was to use "Layerscape Software Development Kit User Guide for Yocto (LSDKYOCTOUG)".³ It was not initially obvious that this could be used since Layerscape is a subset of newer ARM processors in the larger QorIQ family. However, it was discovered that support for the P2020 was included as well. The

¹Link to the OpenEmbedded Layer Index <https://layers.openembedded.org/layerindex/branch/master/layers/>

²Link for downloading Linux SDK for QorIQ Processors https://www.nxp.com/webapp/swlicensing/sso/downloadSoftware.sp?catid=SDK_ENABLEMENT

³Link to the Layerscape Software Development Kit User Guide for Yocto (LSDKYOCTOUG) <https://www.nxp.com/docs/en/user-guide/LSDKYOCTOUG.pdf>

build environment was set up the same way as for the "Linux SDK for QorIQ Processors".

The current release is based on "Dunfell", the second most recent major release of Yocto. With this build system, a successful build could be made that was able to boot on the P2020RDB. However, to build an image with the PREEMPT_RT patch we had to downgrade to the prior major release "Zeus" since support to build with the patch has not yet been implemented for the "Dunfell" release by NXP. Another attempt was made to build an image with PREEMPT_RT, but errors were encountered once again. Eventually, we found reasons to believe that support for PowerPC is slowly dropping which could be the source of our problems working with the P2020 processor [50]. We therefore decided to move our work with Yocto to the LS1046ARDB board.

3.2.2 Moving ahead with LS1046A

Layerscape Software Development Kit User Guide for Yocto

In another attempt using Yocto, the project proceeded with the "Zeus" release. The git repositories were cloned according to the Zeus branch's readme-file in the QorIQ Yocto SDK repository.⁴ A minimal image was built by running:

```
$ bitbake fsl-image-networking
```

The PREEMPT_RT patch could be added successfully by adding the following line in the `local.conf` file:

```
PREFERRED_PROVIDER_virtual/kernel = "linux-qoriq-rt"
```

By running:

```
$ bitbake linux-qoriq -c menuconfig
```

we entered an interface where the preemption settings could be changed from "No Forced Preemption" to "Preemptible Kernel". The configuration was then saved and exited.

The previously mentioned LSDKYOCTOUG document provides the necessary bootargs and addresses used in U-Boot.

The following commands in U-Boot were then executed:

```
=> setenv bootargs root=/dev/ram0 rw console=ttyS0,115200
earlycon=uart8250,mmio,0x21c0500 ramdisk_size=0x10000000
=> tftp 0x82000000 Image-ls1046ardb.bin
=> tftp 0xa0000000 fsl-image-networking-ls1046ardb.ext2.gz.u-boot
```

⁴Link to the Zeus branch's readme-file in the QorIQ Yocto SDK repository <https://source.codeaurora.org/external/qoriq/qoriq-components/yocto-sdk/tree/readme?h=zeus>

```
=> tftp 0x8f000000 fsl-ls1046a-rdb-sdk.dtb
=> bootm 0x82000000 0xa0000000 0x8f000000
```

Unfortunately, this did not seem to boot as intended. At this point, we started to investigate if it was possible to use the FIT-image as described in the QorIQ Yocto SDK repository. With the same bootargs, the following commands were used to transfer and boot the image:

```
=> tftp 0xa0000000 FIT-image.bin
=> bootm 0xa0000000
```

The boot process ended successfully, and by running `$ uname -a` we could confirm that the kernel had been patched and enabled with `PREEMPT_RT`. With future documentation and measurement in mind, we included the `rt-test` suite to the build by adding the lines:

```
rt-tests \
hwlatdetect \
```

to the variable `IMAGE_INSTALL_append` in the file `fsl-image-networking.bb`. The build was successful and booted properly again.

3.3 Benchmarks

In this section, we present how we executed the benchmark tests and the results of these. The goal of these tests was to compare the context switching latency and the interrupt response latency between a basic (vanilla) version of the Linux kernel and a version that included the `PREEMPT_RT` patch. We used the `lat_ctx` benchmark for measuring the context switching latency, and the `cyclicttest` benchmark for measuring the interrupt response latency. All tests were conducted with the LS1046ARDB and Linux distributions produced by Yocto.

Section 3.3.1 and 3.3.2 describes the set up of `cyclicttest` and `lat_ctx` respectively, while Section 3.3.3 presents the benchmark results.

3.3.1 cyclicttest

Since the LS1046A has four cores, the number of threads for the test was set to four. The number of loops was set to 1,000,000 which takes roughly 20 minutes to finish and the priority was set to 80. Ideally, the tests should run for several hours to reach the edge case for maximal latency. However, to compare distributions rather than investigate the maximum latencies, a shorter period may be sufficient [51].

We conducted two types of benchmarks, one on an idle system and one under system load. These benchmarks were conducted on a vanilla kernel as well as one with the `PREEMPT_RT` patch. The system load was added by running `hackbench` for all four cores and performing enough loops to run at least as long as `cyclicttest`. By

appending "&" after the options, `hackbench` was executed in the background.

```
$ hackbench -T 4 -l 10000000 -p &
$ cyclicttest -t4 -m -n -S -d 0 -l 1000000 -p 80 -q -H 1000
```

From the results of the four tests, Figure 3.1 and Figure 3.2, were generated with *gnuplot* which is a command-line program that can generate plots of data.

3.3.2 lat_ctx

`lat_ctx` may be executed with a varying number of participating processes and different work buffer sizes to cause pressure on the data cache [38, 52]. The varying buffer sizes means that a process does some work before it gets switched [39]. The work corresponds to summing up of an array of a specified size of approximately 2.7 thousand instructions. The effect of this is that the data and instruction cache get polluted resulting in larger context switching latencies. We performed similar tests on both kernels. The benchmark was executed four times per kernel accordingly, where `<buffer size>` was set to 0, 1, 4, and 16 for the different runs. The subsequent numbers are the participating processes that are tested for each run.

```
$ lat_ctx -s<buffer size> 2 4 6 8 10 12 14 16
```

The plots were generated by the programming and numeric computing platform, MATLAB.

3.3.3 Results

Results - cyclicttest

The results from executing `cyclicttest` on an idle system are shown in Figure 3.1. The x-axis represents the latency in microseconds and the y-axis represents how many times a latency occurred. In the figure, we can see that the vanilla kernel had a higher maximum latency than the patched kernel. It can also be seen that the vanilla kernel has a higher number of occurred minimum latencies than the patched kernel. The maximum, minimum, and average latencies for both kernels are compiled into Table 3.1. The minimum latency is the same for both kernels while the vanilla kernel has a lower average latency.

A second comparison was made with the results we got from executing `hackbench` simultaneously with `cyclicttest`. This time, the maximum latencies for the two kernels differ even more. The latency plot is presented in Figure 3.2 and the values of the maximum, minimum, and average latencies are presented in Table 3.2. Again, we can see that for the vanilla kernel, the average latency is lower and the number of occurred minimum latencies are higher compared to the patched kernel.

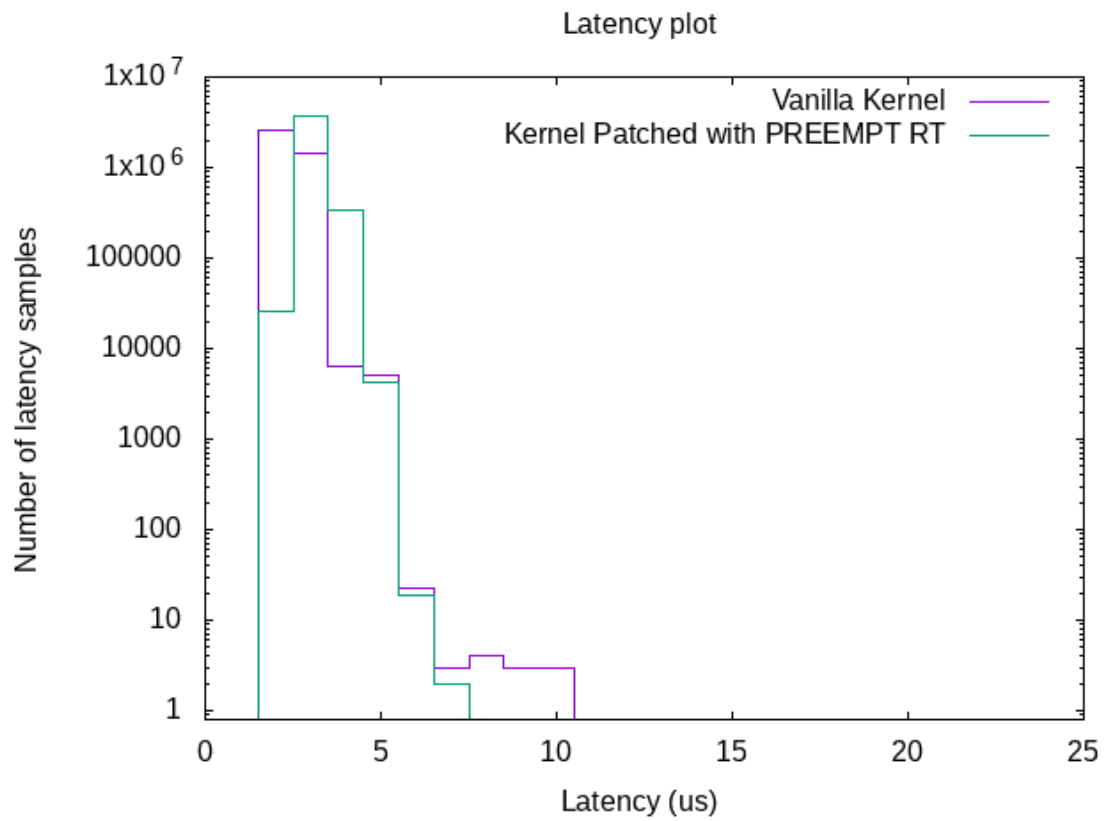


Figure 3.1: Results from `cyclictest` *without* background load for the vanilla and the PREEMPT_RT patched kernels.

Kernel Type	Min. Lat.	Avg Lat.	Max Lat.
PREEMPT	2	3	7
Vanilla	2	2	10

Table 3.1: Minimum, average and maximum latency from `cyclictest` on an idle system.

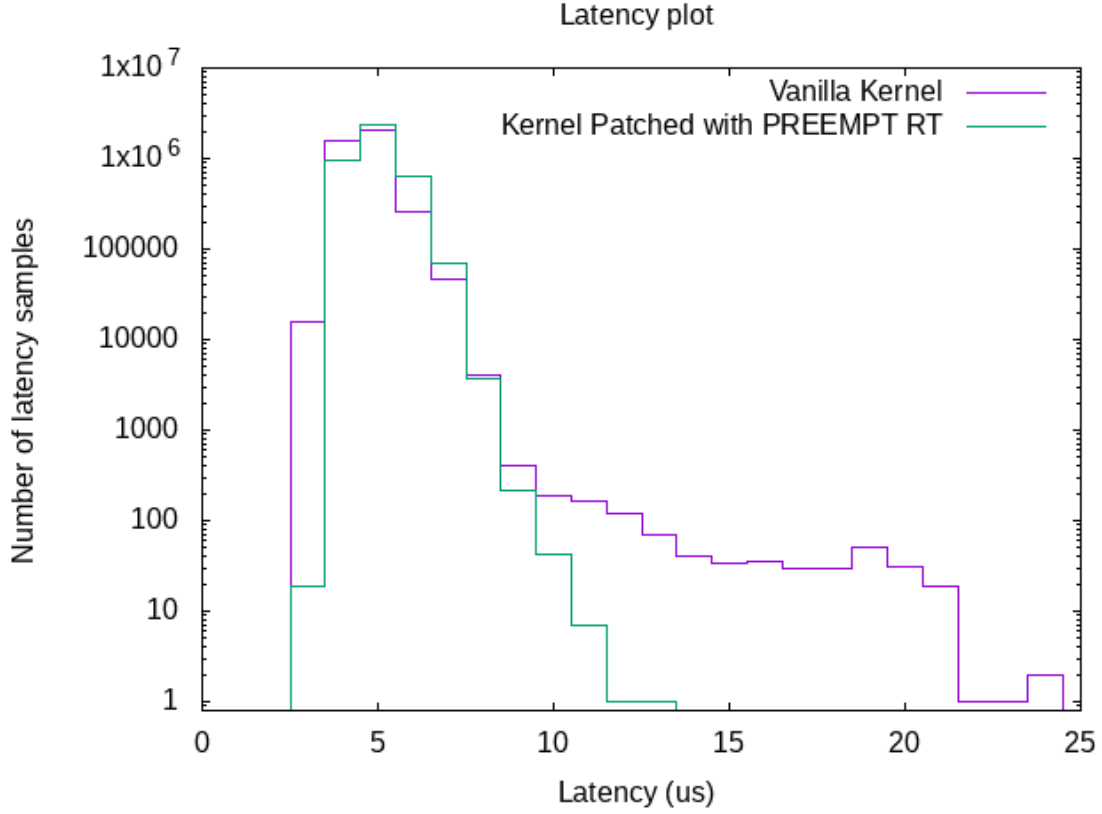


Figure 3.2: Results from `cyclictest` *with* background load for the vanilla and the `PREEMPT_RT` patched kernels

Kernel Type	Min. Lat.	Avg Lat.	Max Lat.
PREEMPT	3	4.5	13
Vanilla	3	4	24

Table 3.2: Minimum, average and maximum latency from `cyclictest` on a system with a background load.

Results - LMbench

`lat_ctx` was executed on an idle system because neither of the prior papers we read mentioned any load on the system [38, 52]. The results for the vanilla kernel are shown in Figure 3.3, while the results for the patched kernel are shown in Figure 3.4. The x-axis in the plots shows the number of participating processes and the y-axis shows the measured task switching latency in microseconds. The four graphs in each plot represent the runs with different buffer sizes. In both plots, it can be seen that the latency is increasing when involving more processes. The most visible difference between the two plots is the behaviour of the graphs. In the vanilla kernel, the measuring points are more scattered while in the patched kernel, the measuring points are increasing in a more consistent fashion. The context switching latencies for the patched kernel are slightly higher for almost all measuring points than for the vanilla kernel. However, an important note is that LMbench reports the average

task switching latency captured and does not include the maximum or minimum task switching latency [38].

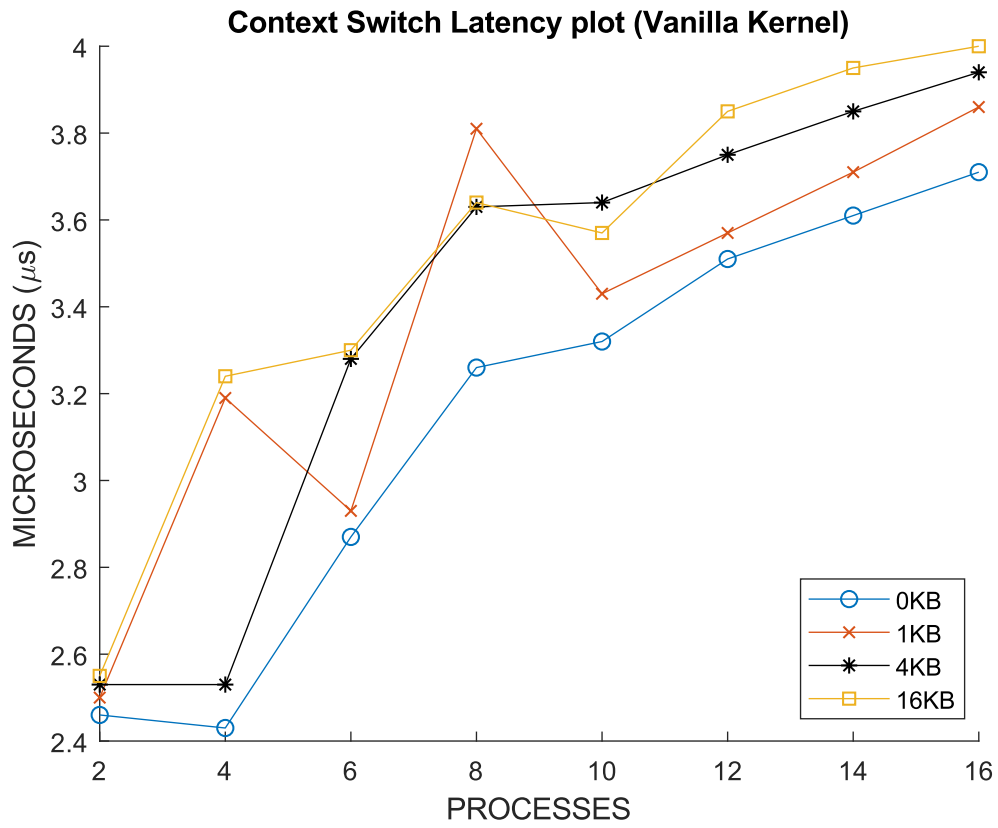


Figure 3.3: Plot based on results from `lat_ctx` for the vanilla kernel.

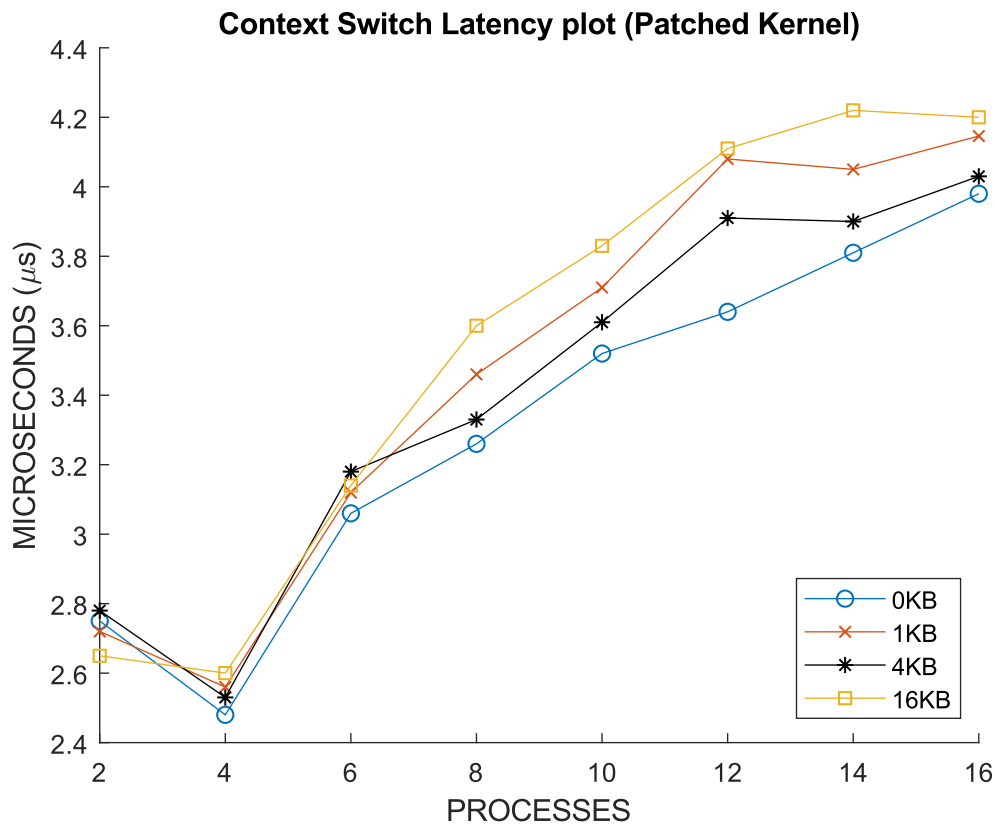


Figure 3.4: Plot based on results from `lat_ctx` for the patched kernel.

4

Pointers for Implementation on the cOBC

Implementation of Linux on the cOBC is not as straightforward as with the RDBs. This is because the cOBC uses its own custom bootloader. The custom bootloader is fairly simple compared to U-Boot, since it only sets up the memory management unit and loads a binary file into the RAM. It then instructs the processor to execute the transferred program. The custom bootloader cannot currently pass components such as rootfs, DTB, and arguments to the kernel.

The first section in this chapter looks into the requirements to implement Linux on the cOBC. The second section looks into how this may be done with the help of boot wrappers for the PowerPC architecture. The final section explains how a Linux distro is built for an embedded computer with the P2020 processor, using Buildroot version 2020.08.

4.1 Requirements to boot on the cOBC

According to the documentation in the kernel source code for the PowerPC architecture, some processor registers must be pre-loaded with specific pointers when the execution of the kernel code begins.¹ These pointers are to the memory address of the board info structure (DTB) and the init RAM disk (rootfs).

*r3 - Board info structure pointer (DRAM, frequency, MAC address, etc.)
*r4 - Starting address of the init RAM disk
*r5 - Ending address of the init RAM disk

Two other registers may contain the start and end address of the kernel command line arguments:

*r6 - Start of kernel command line string (e.g. "mem=128")
*r7 - End of kernel command line string

¹Link to kernel execution entry point code where the processor registers are described https://github.com/torvalds/linux/blob/master/arch/powerpc/kernel/head_fsl_booke.S

However, the command line arguments could also be supplied via the "chosen" node in the DTS that passes data between the firmware and the Linux kernel [53].

```
chosen {  
    bootargs = "<Bootargs to pass to the kernel>";  
};
```

Another way to fulfill these requirements can be done with the help of a boot wrapper.

4.2 Boot wrappers

U-Boot is the most commonly used bootloader for embedded Linux systems [25]. For a kernel image to be bootable, U-Boot (or any other bootloader being used) must be able to extract it. A boot wrapper can be used to generate target images by using a "wrapper script". The script wraps the kernel image with a header and generates a single binary file that is recognizable by the bootloader [54]. An example of such a target image is the uImage which is a kernel image wrapped with the U-Boot header.

For RUAG's custom bootloader, it would be required to create a self-contained binary file that does not rely on a specific bootloader. We suggest a wrapped image containing the DTB and rootfs which is then executed from any location in the RAM.

According to the official Linux kernel documentation, there are two target images that are available for the PowerPC architecture that potentially could be used to solve the problem [54]. These are *zImage* and *simpleImage*.

zImage

A zImage is a compressed version of the Linux kernel that self-extracts. According to the documentation in the wrapper script, it says that *"This script takes a kernel binary and optionally an initrd image and/or a device-tree blob, and creates a bootable zImage for a given platform"*² However, this contradicts the kernel documentation for PowerPC which states that a zImage is an *"Image format which does not embed a device tree."* and *"This image expects firmware to provide the device tree at boot"* [54].

Either way, Buildroot does not seem to provide this support for our particular processor core from what we have seen in the configuration. Attempts were also made to manually use the wrapper script to generate the zImage, but were unsuccessful. It may be possible to implement support for this, but further investigation is required.

²Link to the wrapper script located in the kernel source tree <https://github.com/torvalds/linux/blob/master/arch/powerpc/boot/wrapper>

simpleImage

simpleImage is another target image available for PowerPC which fits the criteria of not relying on the bootloader according to the kernel documentation [54]. However, in a Github-file it says that the only architecture supporting the simpleImage-format is the MicroBlaze architecture, an embedded processor optimized for implementation in Xilinx FPGAs [55].³ In the Buildroot interface, this was proved to be the case because it was not possible to create a simpleImage for the P2020. Since there is close to no documentation regarding simpleImage, it is difficult to determine if anything in particular can be done to circumvent this.

4.3 Build for P2020/cOBC with Buildroot

The same host setup was used in this section as in Chapter 3. The configuration for Buildroot was made in the menuconfig interface shown in Figure 4.1. It was initiated via the command

```
$ make menuconfig
```

in the Buildroot source folder.

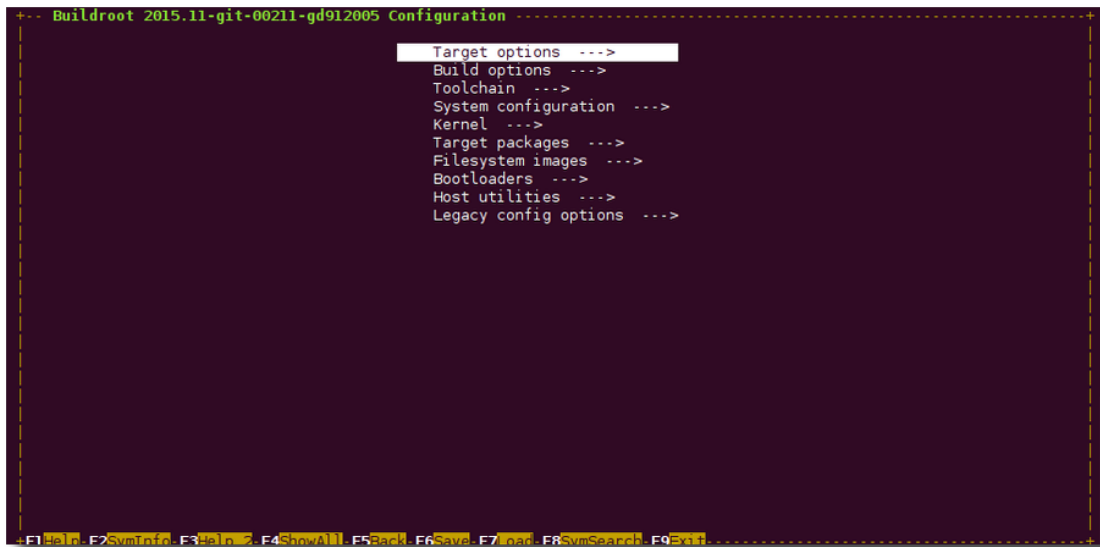


Figure 4.1: The menuconfig interface where it is possible for a user to choose Buildroot configuration.

In the interface, features were added manually within the headings "Target Options", "Kernel" and "File system images". When the configuration is finished, it can be saved in a *.config* in the root folder. To build according to the configuration, run:

³Link to the Github-file containing information about the configuration options <https://github.com/buildroot/buildroot/blob/master/linux/Config.in>

```
$ make
```

The output is placed in the folder `./output/images`.

We generated a successful build for the P2020RDB as a starting point that can be further developed for the cOBC. In Appendix B, the build configuration is specified as a *defconfig* file, which is a configuration file with the default options omitted. Below are some explanations of the relevant categories, and how the options were set.

Target Options

Here, the target architecture and processor core are specified according to the processor. For the P2020, the architecture was set to PowerPC and e500v2 as the core.

Kernel

Here, a choice can be made to build with a recent kernel version or to build from a custom kernel tree provided via sources such as git repositories or tarballs. In this build, a git repository was specified as a kernel source provided by NXP patched with PREEMPT_RT.⁴

Building a DTB can be done from an In-Tree or Out-of-Tree DTS. In-Tree means that a module is already built into the kernel source while an Out-of-Tree module is loaded locally. In the build for the P2020RDB, an In-Tree DTS was used since there is one available from the kernel source. For the cOBC however, an Out-of-Tree DTS would be necessary since RUAG has their own DTS for their custom board.

The kernel also needs a kernel configuration which can be provided via a defconfig file. Just like the DTS, it may be provided In-Tree or Out-of-Tree. In our case, we used a defconfig called *mpc85xx_smp_defconfig*. In Yocto's meta-freescale layer, the same defconfig is specified for the P2020RDB. Note that the defconfig is available In-Tree but not as an actual file, but rather a combination of the following defconfigs merged into one:

```
mpc85xx_basic_defconfig, 85xx-32bit, 85xx-smp, 85xx-hw, fsl-emb-nonhw
```

It is not enough to fetch and use a patched kernel, but the real-time configurations have to be activated in the defconfig as well. Since we cannot edit an In-Tree defconfig, it was downloaded from the source, edited, and provided Out-of-Tree. The following lines were added to the downloaded defconfig:

```
CONFIG_PREEMPT=y  
CONFIG_PREEMPT_RT_BASE=y  
CONFIG_PREEMPT_RT_FULL=y
```

⁴Linux Tree for QorIQ support - Kernel Patched with PREEMPT_RT <https://source.codeaurora.org/external/qoriq/qoriq-components/linux/tree/?h=linux-4.19-rt>

A *kernel binary format* can be selected for a build to compress or wrap the kernel image with the desired header. For the RDB, we selected zImage since it can be used by U-Boot.

File system images

Here, options are set for the rootfs. For example, if the build should generate a rootfs and what compression to use. These options may be specified according to the deployment method for the target. The enabled options for this build made it possible to boot on the P2020RDB.

5

Discussion

This chapter discusses the separate build systems presented in in Chapter 3 & 4, and the benchmark results from Chapter 3. It also compares some of the different building blocks of Linux, and VxWorks as a complete alternative to Linux. The chapter also discusses the challenges of implementing Linux on the cOBC.

Yocto Vs. Buildroot

For this thesis project, we initially opted for Yocto as our build system, but later on we decided to transition over to Buildroot due to difficulties we encountered with Yocto.

Our literature study showed that both Yocto and Buildroot were the main options for porting Linux to embedded systems. One of the main reasons we initially chose Yocto was because a lot of information and support from NXP's side was provided via Yocto. Even though Yocto is more complex and can be difficult to work with as a newcomer, getting started was quite easy in our experience. The Yocto documentation is well written and contains a starter guide to help new users with their very first build. However, as soon as we wanted to make even minor changes, we perceived a steep learning curve because of the significant amount of research and investigation needed. Another challenge is the confusing workflow since it differs from traditional desktop and server software development. Packages are for example added by modifications in the configuration instead of from Internet-hosted package libraries [56].

Even though both Yocto and Buildroot have the same goal of generating the separate Linux components, they have their separate use cases [57]. Buildroot focuses on simplicity and is straightforward to use. The codebase comes in under 1,000 lines, making it straightforward to generate a simple distro. Yocto, on the other hand, has a codebase of over 60,000 lines of code and have far more packages available. There are many ways to solve a problem, which can be frustrating since different sources can become somewhat contradicting. This also makes it challenging to learn the best practices.

In retrospect, we are sure that if we had chosen Buildroot over Yocto at the beginning of this thesis we would have achieved a successful build much sooner. SpaceX, who

has explored more of the work with building Linux distros suitable for spacecraft, has also opted for using Buildroot [4].

A third build system we encountered was Flex-builder, which is developed by NXP. However, it was not extensively explored since it does not seem to be widely adopted yet and lacks support for the P2020 processor.

How do we interpret the benchmark results?

An interesting note to highlight from the results of `cyclictest` is the obtained average latencies. The vanilla kernel has a lower average latency in both cases, seen in Table 3.1 and 3.2, and this can be somewhat surprising. However, there is an explanation for this. Degraded throughput and increased minimum latencies are the costs of determinism. In [58], the author explains it as *"... real-time improvements come at the cost of possible performance degradation. An RTOS may (or may not) sacrifice performance to become more deterministic"*. In Figures 3.1 and 3.2, it is seen that the vanilla kernel does indeed have a higher number of occurred minimum latencies compared to the patched kernel. However, regarding real-time systems, we care more about the maximum latency since timing guarantee is more important than fast execution time. As seen in both tables, the maximum latency is lower on the kernel patched with `PREEMPT_RT`.

The results obtained by executing `lat_ctx` were rather expected. The graphs in Figures 3.3 and 3.4 behave differently. In the first figure, the measuring points are scattered while in the second figure, the measuring points is increasing consistently. Another thing we mentioned about the results were that the patched kernel's measuring points was higher than the vanilla kernel. We believe the reason for this is that the measuring points are based on the average latency. If the values were based on the maximum latency, we suspect the kernel patched with `PREEMPT_RT` would have had lower values than the vanilla kernel.

The tests clearly show differences in the results and, as we expected, the `PREEMPT_RT` patch has increased the deterministic behavior of our Linux distro. Even though this can be established it is still too difficult to define how hard or soft the real-time system is. It is up to the customers to determine if the system meets their requirements based on what applications will run on the platform in the future, which is why it is still important to conduct tests thoroughly.

What could be done differently in the benchmarks?

As stated in Section 2.6.5, `cyclictest` should preferably run for a longer time to be able to identify the edge cases for maximum latency and generate even more accurate numbers.

`lat_ctx` could have been run together with `hackbench`, as it was done for `cyclictest`, to see if the differences between the two kernels become even clearer for a system under load.

Another factor influencing the results is the system load. The most accurate latency measurements are made while running the actual real-time application on a system together with other non-real-time applications that are usually running in the background as well [47]. Generating a system load with `hackbench` that should be as close to the final application is complex and difficult since an artificial load may not trigger system latencies as the actual applications would. Therefore, the obtained results may not reflect the actual maximum latencies but are used to make general comparisons in this thesis.

Comparison of building blocks used to build a Linux distribution

The kernel used in the implementation for the cOBC in Chapter 4 was based on the LSDK20.04 version 4.19 (patched with `PREEMPT_RT`). We felt it could be beneficial to choose a Linux kernel intended for NXP QorIQ platforms since the P2020 is a part of the QorIQ family. Due to a lack of time, we did not investigate the differences between the vanilla kernel and the one provided in the QorIQ Linux tree.

Another building block we considered to investigate further was the C library (`libc`). A `libc` is a set of named functions that can be used in programs as they are reliable and optimized for performance [59]. The small `libc` used in the builds we produced is a library for developing embedded Linux systems called `uClibc`. It is much smaller than the GNU C Library (`glibc`) which is by far the most widely used `libc` on Linux [60]. In addition, practically all applications supported by `glibc` work perfectly with `uClibc` as well [61].

Towards the end of the thesis project we found a `libc` called `musl` (pronounced muscle). In a Q&A held by developers at SpaceX, they acknowledged that `musl` has been used in their spacecrafts [4]. As an alternative to `glibc` and `uClibc`, its goal is to meet the needs of tiny embedded systems but also typical desktops and servers [62]. In the Embedded Linux Conference in 2015, Rich Felker presented several reasons for switching from `uClibc` to `musl`, two of these being due to technical benefits for `musl` as well as `musl`'s project health [63]. `musl` supports both the PowerPC and ARM architecture and is an option in Buildroots toolchain menu [63].

In general, we believe that starting with a minimal build and then including the necessary packages is a favorable way to proceed. This to avoid unnecessary programs interfering with the critical ones on the computer.

Improvements of real-time properties

It has already been established that `PREEMPT_RT` can be used to improve the real-time properties of Linux [4].

Other factors improving the real-time properties could be to avoid unnecessary packages, as was discussed above. Other tools we have come across but not investigated further are Xenomai and RTAI. Xenomai is a framework cooperating with the Linux kernel to provide hard-real time support for user-space applications among other

things. RTAI is a co-scheduler and an interface that allows a user to write applications with strict timing constraints for Linux.

Comparison between VxWorks and Linux with PREEMPT_RT

Wind River themselves claims in the VxWorks product overview that it is the industry's most trusted and widely deployed RTOS for mission-critical embedded systems that must be secure and safe [20]. The product overview also states that companies, regardless of industry or device type, can rely on the VxWorks pedigree of security, safety, high performance, and reliability. Embedded Linux with the PREEMPT_RT patch, however, has also been shown to be effective in mission-critical embedded systems but due to the project's time frame, we did not perform any tests to compare real-time properties for these OS.

Linux and VxWorks have their architectural differences but share UNIX-like features like a shell and shared memory. Furthermore, even though there are fundamental differences in their implementation, they share similarities like almost identical interfaces for accessing devices and conceptual I/O interfaces [64].

Linux is open-source and thereby open freely for anyone who wishes to download it. VxWorks, however, is proprietary and entails a cost for a license. Wind River has not provided any pricing information for VxWorks but can be obtained through a request for a custom quote. We have reasons to believe that a single developer license costs approximately \$18,000-\$20,000 according to a forum where anyone can make an entry.¹

The popularity of Embedded Linux is increasing, and in 2019, EE Times (Electronic Engineering Times), an electronics industry magazine, did a market study for several purposes [65]. These were to identify used technology, aspects of the embedded development process, and used operating systems, amongst other things. The participants in the study were subscribers to EE Times. Regarding operating systems, participants answered a question about what operating systems they are currently using and what operating systems they are considering using in the next 12 months. Embedded Linux got 21% and 31% on the first and second questions respectively, an increase of 10 percentage points, while VxWorks got 5% on both questions. This shows an increasing interest in Embedded Linux. The data is based on 958 answers and is claimed to be highly projectable at 95% confidence with +/-3.15%.

Implementing Linux on the cOBC

While building a Linux distro intended for the cOBC we encountered several difficulties. Although a complete Linux distro with the PREEMPT_RT patch could successfully boot on the P2020RDB at last, we experienced a few obstacles making it more difficult than necessary. Firstly, combining the three components: the kernel image, DTB, and rootfs into a single binary file for RUAG's custom bootloader appeared to be more difficult than expected. Initially, we considered generating

¹Link to VxWorks Wiki discussion <https://en.wikipedia.org/wiki/Talk%3AVxWorks>

a simpleImage or a zImage containing the three components but simpleImage was not available for the P2020 in Buildroot and zImage did not contain all three components. Due to this, we consider that it would be advantageous to use a widely adopted bootloader like U-Boot since all build systems mentioned in this thesis can produce an output that can be booted with U-Boot.

Another already mentioned difficulty about working with the P2020 was that we found reasons to believe that support for PowerPC is slowly dropping. As we mentioned in Chapter 4, Buildroot had to be downgraded from version 21.02 to 20.08 to generate a successful build for the P2020. It is disadvantageous to work with outdated software and hardware since the support and documentation will eventually not be able to address problems that may arise. An advantage, however, for working with older processor architectures is of course that they are proven to work after many years of usage. Regarding cosmic radiation, larger transistors which are used in older processors are less vulnerable to cosmic rays because smaller transistors require a less electrical charge to flip between binary states [66]. Hence, smaller transistors are more vulnerable when getting struck by cosmic rays.

Besides the processor and the custom bootloader, we did not experience any other difficulties working with the provided hardware and we believe that as long as the DTS describes the board's hardware correctly, no further problems should arise.

6

Conclusion

The goal of this thesis was to investigate the possibilities to use Linux as an RTOS on space computers, partly by investigating if and how an implementation could be made on RUAG's computers. We have used the build systems Yocto Project and Buildroot to build Linux distributions that we were able to boot on two different RDBs provided by RUAG. The main contributions of this thesis are:

- An evaluation of the build systems Yocto and Buildroot.
- Results of the benchmark tests `cyclictest` and `lat_ctx`.
- Guidelines and suggestions on how RUAG can continue to work with Linux on their cOBC.
- A high-level comparison of the pros and cons of Linux vs VxWorks.

Regarding the build systems we can, through our observations, conclude that Buildroot is more user-friendly than Yocto. We find it easier to work in an interface which Buildroot provides rather than working with Yocto's many configuration files. However, we perceive that Buildroot can be more limited in a build because of its smaller code base and fewer available packages.

The results we got from running `cyclictest` and `lat_ctx` on the LS1046ARDB indicate that the `PREEMPT_RT` patch has improved the deterministic behaviour of our tailored kernel. This was compared to a kernel without the `PREEMPT_RT` patch. The maximum delay it takes for a system (with and without a running background load) to react to an interrupt has decreased significantly. We have also observed how the context switching latency in the patched kernel is increasing in a more consistent fashion when the number of involved processes are increasing. Because of these improvements, and as it is in use for other space applications today, we believe Linux is suitable on platforms having hard real-time requirements. We see the possibilities of RUAG meeting customers interest in using Linux on their cOBC in the future.

There are a vast number of decisions that can be made to build a tailored Linux distro. A couple of suitable building blocks we have discussed are the kernel and the choice of libc. We found it beneficial to use the kernel from the QorIQ Linux tree

since it was provided by NXP. We also believe that the libc musl would be a good addition because of its small size and large functionality. Besides the patch, we have discussed that the chosen features and building blocks of a Linux distro should also be carefully evaluated. This to reduce the chance of installing interfering packages, and the size of the build.

In the original plan for our thesis project, we had the intention to investigate how Linux and COTS hardware could be protected against cosmic radiation by redundancy techniques. However, due to the project's timing constraints, we did not address this topic in the thesis.

6.1 Future Work

Besides Yocto and Buildroot, there is Flexbuild which is another build framework that can generate various components including a Linux distro. Other frameworks and applications like Xenomai and RTAI can be utilized to improve the real-time properties of a Linux distribution. To create an understanding of which approach is the best, a thorough investigation should be made to compare these build tools, frameworks, and applications against each other to explore their advantages and disadvantages.

As the libc musl was not included in our presented builds, it would be of interest to include it in future builds and conduct tests to study if musl brings any advantages.

Since neither a `simpleImage` nor a `zImage` could be created in Buildroot to include a kernel image, DTB, and rootfs, a solution could be to develop a custom wrapper to do this instead. Another approach could be to implement support for the custom bootloader that would pass static pointer addresses to the processor registers. It could then make sure to load the binaries of the core components into the correct addresses in memory.

Bibliography

- [1] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time linux kernel: A survey on preempt_rt. *ACM Computing Surveys*, 52:1–36, 02 2019.
- [2] Rtos for safety critical systems. <https://www.intervalzero.com/rtos/rtos-for-safety-critical-systems/>. Accessed: 2021-05-22.
- [3] Inc. Wind River Systems. Over 20 years in space. <https://www.windriver.com/inspace/>. Accessed: 2020-11-19.
- [4] We are spacex software engineers - we launch rockets into space - ama. https://www.reddit.com/r/IAmA/comments/1853ap/we_are_spacex_software_engineers_we_launch/, February 2013. Accessed 2021-03-25.
- [5] Dragon’s ”radiation-tolerant” design. <https://aviationweek.com/dragons-radiation-tolerant-design>, November 2012. Accessed 2021-03-25.
- [6] RUAG Space. Space. <https://www.ruag.com/en/products-services/space>. Accessed: 2020-11-19.
- [7] The European Space Agency. Onboard computers and data handling. https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Computers_and_Data_Handling/Onboard_Computers_and_Data_Handling. Accessed: 2021-02-05.
- [8] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.
- [9] What is radiation hardening? <https://phoenixwi.com/materials-testing/radiation-hardening/#whatis>. Accessed 2021-05-20.
- [10] Computer Security Resource Center. commercial-off-the-shelf (cots). https://csrc.nist.gov/glossary/term/commercial_off_the_shelf. Accessed: 2021-02-05.
- [11] QorIQ p2010 and p2020 processors. <https://www.nxp.com/docs/en/fact-sheet/QP20XXFS.pdf>. Accessed 2021-05-20.
- [12] NXP. Layerscape 1046a and 1026a processors. <https://www.nxp>.

- com/products/processors-and-microcontrollers/arm-processors/layerscape-processors/layerscape-1046a-and-1026a-processors:LS1046A. Accessed 2021-03-25.
- [13] Mahesh Parahar. Difference between operating system and kernel. <https://www.tutorialspoint.com/difference-between-operating-system-and-kernel>. Accessed 2021-02-23.
 - [14] Vincenzo Gulisano. Operating systems – EDA093/DIT401 introduction to operating systems, September 2019. Chalmers University of Technology. Lecture notes. Accessed 2021-02-10.
 - [15] Nikita Ishkov. A complete guide to linux process scheduling. Master's thesis, 2015.
 - [16] What is an embedded os? design support. <https://www.dsl-ltd.co.uk/support/embedded-os/>. Accessed 2021-06-04.
 - [17] The Linux Foundation. Intro to real-time linux for embedded developers. <https://linuxfoundation.org/blog/2013/03/intro-to-real-time-linux-for-embedded-developers/>. Accessed: 2020-12-04.
 - [18] THE LINUX FOUNDATION. Intro to real-time linux for embedded developers. <https://www.linuxfoundation.org/blog/2013/03/intro-to-real-time-linux-for-embedded-developers/>, March 2013. Accessed 2021-03-25.
 - [19] Danilo Mocerì. *Benchmarking of real-time embedded Linux devices*. PhD thesis, Politecnico di Torino, 2018.
 - [20] Inc Wind River Systems. Vxworks product overview. <https://resources.windriver.com/vxworks/vxworks-product-overview>. Accessed 2021-04-23.
 - [21] M. Li, Z. Shang, Q. Hu, G. Yang, Y. Li, and F. Sun. Analysis and testing of key performance indexes of vxworks in real-time system. In *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 369–374, 2018.
 - [22] What is context switching in operating system? <https://afteracademy.com/blog/what-is-context-switching-in-operating-system>, November 2019. Accessed 2021-05-20.
 - [23] The Linux Foundation. Cyclicttest. <https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclicttest/start>, August 2018. Accessed 2021-04-19.
 - [24] Frank Rowand. Using and understanding the real-time cyclicttest benchmark. In *Embedded Linux Conference*, volume 58, page 58, 2013.
 - [25] John Bonesio. Bootloaders for embedded linux systems. <https://>

- thenewstack.io/bootloaders-for-embedded-linux-systems/, Oct 2020. Accessed 2021-03-18.
- [26] Jagdish Gediya, Jaswinder Singh, Prabhakar Kushwaha, Rajan Srivastava, and Zening Wang. 7 - open-source software. In Robert Oshana and Mark Kraeling, editors, *Software Engineering for Embedded Systems (Second Edition)*, pages 207–244. Newnes, second edition edition, 2019.
- [27] What is linux? <https://www.linux.com/what-is-linux/>. Accessed 2021-05-20.
- [28] Kconfig.preempt. <https://github.com/torvalds/linux/blob/master/kernel/Kconfig.preempt>. Accessed 2021-03-25.
- [29] About the yocto project. <https://www.yoctoproject.org/about/>. Accessed 2021-03-25.
- [30] Openest. Board support package for linux. <https://openest.io/en/2019/12/30/board-support-package-linux-bsp-what-is-it/>, 2019. Accessed 2021-03-02.
- [31] Thomas Petazzoni and Free Electrons. Buildroot: a nice, simple and efficient embedded linux build system. In *Embedded Linux System Conference*, volume 2012, 2012.
- [32] Device tree usage. https://elinux.org/Device_Tree_Usage. Accessed 2021-05-11.
- [33] fdt flattened device tree support. <http://manpages.ubuntu.com/manpages/focal/man4/FDT.4freebsd.html>. Accessed 2021-05-11.
- [34] Xilinx Inc. U-boot images. <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842374/U-Boot+Images>. Last updated on October 22, 2019 by Terry O’Neal. Accessed 2021-03-09.
- [35] Cypress Semiconductors. Universal asynchronous receiver transmitter (uart) 2.0. <https://www.cypress.com/file/132486/download>. Accessed 2021-02-23.
- [36] GNU. *Screen User’s Manual*, 4.6.2 edition. Chapter 1 – Overview.
- [37] K Sollins. The tftp protocol (revision 2). Technical report, STD 33, RFC 1350, MIT, 1992.
- [38] Gilles Chanteperdrix and Richard Cochran. The arm fast context switch extension for linux. In *Real Time Linux Workshop*, pages 255–262, 2009.
- [39] Carl Staelin and Larry McVoy. lat_ctx - context switching benchmark. http://manpages.ubuntu.com/manpages/trusty/lat_ctx.8.html. Accessed 2021-05-11.
- [40] Carsten Emde. Long-term monitoring of apparent latency in preempt rt linux realtime systems. *RTLWS12*, 2010.

- [41] The Linux Foundation. Cyclictest - number and affinity of measuring threads. <https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/options/threads>, August 2018. Accessed 2021-04-19.
- [42] The Linux Foundation. Cyclictest - thread wake-up interval. <https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/options/interval>, August 2018. Accessed 2021-04-19.
- [43] The Linux Foundation. Cyclictest - thread real-time priority. <https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/options/priority>, August 2018. Accessed 2021-04-19.
- [44] The Linux Foundation. Cyclictest - test duration. <https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/options/duration>, August 2018. Accessed 2021-04-19.
- [45] The Linux Foundation. Cyclictest - prevent memory page out. <https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/options/mlockall>, August 2018. Accessed 2021-04-19.
- [46] The Linux Foundation. Cyclictest - use clock_nanosleep(). <https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/options/nanosleep>, August 2018. Accessed 2021-04-19.
- [47] The Linux Foundation. Cyclictest - test design. <https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/test-design#option-selection>, 08 2018. Accessed 2021-03-18.
- [48] Ubuntu Manpage Repository. hackbench - scheduler benchmark/stress test. <http://manpages.ubuntu.com/manpages/xenial/man8/hackbench.8.html>, August 2018. Accessed 2021-04-19.
- [49] Linux Foundation. Yocto project overview and concepts manual. <https://www.yoctoproject.org/docs/3.1/overview-manual/overview-manual.html#gs-features>. Accessed: 2021-02-01.
- [50] Compilation failure linux-qoriq-rt 4.14. <https://community.nxp.com/t5/T-Series/Compilation-failure-linux-qoriq-rt-4-14/m-p/1006705>. Accessed 2021-05-07.
- [51] Jan Kiszka. Towards linux as a real-time hypervisor. In *Proceedings of the 11th Real-Time Linux Workshop*, pages 215–224. Citeseer, 2009.
- [52] Hyok-Sung Choi and Hee-Chul Yun. Context switching and ipc performance comparison between uclinux and linux on the arm9 based processor. In *SAM-SUNG Tech. Conf*, 2005.
- [53] Grant Likely. Linux and the device tree. <https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt>. Accessed 2021-05-19.
- [54] The powerpc boot wrapper. <https://www.kernel.org/doc/html/latest/powerpc/bootwrapper.html>. Accessed 2021-05-12.

- [55] Inc Xilinx. Microblaze processor reference guide. https://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf. Accessed 2021-05-17.
- [56] Drew Moseley. Why the yocto project for my iot project? <https://www.embedded.com/why-the-yocto-project-for-my-iot-project/>, July 2017. Accessed 2021-03-02.
- [57] John Bonesio. Deciding between buildroot & yocto. <https://lwn.net/Articles/682540/>, Apr 2016. Accessed 2021-05-13.
- [58] Kushal Koolwal. Investigating latency effects of the linux real-time preemption patches (preempt rt) on amds geode lx platform. *RTLWS11*, 2009.
- [59] C standard library functions. <https://www.programiz.com/c-programming/library-function>. Accessed 2021-05-14.
- [60] libc - overview of standard c libraries on linux. <http://manpages.ubuntu.com/manpages/trusty/man7/libc.7.html>. Accessed 2021-05-14.
- [61] C standard library functions. <https://www.uclibc.org/about.html>. Accessed 2021-05-14.
- [62] About musl. <https://musl.libc.org/about.html>. Accessed 2021-05-14.
- [63] Rich Felker. Transitioning from uclibc to musl for embedded development. https://elinux.org/images/e/eb/Transitioning_From_uclibc_to_musl_for_Embedded_Development.pdf, March 2015. Accessed 2021-05-14.
- [64] Timesys Corporation. Embedded linux development tutorial – porting vxworks® application to linux. <https://www.timesys.com/pdf/Porting-VxWorks-Applications-to-Linux-AppNote.pdf>. Accessed 2021-04-29.
- [65] AspenCore. 2019 embedded markets study. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf, March 2019. Accessed 2021-04-23.
- [66] Peter Dockrill. Rogue cosmic rays from outer space are causing havoc with our smartphones. <https://www.sciencealert.com/rogue-cosmic-rays-from-outer-space-are-causing-havoc-with-our-smartphones>, February 2017. Accessed 2021-05-19.
- [67] A Chalmers University of Technology Master’s thesis template for L^AT_EX. Unpublished. Frisk. D, 2018.
- [68] Shahriar Shovon. Installing and configuring tftp server on ubuntu. https://linuxhint.com/install_tftp_server_ubuntu/, 2019. Accessed 2021-02-18.

A

Appendix A

A short guide for connecting the target board's console to a window at the host computer with **screen**, and then setting up a TFTP server.

1. Display detected serial connections by running

```
$ dmesg | grep ttyUSB
```

on the host. Linux uses the prefix *ttyUSB* for USB based serial ports. Our output was

```
usb 1-10: pl2302 converter now attached to ttyUSB0
```

Hence, the connected serial port is *ttyUSB0* and will be used in the next step when running **screen**.

2. Start **screen** by running

```
$ sudo screen /dev/ttyUSB0 115200
```

The default configurations were already set for us and should be: 8 bits per byte, No parity, 1 stop bit, Flow control hardware: Yes, Flow control software: No. 8N1 Yes No, in short.

3. Start/reset the board and the boot process should be seen in the connected window. When "Hit any key to stop autoboot", hit any key on the keyboard.

The next step is to set up a TFTP server to be able to transfer the necessary files into the memory of the reference design board. We followed a guide to install the *tftpd-hpa* TFTP server package on Ubuntu [68]. To make sure the server was working and reachable, we sent a ping in U-Boot to the serverip (host computers IP address). The response was:

```
=> host <serverip> is alive
```

Then the necessary files should be copied from the image directory into the same directory the TFTP server is running. Files can now be transferred from the host.

B

Appendix B

The defconfig file to build Linux with the PREEMPT_RT patch for the P2020RBD with Buildroot:

```
BR2_powerpc=y
BR2_powerpc_8548=y
BR2_KERNEL_HEADERS_4_19=y
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_CUSTOM_GIT=y
BR2_LINUX_KERNEL_CUSTOM_REPO_URL="https://source.codeaurora.org/external/qoriq/qoriq-components/linux"
BR2_LINUX_KERNEL_CUSTOM_REPO_VERSION="LSDK-20.04-V4.19-RT"
BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG=y
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="mpc85xx_smp_defconfig"
BR2_LINUX_KERNEL_ZIMAGE=y
BR2_LINUX_KERNEL_DTS_SUPPORT=y
BR2_LINUX_KERNEL_INTREE_DTS_NAME="p2020rdb"
BR2_TARGET_ROOTFS_CPIO=y
BR2_TARGET_ROOTFS_CPIO_GZIP=y
BR2_TARGET_ROOTFS_CPIO_UIMAGE=y
BR2_TARGET_ROOTFS_EXT2=y
BR2_TARGET_ROOTFS_EXT2_GZIP=y
# BR2_TARGET_ROOTFS_TAR is not set
```