POLITECNICO DI TORINO

Faculty of Engineering

Master of Science in Computer engineering

Master Thesis

Benchmarking of real-time embedded Linux devices

Advisor:
Prof. Massimo Violante

Candidate:
Danilo Moceri

Company tutor
AROL Group S.p.A.
Mr. Gianpaolo Macario

September 2018

A special thanks goes
to all those who have
encouraged and supported me,
allowing me to reach this
important milestone

Acknowledgements

First of all, I would like to thank my advisor Professor Massimo Violante for his constant support during the development of the whole thesis. Moreover, he provided me with all the needed instruments for the thesis development in order to accomplish the goals.
I would like to thank also AROL Group S.p.A., and in particular, Mr. Marco Cipriani and Mr. Gianpaolo Macario who helped and guided me.
Finally but not least, I would like to acknowledge all those who continuously encouraged me during the thesis development, my girlfriend, and I must express my very profound gratitude to my parents who allowed me to reach this important milestone.

Summary

Industry 4.0 is coming and production lines need new technologies to deal with customers that increasingly ask for new features. From the computer technology perspective, the trend goes toward the use of single board computers equipped with Embedded Linux for dealing with many functionalities required.
In this scenario, where most of the industrial control systems needs real-time performance, developers have to deal with the fact that Linux was not meant to be used in real-time situations. However, over the years, many have been the attempts of making Linux working real-time, and today the most common solution named PREEMPT_RT helps to gain some real-time capabilities.
In this picture, where computing systems offer different time responses one from the other, companies need to choose carefully the right hardware device, and before updating their systems they need to know some numerical measurements in terms of real-time performances to estimate whether the new device will be powerful enough or not.
Considering the previously defined context, this thesis has the main goal of introducing techniques for benchmarking of different devices equipped with embedded Linux for real-time applications.

Contents

List of Figures

List of Tables

Listings

Introduction

The so-called Fourth industrial revolution is putting forward day after day, consequently, the new concept of connected production lines is increasingly required by customers who ask for new features, to be ready as soon as possible, without too much effort and with the lowest price. Of course, those requests rise to new technological challenges which have to be coped in order to keep the pass. Technological challenges come out in almost every field but, from the computers point of view the trend goes toward the use of embedded devices, namely single board computers, that equipped with Embedded Linux are able to deal with most of the required functionalities.
Being Embedded Linux an open source solution, it represents one of the best ways to cope with the increasingly requests of functionalities that have to be managed as soon as possible in order to reduce the time-to-market. However, considering the scenario of industrial control systems, where most of devices need real-time capabilities, developers have to cope with the fact that Linux, as it is, was not meant to be used in real-time applications.
Hence, over the years, many attempts to make Linux working real-time have been done, and today different roads can be taken to get real-time capabilities even into Linux Embedded systems. In fact, there exist solutions that can be easily applied even to standard Kernels, namely, for example, the PREEMPT_RT patch which helps to gain some real-time capabilities.
In this scenario, where the various single board computers offer different real-time performances, the industries require these embedded systems to be reliable in order to run different functionalities, included the control ones that in the past have relied on Programmable Logic Controllers (PLCs) thanks to their trustworthiness. Therefore, new methodologies and tools are needed to let industries choose the right hardware solution that enables the different functionalities respecting the timing constraints.
Unfortunately, due to the complexity of Linux systems, mathematical proofs, claiming that the systems will never miss their deadlines, are not feasible. Hence, some experimental results have to be produced in order to decide whether or not the system is powerful

enough.
Within the previously defined context, this thesis introduces techniques and tools for benchmarking of different devices, equipped with real-time embedded Linux.
During the debate, many concepts and topics will be analysed, specifically, the content will be distributed as follows:

- The **first chapter** introduces the main thesis motivations: as first step, the Industry 4.0 concept will be briefly explained, then the evolution of Industrial Control Systems (ICSs) will be summarized, describing the newer technology devices that integrates multiple functionalities into Single Board Computer (SBC). Finally, the mixed criticality systems will be described and analysed because the previously cited SBC can be easily recognized as belonging to that class of systems when applications with different priorities have to run on the same hardware.

- Once understood that SBC are going to be used in ICS, the **second chapter** deals with Linux for embedded systems: as preliminary step an analysis about the usage of Linux as operating system will be provided, then the main components of a Linux embedded system will be introduced outlining the main differences with Desktop environments. Finally, a summary about the Yocto Project (YP), one of the main way to get a Linux distribution for embedded systems, will be presented.

- Being Linux the chosen operating system for industrial devices, the **third chapter** deals with the fact that Linux was not meant as a real-time operating system: within the first few pages, the main ways to make Linux working real-time will be analysed, then the main effort will be focus on describing the so called PREEMPT_RT patch, which can be applied to mainline kernels and enables the Linux system to gain real-time capabilities. Finally, the concept of scheduling latency and other background informations will be introduced to get into the main tools and methodologies used to measure real-time capabilities of a Linux system.

- Once got an overview of all the needed background information, the **fourth chapter** is about the actual experimental tests and results: before presenting the measurements the under test devices' characteristics will be summarized, then the test scenarios and configurations will be presented. Finally, the obtained results will be analysed, looking further into the main bottlenecks and proposing possible solutions to improve the system real-time capabilities.

- Finally, the **fifth chapter** is about a conclusion that summarizes what has been done and what could be still analysed.

Chapter 1

Motivations

This chapter will present an introduction to the thought of Industry 4.0. Then, a generic model of an industrial control system will be introduced describing its evolution, specifically, the main effort will be focus on introducing the new idea of control systems that integrate different functionalities into a single device. Finally, the concept of mixed criticality system will be presented since the new all-in-one systems can be easily located in this context.

1.1 Industry 4.0

This section has the goal of introducing the reader to the concept of Industry 4.0: the new conception of the industrial environment.

Industry 4.0 is the term used to describe the concept of the Fourth industrial revolution: the main idea behind the new industrialization process regards the creation of interconnected production lines. The Fourth industrial revolution affects manufacturing technologies in terms of automation, data exchange and computing systems, and of course, within this innovative idea, new computing, communication, and automation technologies troubles are going to be faced in order to keep the pass with the increasing demands.

The industry 4.0 trend finds its main support on the Internet of Things and on the cloud computing, these technologies help also in enabling the new concept of smart factories. Within the new smart factories, cyber-physical systems monitor physic processes thanks to the Internet of Things (IoT) technologies that guaranties communication and cooperation between devices and with humans.
The main design principles behind the Industry 4.0 concept can be summarized in: [5]

- **Interoperability:** it is the ability of machines, devices, sensors, and people to connect and communicate each other, very often it is supported by the IoT.
- **Information transparency:** it represents the systems ability to create virtual copies of the physical data for enhancing the model with sensor data.
- **Technical assistance:** it is the ability of systems to be self-assisted through mechanisms that support humans, by aggregating and visualizing information, for making quick decisions and for solving urgent problems as soon as possible. Moreover, the abilities of supporting people during their work for the reduction of unsafe situations or for advising people when they do too much heavy work can be added.
- **Decentralized decisions:** it represents the ability of systems to make decisions on their own, performing their tasks as much autonomously as it is possible.

As said, the main idea of Industry 4.0 is based on connecting machines and systems for creating intelligent networks in which devices can control each other autonomously, collecting useful data and acting accordingly to prevent undesired events and to enhance productivity.

Examples of Industry 4.0 applications are given by devices that can autonomously predict failures, triggering maintenance processes to avoid economic lost, or systems that can self-organize logistics, reacting to unexpected changes in customer requests.

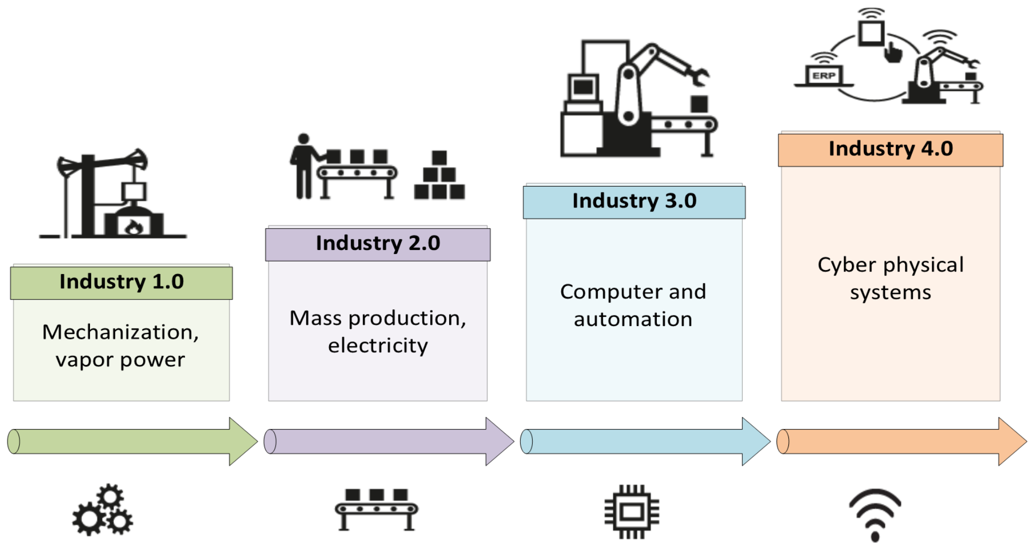


Figure 1.1: Industry evolution

Figure 1.1 shows the main industry evolutions, so far with the so called Third industrial revolution, networks and processes connectivities have been limited just to one factory. Instead, with the Industry 4.0 concept these boundaries are going to no longer exist, indeed, multiple factories will be interconnected without any geographical limit.

Therefore, in an Industry 4.0 factory, in addiction to the already in use methodologies for the monitoring and the fault diagnosis, the devices will be able to self-predict problems and health status, thanks to the integration of information coming from the different factories located worldwide.

Obviously, all these features and functionalities offered by the Industry 4.0 imply new challenges for their implementation. Especially for the computer industry, the following troubles can be easily identified:

- **Security issues:** due to the interconnectivity of many devices, challenges in terms of security violations and information hiding can easily arise.

- **Reliability and stability issues:** in order to make devices working properly and as expected requirements have to be always respected, for example, applications which require real-time behaviour have to be provided with devices and systems able to guarantee that timing requirements are fulfilled without any exception.

- **Flexibility and adaptability issues:** because technology evolution happens really quick, there is the need of having flexible devices and systems to cope with customers' requirements as soon as possible. In fact, time to market always plays a key role in industrial environments and flexible systems are almost a must.

- **Computing capabilities:** for being able to analyse a huge amount of data, powerful devices with good storage capabilities are needed in order to deal with these data.

To sum up, the Industry 4.0 concept improves many system functionalities providing fascinating features, however, to cope with customer requests and to keep the pass, new technological challenges affecting many different disciplines, are going to be faced. As probably understood, within this thesis work, among the various computer challenges the ones inherent to the reliability issues due to real-time behaviour will be treated with major effort, nevertheless, also the flexibility and adaptability issues will be treated indirectly thanks to the usage of open-source solutions highly adaptable.

1.2 Industrial Control Systems evolution

Once an introduction to the Industry 4.0 idea has been presented, this section will introduce a basic model of a generic Industrial Control System (ICS), then the history of these systems will be briefly summarized in order to understand the modern idea of controlling devices integrating many different functionalities. Finally, these new all-in-one devices will be analysed outlining their main advantages and disadvantages.

1.2.1 A generic Industrial Control System (ICS)

What is an Industrial Control System (ICS)?

The term ICS can refer to several types of systems whose main goal is to manage industrial processes using a set of instruments. These systems can include devices, networks and controls, used to operate and automate industrial processes throughout the continuous monitoring of physic processes. ICS are widely used in many industries and fields, such as: electric, transportation, chemical, pharmaceutical, food and beverage, automotive, aerospace, electronic goods, water, oil and natural gas.

When talking about ICS several architectures exist, but, the most commons are Supervisory Control and Data Acquisition (SCADA) systems and Distributed Control Systems (DCSs) [4, 9]. Usually, these kinds of systems are implemented using PLC, more details about these devices will be provided in section 1.2.2.1.
To better understand the evolution of Industrial Control System, it could be useful to have a look at the basic architecture of a generic control system; for this reason, in figure 1.2 the basic control system structure is shown. Of course, this basic model is just an example that helps in understanding how more complex real systems can work.
Generically, an ICS includes sensors, actuators, operator interfaces and last but not least the logic control device.
Sensors measure physical qualities, like: temperature, strength, speed, etc., then, they convert that information into electrical signals, which will be read by the control logic when needed; The actuator devices act accordingly to the information the controller provides them; The operator (or user) interface is where the interaction between the machine and peoples occurs; Finally, the logic device controls all the machine operations and decides the actions to do, i.e. the controller reads inputs from operators and sensors, then elaborates that information and sends signals to the actuators.

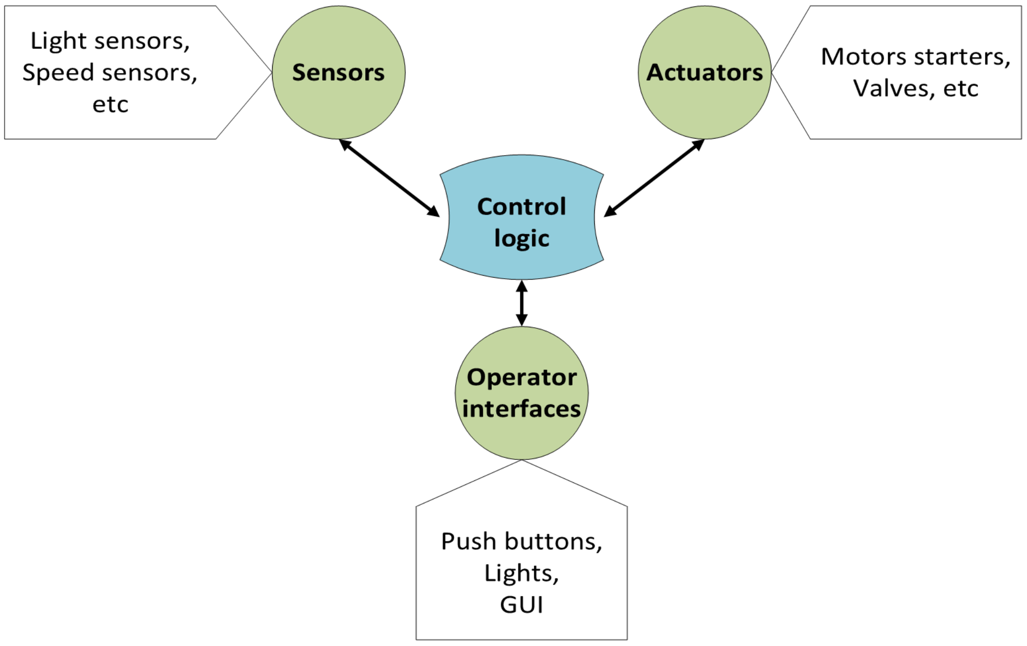


Figure 1.2: A generic industrial control system

Usually, ICS are characterized by a periodic execution where, the most important macro steps inside a time period are:

- Inputs acquisition
- Control algorithm execution
- Outputs actuation

Throughout these steps the ICS can continuously manage complex physic processes, in a periodic fashion as shown in figure 1.3.

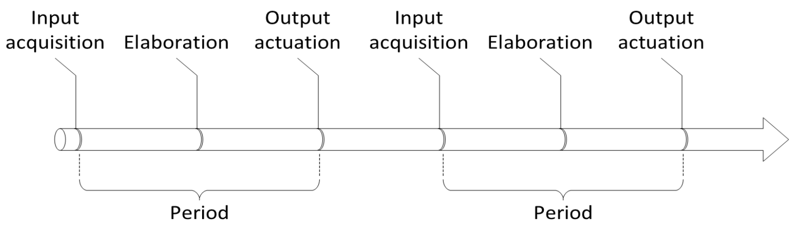


Figure 1.3: Industrial control system time-line

When dealing with this kind of systems, it is important to notice that the so called elaboration time, i.e., the time required for executing the control algorithm, better visible

in figure 1.3, is a crucial element for the correctness of the control functionalities.
In fact, a control system works fine if and only if it produces right results at the right time, i.e. it has to compute the correct results, but even more important it has to respect its deadline.

1.2.2 The history of Industrial Control Systems

When ICS started their computerization process, special devices able to deal with those systems were needed: in particular, for the industrial field PLC have represented the main support since their invention in the late 1970s.

1.2.2.1 Programmable Logic Controller (PLC)

A PLC can be seen as: "an industrial digital computer which has been ruggedized and adapted for the control of manufacturing processes, such as assembly lines, or robotic devices, or any activity that requires high reliability control" [20].
For its characteristics, it can be taken as the perfect example of a hard real-time system since: it has to produce outputs in response to inputs within a well defined amount of time, otherwise unwanted results will occur.
PLC have granted the advent of what was called Third industrial revolution bringing industry's systems to be computer-based, as known today. They have been reliable and flexible devices used in many fields thanks to their capability of being adapted in different use-cases.

Over the years, computer industry started to grow bringing the use of Personal Computers (PCs) also in the industrial field, this revolution affected also PLC that, with the passing of the time, evolved, but in a slower manner. In fact, PLC producers preferred to design their own Application Specific Integrated Circuits (ASICs) using older technologies than newer ones already used for the computer market [2], and as result PLC became very specialized devices in the control field but at the cost of maintaining older architectures.

Although PC evolved faster than PLC, their architectures and the missing of reliable real-time operating systems influenced a lot the choice of do not let them execute the control operations. Therefore, the control domain was considered belonging to PLC due to the fact that reliability is a must for production, where even a small delay, due to wrong behaviours of control systems can affects a lot the production cost.

Just to provide a real example, think about the motivation of this thesis that has been developed in collaboration with AROL Group S.p.A. (a leading reference in designing, manufacturing and distribution of capping equipments) [1]. When a capping equipment stops working properly and for example, a bottle is not closed suitably, production line has to be halted in order for the wrong bottle being removed, then the line has to be restarted, and of course, all these things are really expensive and they should to be avoided as much as possible.

Although PC were not used for controlling purposes, thanks to their good computational capabilities they were heavily used in the industry fields, especially for Human Machine Interface (HMI) purposes and to deal with complex computations.

The need of using also PCs came out because, for example, early PLC relied on push buttons and pilot lights to provide operator interfaces. However, with the passing of the time user control and monitoring interfaces got more complex, requiring the use of newer mechanisms. I.e., PLC required the interaction with people for many purposes, such as: configurations, status and alarm reporting and daily control, hence, just some push button and some Light Emitting Diodes (LEDs) were no longer enough.

To sum up, both PC and PLC have been used within the industrial field, for example: the formers, connected via communication interfaces to the PLC, were used to provide Graphic User Interfaces (GUIs) from which users interacted with the system, while PLC were in charged of dealing with the control algorithms that needed real-time executions.

With the diffusion of distributed architectures and the usage of multiple devices, new communication protocols were introduced to allow the exchange of data between devices, and sometimes, the used communication protocols became real open standards protocols like Modbus, Ethercat, etc. In fact, just to take back the same example, the PLC and the PC in charge of running the HMI needed to communicate each other in order for the user interface to react accordingly with the system status.

Nowadays, the trend goes toward the use of embedded systems that, thanks to their computational power, should be able to merge different functionalities into single devices such as the so called SBC. These new all-in-one devices combine different functionalities, therefore, for example, PLC and HMI services can be run together into a single device, i.e. both the GUI and the control algorithm run on the same embedded system hardware.

1.2.3 Industrial Control Systems with integrated multiple functionalities

The industrial systems evolved over the years expanding their functionalities and creating new architectures, as described in section 1.2.2. Today, almost every device is still evolving both in terms of its intelligence and its interfaces, for example: new simple devices, with networking capabilities, and the main goal of dealing with I/O (reducing the cabling complexity) have been introduced and they are already in use [10].
Although PC connected to separate PLC have been the standard for industrial automation systems, nowadays things are changing and the idea of new machines all-in-one is coming out in almost every field.

Maybe, the most common example is given by a new Smart TV. It combines different functionalities that were offered by many devices during the past, therefore, nowadays just a single device that holds many functionalities can substitute the various components used in the past. In this specific case, the choice of the all-in-one system comes out because it is convenient, uncomplicated, has a small size or simply because individual components are not available any more.
Coming back to the ICS, nowadays, the all-in-one devices have to cope with many features such as:

- HMI including touch-screen display.
- PLC.
- Network interfaces and connectivities.
- I/O of various types.
- Security features.
- Data storing and management.
- Alarm and status notifications.

The most dramatic changes, already tangible, have been done, at the operator interface and the controller level, where functionalities previously offered by different devices have been merged into single machines.
In addiction to what has been done, the new all-in-one devices are going to deal with all the new functionalities required by the concept of industry 4.0, and, of course, the choice

of having a unique device running multiple functionalities has pros and cons that have to be analysed carefully.

Vendors promote these new single board computers devices for many reasons, such as:

- **Price reduction:** a single device is less expensive than multiple devices.
- **Size reduction:** as one device substitutes multiple systems also the space required is reduced.
- **Easiness of use:** since the same device is used for multiple functionalities the same development environment can be used for most features.
- **Reducing of cabling:** less devices need less cables.
- **Reduced time-to-market:** thanks to the easiness of use the time-to-market can be rather reduced.
- **Increasingly functionalities:** the newer devices offer many new features.

As said, even the software becomes easier to develop, in fact, for example: the communication between HMI and the PLC is made simpler by the fact that both software run on the same machine. Supporting this, Sam Schuy, engineering manager at Maple Systems, said "When you have the PLC and the HMI all built into one, you only need one piece of programming software to program both the PLC functionality and the HMI functionality. So from the standpoint of the learning curve, it shortens the amount of time required to learn how to use and program the product."[11]

In table 1.1 a comparison, reporting the main differences between integrated devices and classical separated implementation, is provided:

Feature	Unified device	Separated devices
Physical size	Efficient	Significant space required
Development environment	Excellent integration between HMI and PLC programming	Usually two separate environments are required and the integration requires some effort
Rated for safety applications	Available with some effort	Easily available
Cost	Low	High
Processing capabilities	Medium	High
Typical application	Small/Medium systems	Large systems

Table 1.1: Comparison between unified device and multiple devices implementation

The creation of these new all-in-one devices implies the revaluation of what the system shall do and what it can really do. In fact, although technology evolved and the new systems are much more powerful than they were in the past, ICS have to full-fill their requirements in order to be productive. Hence, within the idea of unified devices there are issues that come out from the integration of multiple functionalities into one device to be taken into account.

Moreover, John Krajewski claims "Most people, when they look at their PLC, they see it as dedicated to safe, efficient, and effective control. That's it. They don't want it doing anything else, and they don't want there to be other functions bringing down that performance, particularly when it relates to safety. If you have a common Central Processing Unit (CPU), and you start doing things in the HMI that start utilizing that CPU so that it's no longer available for control, that can be a bad situation. You don't want people using it to make calls to a database that could have the CPU very busy for a time, when it's also responsible for ensuring that it's going to be doing interlocks that are responsible for safety. That's been the common argument against bringing them together." [11]

To sum up, new all-in-one devices are going to be used, however, since ICS require to full-fill strict constraints when these new systems are used, there is the need of being sure that everything will work fine to avoid unwanted situations and huge incoming losses.

1.3 Mixed criticality systems and heterogeneous devices

The all-in-one devices introduced in section 1.2.3, with its problems, can be easily recognized as belonging to the class of Mixed Criticality systems (MCSs).
In fact, for definition, a MCS is a system made up of hardware and software components that has to execute a given number of applications with different levels of criticality, i.e. the system has to provide several functionalities having distinct urgencies. Very often, MCS are implemented with embedded computing devices in charge of running the specified functionalities, having different criticalities, on the same hardware that has to share computation and/or communication resources [3].
Although the most common examples of MCS can be considered belonging to the functional safety concept, other mixed criticality systems exist and they may need to run functionalities with different urgencies as in the case of ICS. In fact, the criticality level of a functionality is not given just by its safety level but, it can be influenced by many factors, such as: reliability, availability, integrity and security. Therefore, even if the most obvious use of MCS is given by safety-critical systems like aircraft and automotive, also ICS can be considered as belonging to the MCSs and for this reason are treated here.

Considering the case of ICS, companies are trying to keep leading positions by offering newer functionalities and services, like in the case of innovative HMI that use touch-screen displays providing gorgeous GUI. Moreover, the complexity of industrial controlling systems is increasing continuously and systems are equipped with multiple sensors and actuators in order to provide the additional services introduced with the concept of industry 4.0. Along with these extra functionalities and services, the creation of all-in-one devices, integrating the controlling part, is also emerging, especially thanks to the increment in processors power.
Specifically, processors power is also increasing thanks to the so called MPSoC, see section 1.3.1. Hence, today many functionalities can run on the same chip, however, this trend raises the complexity of the MCS since some functionalities have to run respecting strict timing constraints in order to make the systems working fine. Therefore some features have higher priorities than others and things get even more complex due to the usage of MPSoC.

1.3.1 MPSoC

MPSoC are System on chip (SoC) made up of multiple processors and cores. Usually, they target embedded applications thanks to their low cost, high performance, energy efficiency

and low area characteristics.
MPSoC can contain multiple components: processing elements, memories, I/O elements, networking modules, security cores, they all linked to each other via on-chip interconnections, provide the needed performance for multimedia, telecommunications, security and industrial applications [14].
Increasingly MPSoC are also used for real-time applications, i.e. systems whose behaviour depends not only on their functionality but also on their response time (these systems will be better explained in section 3.1.1).
Very often, the running of real-time applications in a MPSoC raises the problem of MCS because the available resources have to be shared in order to execute various tasks with different criticality. Of course, many are the reasons that can cause an increasing in latency due to sharing of resources within a MPSoC, just to summarize some of them:

- **Translation Lookaside Buffer (TLB) misses:** when dealing with systems using Memory Management Unit (MMU) and separation between virtual and physical addresses, TLBs help to quickly convert virtual addresses into physical ones, acting as a sort of caches where most used addresses, already converted, are stored. Therefore, due to the execution of various applications by different cores, TLB misses while executing real-time tasks can cause unexpected latencies.
- **Processor caches:** usually, caches provide high speed buffers between CPU and main memory, but especially on multi-core devices cache sharing can easily cause bigger latencies.
- **Bus contention:** on-chip buses are used to let the different SoC's peripherals communicating: but, for example, when memory transfers happen directly through Direct Memory Access (DMA) peripherals, the channel used to move data is shared with the CPU that depending on the available bandwidth can slow down the execution of some tasks. Therefore, bus contentions are source of non-determinism in MPSoC and can cause latency to increase.

Summarizing, MPSoCs are really powerful devices that allow the integration of different application into a single SoC. However, as already anticipated and as will be better shown, the sharing of resources inside the MPSoC can cause real-time applications to miss their deadlines.

Chapter 2

Embedded systems and Linux

All-in-one devices, such SBC, are going to be used within ICS. These embedded systems increasingly offer functionalities and Operating Systems (OSes) are needed for dealing with that complexity.
In this scenario, the choice of embedded Linux as OS seems to be most affirmed one. Therefore, within this chapter, Linux as embedded operating system will be presented and its main advantages will be analysed with respect to other OS.
Then, the basic structure and the principal components of a Linux embedded system will be described outlining the main difference with Linux for Desktop environments.
Finally, since the creation of a custom Linux distribution can be a hard procedure, the Yocto project, a set of tools used for dealing with that complexity, will be introduced.

2.1 Why Linux?

Once got that ICS are evolving toward the use of embedded devices that integrate multiple functionalities, after selected the hardware platform, the main opened question remains which OS to use.
Over the years, electronic devices increased their performances and functionalities, the most straightforward example of that evolution is given by the computer industry. Of course, along with computers also the embedded world has gone ahead providing more capabilities for the tiny embedded systems.
Increasingly, these embedded systems needed new functionalities, such as: connection to networks, multitasking capabilities and other complex stuffs. For this reason, very often, they required a networking stacks and/or services increasing the complexity of the operating system [7] that had to provide these functionalities.

The overall situation brought to the creation of many OS, each with different characteristics and capabilities. Nowadays, the choice of the right operating system for the embedded world is very wide, even if, the trend goes toward open-source systems.
The main reason of this choice comes from the fact that an open source system offers many advantages with respect to a closed system.
Among the many advantages that an open system offers, the most obvious one is the full control of the system made available when using open source solutions. In fact, especially for devices that need customizations due to their limited resources, an open source OS offers the perfect environment.

Linux, as open-source operating system, is perfect thanks to its characteristics of safety, stability, reliability and portability, and although it was originally meant to run on PC platforms, today it is suitable for embedded systems and its use on these devices is growing day after day.
Thanks to its versatility the Linux kernel can be found basically everywhere, common examples of embedded devices using Linux as OS can be found on: smart TVs, in-vehicle infotainment, networking devices (routers, switches, access points), machines controls, industrial automations, medical instruments and so on. Maybe, the most familiar use of Linux is in mobile devices such as smart-phone and tablet, there, the Linux system offers also the support for touch-screen displays and network module enlarging its supporting of different hardware devices.
The examples in figure 2.1 and figure 2.2 give a proof that Linux is really used in many embedded systems which are basically spread everywhere.
As anticipated, many are the advantages that a Linux system offers [19]:

- **Robust features:** Linux offers a multi-threading and multi-functional operating system. It also provides many features such as: graphics and communication supports, and much more.
- **Scalability:** Linux can run both on large systems, with huge amount of memory, and on embedded devices, with small memories. This flexibility allows to tune the Linux system depending on the platform it has to run. In fact, thanks to the modularity and scalability Linux offers, it is widely adopted in embedded systems.
- **Widespread:** Thanks to the open source community Linux grew over the years creating one of the most used operating system, it is well known by programmers and developers who over the years created a huge number of applications that cover many useful domains.

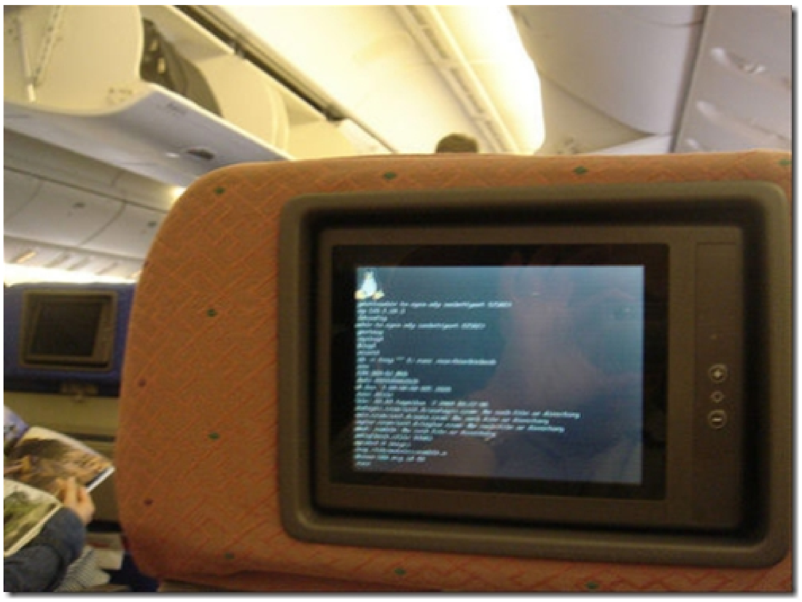
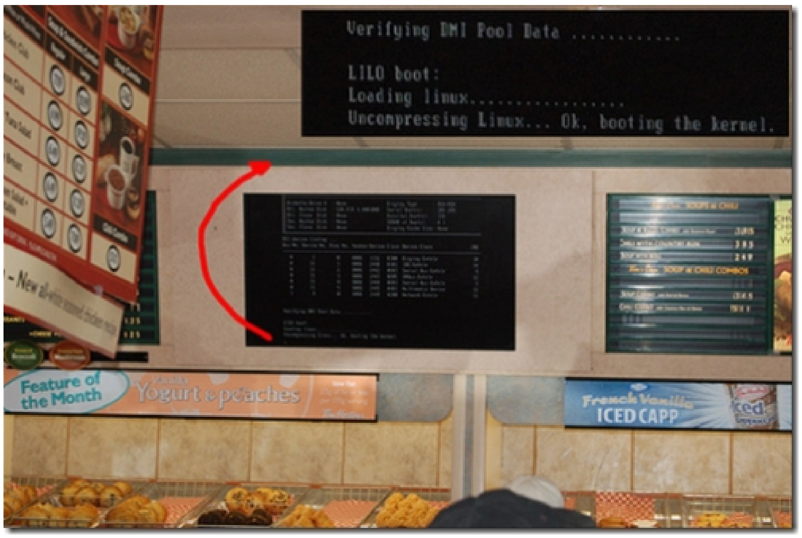


Figure 2.1: Example of Linux usage in airplane [12]



```
Verifying DMI Pool Data ............

LILO boot:
Loading linux................
Uncompressing Linux... Ok, booting the kernel.
```

Figure 2.2: Example of Linux usage [12]

- **Low cost:** Other operating systems are much more expensive than Linux that comes for free almost every time.
- **Easy customization:** Linux is easy to customize and features can be added and removed depending on what developers need.
- **Support:** Linux developers offer support when new issues are discovered. Moreover,

being Linux open source, its documentation is easily available without restrictions.

When dealing with the industry world, the main parameters required by industrial applications are: high performances, high flexibility, low-cost and reduced time-to-market. All these requirements find good support in Linux, that over the years has gained all the benefits given by the open source community as described previously.
Although Linux could seem to be perfect also for ICS, some drawbacks, in using it, have to be taken into account. In fact, especially when the system to be developed requires real-time capabilities, some problems can arise due to the complexity of Linux itself. More details about these problems will be provided in section 3.1.

2.2 The main components of Linux embedded system

As already said, thanks to its characteristics, Linux is one of the most used OS also within the embedded field. In this section, a typical architecture of Linux embedded system with its main components will be introduced and analysed.
Moreover, the main differences of using Linux in embedded systems with respect to use it in Desktop environments will be outlined. Indeed, since usually embedded systems have limited capabilities, dealing with Linux for these small devices is slight different from dealing with Linux for Desktop PC.

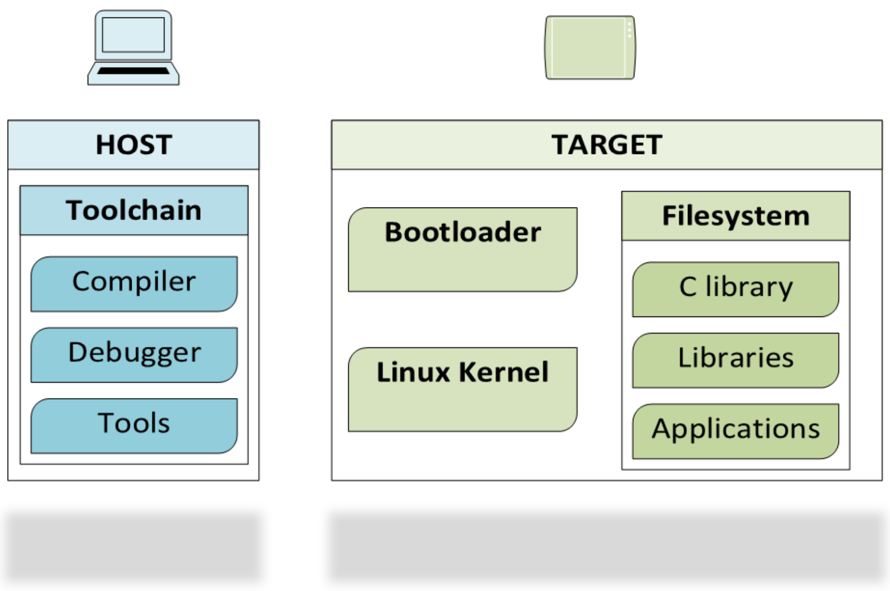


Figure 2.3: The main components of a Linux Embedded system

In figure 2.3, a generic architecture representing a development environment for a Linux embedded system is shown. From there, two main devices can be easily recognized:

- The target: it is the embedded device on which Linux along with the application software will run.
- The host: it is a computing machine where all the development tools reside, usually it has good computational capabilities since it used to cope with the limited performance offered by the embedded target.

Usually, the software tools that run on the development host make up the so called Toolchain that in its turn is made up of different software components such as:

- The compiler
- The debugger
- The binary utilities

While, on the other side, the target device, in the specific case of Linux, is made up of other different software components:

- The boot-loader.
- The Linux kernel.
- The root file system whose main elements are: the c library, system libraries and user applications.

Since all these components are really important to understand how a Linux system works, they will be analysed one by one.

2.2.1 The Toolchain

The Toolchain is the first element that is needed to deal with a Linux embedded system, basically, it is made up of all the tools needed to develop the software that has to be run on the target device.
There are many ways to obtain a working Toolchain, in fact, it can be compiled from sources, or it can be simply downloaded and installed, or as in most of the cases, it can be created using a build management system like the YP, see 2.3.1 for more details.
The most common tools present in an embedded Toolchain are shown in figure 2.4, and since it can be interesting analysing them, a small description for each of these tools will be provided in the following part of this section.

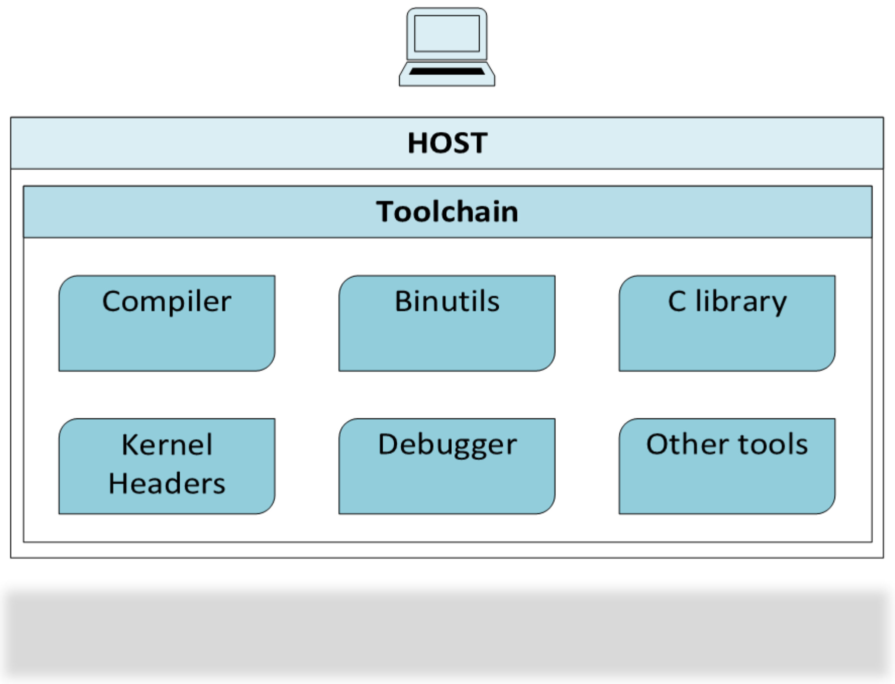


Figure 2.4: The toolchain

2.2.1.1 The compiler

Probably the compiler represents one of the most fundamental part of a Toolchain.

Dealing with Linux, in most of the cases, it will be a version of GCC, since it represents the only practical way for compiling a Linux kernel.

When some code has to be compiled for an embedded system, usually, the used process takes the name of cross compilation because: the compiler runs on an x86 machine, but it is used to compile software that will run on another architecture, such as ARM platforms.

In a few words, the cross compilation can be considered as a straightforward process that runs on a host PC and starting from source files generates instructions that can be understood by the target hardware.

In figure 2.5, the path marked by the continuous line represents the cross-compiling process, while, the dashed lines show some variants of the cross compilation process. In fact, there is also the possibility of running the compiler on the hardware target.

However, this process requires more performances by the embedded hardware and it is almost unused. Moreover, even in this case, very likely the compiler that runs on the embedded hardware has been obtained through a cross compilation process.

Of course, there is also the possibility to compile a program for running it on the development host itself. Indeed, it could be useful to run a given program on the development PC before compile it for the target hardware.

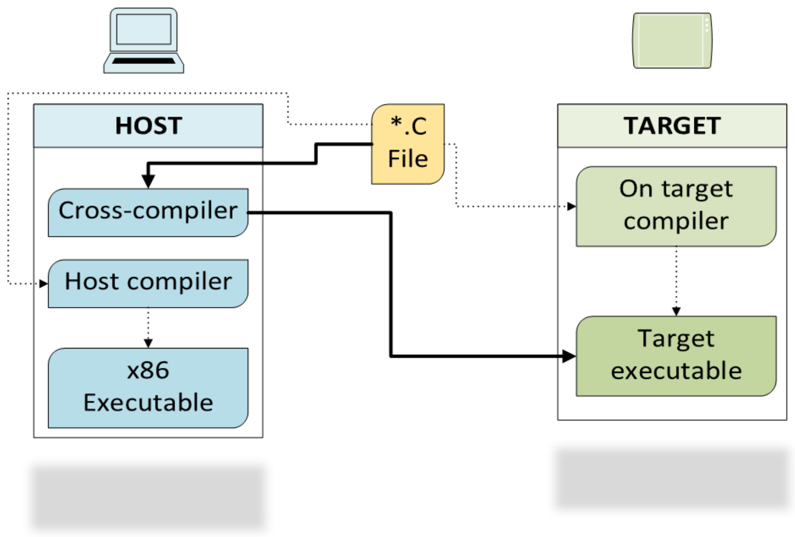


Figure 2.5: The cross compilation process

2.2.1.2 The debugger

Sooner or later, during the development process of an application, debugging tools will be needed. Since in the case of an embedded system it can be difficult to debug, special tools, usually integrated with the Toolchain, are used.
Typically, GDB is the most used tool for debugging. It is made up of two different software components: the first runs on the host, while, the second one is executed on the target hardware and it is usually called GDB server.
These two software components communicate each other, through an Ethernet or a serial connection, allowing the debugging of applications running on the embedded hardware.
In figure 2.6 the general architecture used for debugging embedded systems with GNU Debugger (GDB) is shown.
Similar tools exist also in the case of Linux Kernel debugging. However, in that case it is suggested to use JTAG debuggers.
They offer the best debugging capabilities because, they guarantee direct access to the CPU allowing the inspection of executed instructions and memory areas in a non intrusive manner.
However, if it is not possible to access the JTAG interface the most common tool used

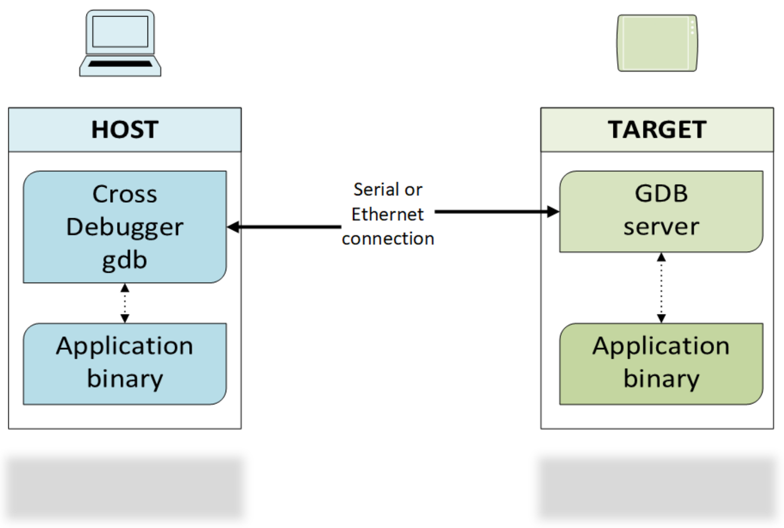


Figure 2.6: The debugging process

for kernel debugging is called KGDB. It allows to debug the Kernel using a serial or a network connection, and it can be simply used enabling the corresponding Kernel feature.

2.2.1.3 The binary utilities

The binary utilities, *Binutils* for short, is a set of tools used for creating and managing binaries and source files such as: object files, libraries and assembly sources.

Binutils is a set of utility programs that have different uses: some of them are used during the compilation process itself, like the linker and the assembler, while other tools are used to manage the binaries.

For example *strip* is used to strip symbols from compiled applications to make them smaller, or, tools like *addr2line* are used to convert an address within an application to the corresponding line of source code.

2.2.1.4 Tracing and profiling tools

One of the greatest thing of Linux is the range of software that it offers, for example, there are tools for tracing and profiling of the Linux kernel processes.

These tools can be used for debugging purposes, but also for investigating strange behaviours and for performance analysis as will be better described in section 3.3.

2.2.1.5 Kernel headers

Another important part within a Toolchain is given by the kernel headers. They are useful because they define the interface between applications and the c library as shown in figure 2.7.

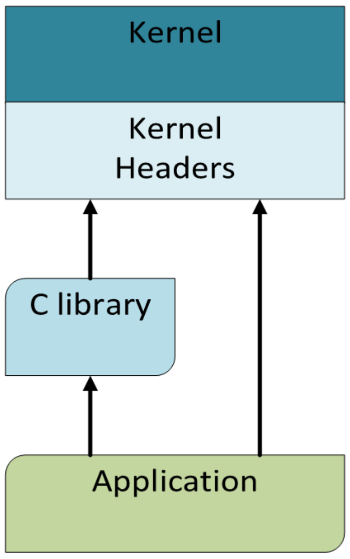


Figure 2.7: Kernel headers

Therefore, the c library and the applications accessing the Kernel Application programming interfaces (APIs) need to be compiled against the Kernel headers in order to use the proper calls' interfaces.
Thus, typically, in an embedded Linux Toolchain it is important that Kernel headers are kept synchronized with the Kernel version running on the embedded hardware.

2.2.1.6 The C library

The C library provides the main interface between applications and the Linux kernel, as shown in figure 2.7.
Definitely, there are many choices available in terms of different C libraries that can be used.
For example, *glibc* is the standard Linux C library used on almost every Linux desktop. It is designed to be fast and to be compliant to the various C standards.
However, *glibc* is not optimized for size, hence, if size is an issue, because maybe the embedded system has restricted flash storage, there are other choices such as *uclibc* which is a lightweight C library designed to be smaller.

Of course, the choice of which c library to use, like everything in an embedded system, has to be traded-off depending on the system requirements and capabilities.

2.2.2 The target embedded system

Up to now, the concept of Toolchain has been introduced and its main components have been presented. To sum up, the Toolchain allows creating and managing all the software that will run on the actual target device.
Once one has these tools, it is important to understand how a Linux embedded system is made up in terms of software components, see figure 2.8.
For this reason, within this section, the main software components that have to run on the Linux target will be presented and analysed

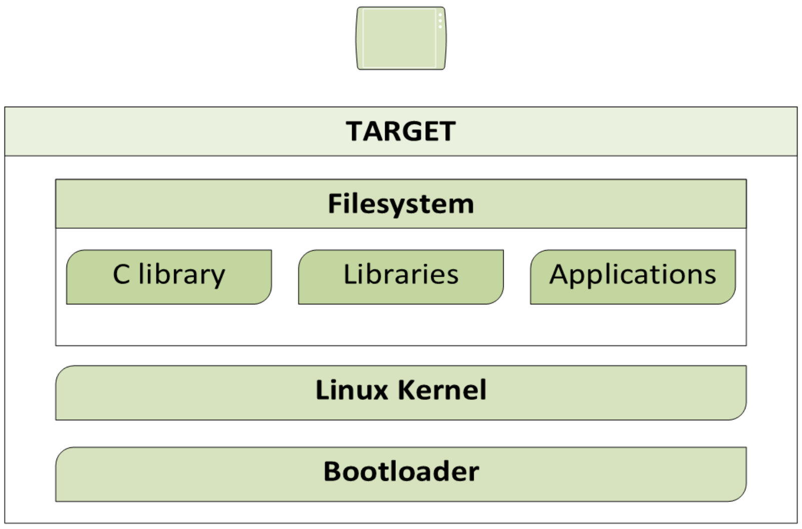


Figure 2.8: The target embedded Linux Components

2.2.2.1 The boot phase and U-boot

When a processor is powered on, it starts fetching instructions at a given address in memory, usually, the code located on that locations takes the name of bootloader.
Basically, in a Linux system, the bootloader is responsible for loading and running the Linux kernel when the system is powered on.
Specifically, when the system is started, the bootloader will do some sort of basic hardware initialization, then it will load the kernel image, from somewhere (SD-card, internal flash, over network), into the main memory, and finally, it will start executing it.

There are many bootloaders available, however, when dealing with embedded systems and Linux, the most used bootloader, that today can be considered the de-facto standard, is named U-boot.
Although it is used to load and run Linux, U-boot is not part of Linux itself, but, it is released with an open source license, and over the years has gained many useful features and functionalities that can be enabled depending on the needs.

U-boot is very configurable and offers support for loading images of the kernel from many storage devices. Moreover, U-boot provides tools for working with memory and other things via a fully interactive shell, indeed, in some sense it is like an operating system.
It can be used to validate new devices since it represents the first step for running the Linux system. Hence, it can be helpful in testing the basic hardware functionalities such as: Universal Serial Bus (USB) connection, network interface, memories, etc.
Moreover, thanks to its versatility some versions of U-boot can fit in small amount of memory offering support even for devices with small storage capabilities.
The main features offered by U-boot are summarized below:

- TCP/IP Stack
- File system managing (ext, fat32, jffs2, ubifs)
- USB Stack
- Serial/network console
- NAND, NOR, EEPROM, SD managing
- High-Definition Multimedia Interface (HDMI) management and Splash Screen
- Command line scripting

Usually, the U-boot software architecture is split in two halves: the first half, written mostly in assembly, runs from the on-chip memory (e.g. the on-chip SRAM or FLASH) and initializes the memory controller.
Then the control is passed to the second half that initializes the other peripherals, and loads the Device Tree Blob along with the Linux kernel, providing also a command line interface from which the user can interact with the bootloader before the kernel is run.
In figure 2.9, a typical boot process involving u-boot as bootloader is shown.

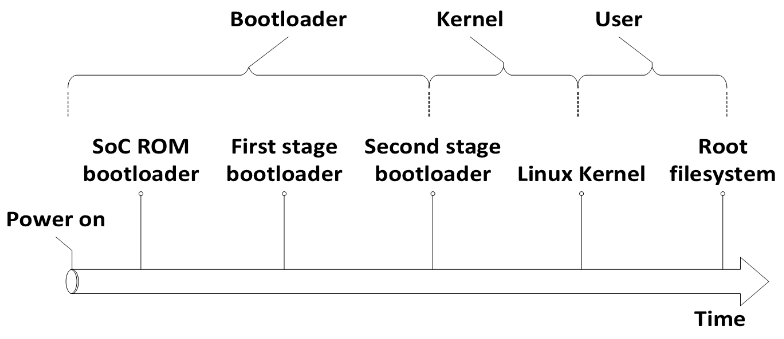


Figure 2.9: Bootloader timeline

2.2.2.2 Device trees

To manage the specific hardware resources, composing the embedded device, the Linux Kernel has to know which peripherals are available, i.e. it has to know information related to the connected and used hardware such as: Input/Output (I/O) devices, Memories, etc.
To let the Linux Kernel get information related to system hardware two main solutions are possible:

- Do not use Device Trees: in this case the hardware information are hard-coded into the Kernel binary code, thus, each modification of the hardware definition implies rebuilding the whole Kernel. With this setting, the bootloader loads the Kernel image and executes it just after preparing some additional information to be passed to the Kernel through CPU registers.

- Use Device Trees: in this case the hardware information is provided to the Kernel, by the bootloader, and specifically, using a binary file called Device Tree Blob. In this way, changes in the hardware definition and configuration do not imply the Kernel recompilation, since just the DTB file can be modified. Of course with this method, time and effort needed for the compilation process can be reduced.

Entering more in details, usually, a DTB file is produced starting from a Device Tree Source (DTS) through a compilation process. The latter, is a tree data structure with nodes that describe physical devices in a system.
In figure 2.10, the boot process both with and without using device trees is shown, and the main memory content can be seen from there.

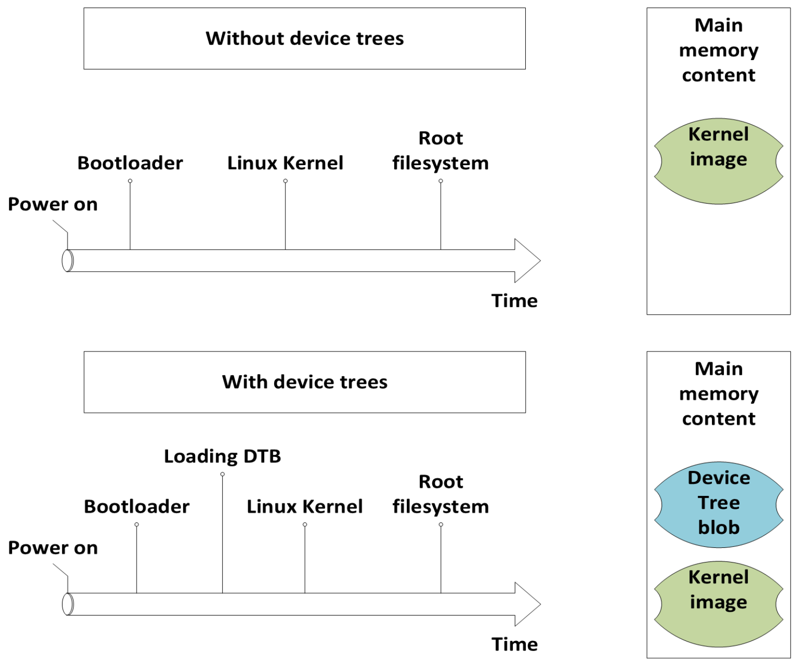


Figure 2.10: Boot process with and without DTB

2.2.2.3 The Linux kernel

The Linux Kernel, as the name suggests, represents the operating system heart, and although sometimes the name is confused, when talking about Linux the actual system is the Linux Kernel itself.
The Linux kernel is responsible for doing all the operations that an operating system has to do. Basically, the main Kernel's goal is to mange the system resources allowing programs to run while they are sharing these hardware resources.
Commonly, the main resources managed by the Linux kernel, but more in general by operating systems, are:

- The CPU resources: being the CPU responsible for running programs, the kernel has to decide which process has to run, when and in which processor has to execute it.
- The memory resources: the Kernel is in charge of managing the memory areas, deciding when to allow new allocation requests and how to manage the available memory.
- The input/output resources: the kernel manages requests from applications that need to perform I/O operations and provides methods for using the various devices.

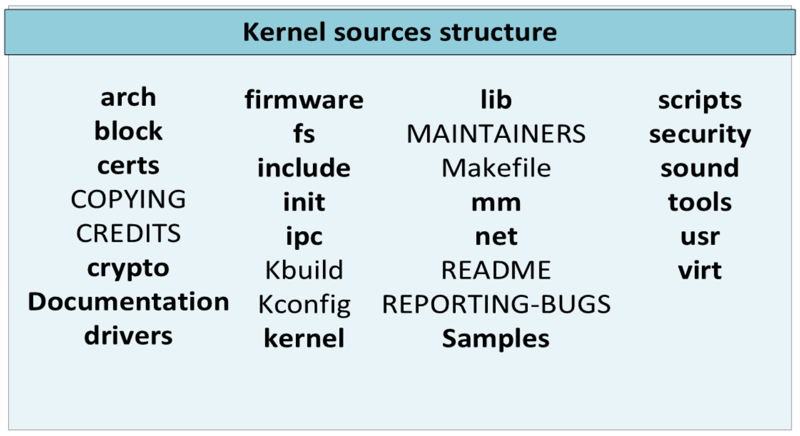
Along with resource managing, the Linux kernel manages all the processes running within the system, providing:

- Mechanisms for tasks to communicate with each other.
- Device drivers.
- API allowing applications to interact with the hardware (such in the case of Ethernet connections or Displays).
- Power management policies.

Thanks to the open-source community, the Linux source code updates frequently, changing its version approximately every three months. Of course, from one version of the Kernel to another the source code changes, but, its layout remains almost the same among the various releases.

The source code

Once downloaded, the Linux Kernel layout, in terms of directories and files, looks like the one shown in figure 2.11.



Kernel sources structure

arch	**firmware**	**lib**	**scripts**
block	**fs**	MAINTAINERS	**security**
certs	**include**	Makefile	**sound**
COPYING	**init**	**mm**	**tools**
CREDITS	**ipc**	**net**	**usr**
crypto	Kbuild	README	**virt**
Documentation	Kconfig	REPORTING-BUGS	
drivers	**kernel**	**Samples**	

Figure 2.11: Kernel sources layout

Most of the kernel's directory names are self-explanatory, for example: fs stands for file system, mm refers to memory management sources, and so on.
One more interesting directory could be the arch one, because it contains all the architecture specific source code used to deal with the different CPU architectures for which Linux provides support. When talking about the Linux Kernel, the architecture specific source code represents about 20 percent of the entire Kernel source code, and this percentage is

not so high considering that therefore, the 80 percent is CPU independent. Along with the arch directory, also the drivers folder, containing lots of different device drivers, contains architecture specific source code.
Within the shown Kernel source layout, another interesting file is the makefile. It allows controlling what has to be built and how it has to be built. In fact, if one wants to start the building of a Linux Kernel, the first step consists in setting all the environment variables that are used by the makefile to compile the code for the desired device and architecture.

User space vs Kernel space

Looking at figure 2.12, that shows a top-level view of a typical Linux system, the most straightforward thing to be noted is the difference between User mode and Kernel mode, in fact, Linux offers the separation between the two modes.

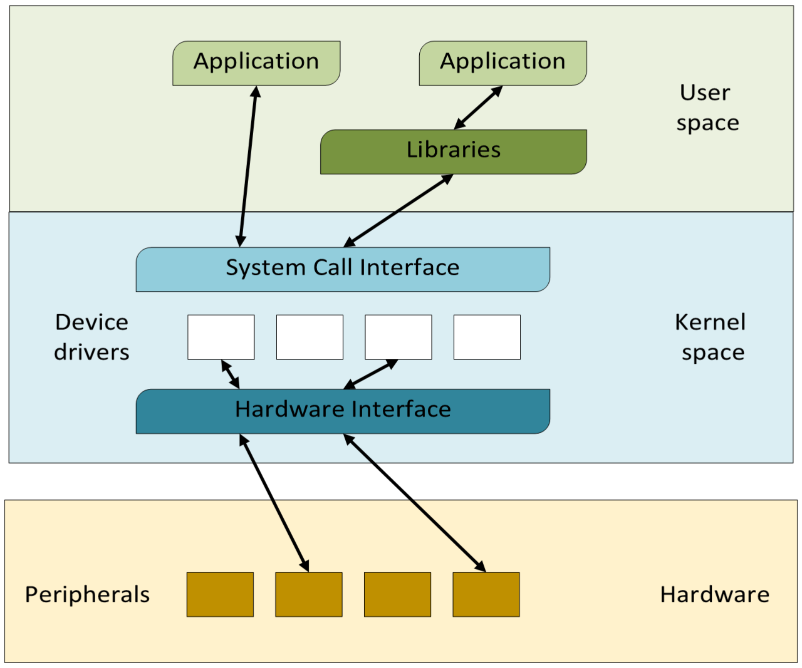


Figure 2.12: Kernel space vs User space

When an application that runs in User mode is given, it has no direct access to any hardware, but, it can only ask to the Kernel, through a c library function (such the open) that will be intercepted by the system call interface, to access a device functionality.

Thus, the code that runs in User mode or User space effectively runs at the lowest level of privilege within the system, and it can only ask for things to be done by the Kernel. Conversely, the code that runs in Kernel space has complete access to the whole system

having the possibility to see and access all the memory and hardware connected to the system.
Of course, the main reason of this separation comes out from the fact that the code running in Kernel space can easily break the system producing unwanted results, while thanks to Kernel and User space separation more reliable systems can be built.

Being the Linux kernel very flexible, it is also possible to configure the Kernel such that even if it is already built, new Kernel parts can be added dynamically. Specifically, the Kernel parts that can be added to running systems are the so called Kernel modules and can be loaded and unloaded dynamically. This mechanism allows high flexibility enabling Kernel to fit in very small amount of memory, that can be loaded quickly. Then the Kernel can add the needed functionalities at run time.
In figure 2.13 the loading and unloading process of a generic Kernel module in Linux is shown.

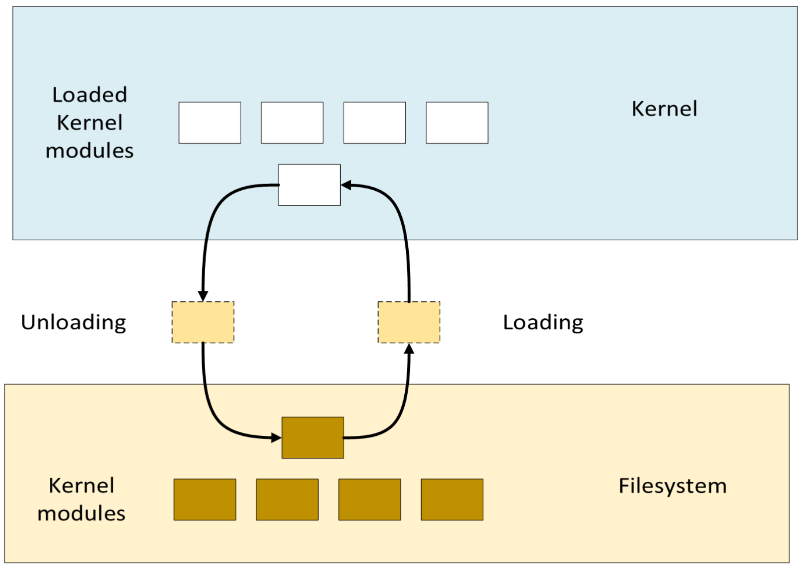


Figure 2.13: Kernel modules loading and unloading

2.2.2.4 The Linux root file system

The final component of a generic Linux embedded system is the so called file system. Usually, it contains many libraries and applications, but it is not easy to define a standard that includes a given number of them because it depends on what the final device has to do and which are their needs.

In order to analyse a generic file system, maybe, it is easy to look at what are the minimum requirements to let the system boot. Then all the wished extra features can be added to that basic system.
Therefore, considering a minimal Linux file system, it should contain:

- An Init application: once the Linux Kernel mounted the file system the first thing it looks for is the Init application. It is the first User space process to be executed and it is the parent or grandparent of all the other User space processes within the system.

- A shell and tools: typically, during the development or during maintenance a shell is needed along with the tools it usually offers. Most of the time to get these functionalities into the system, projects like Busybox are used to provide all the needed tools within the shell environment.

- C library: basically, the c library is fundamental since it will be used by the applications.

- Device files: in Linux systems the main idea for representing hardware devices is to use files. Therefore, the well known */dev* directory is fundamental to let user space applications access devices.

- Virtual file systems: because they do not really exist, virtual file systems are not present on the actual storage device, but they are maintained by the Kernel in memory such that it is possible to interact with them like real directories and real files. Examples of virtual file system are: the *proc* directory which provides process information allowing some management of the tasks in execution, and the *sys* directory that provides information about the system hardware.

In figure 2.14, the layout and a description for each directory of a common file system is shown.

To sum up, most file systems have the previously described components and much more depending on the final application requirements. When the file system is ready, it has to be read by Linux that during the lifespan of a project has to deal with different scenarios. In fact, thanks to its versatility the Linux Kernel allows the mounting of the file system from various locations such as: network, on-board flash storage and removable media(USB key, SD card).

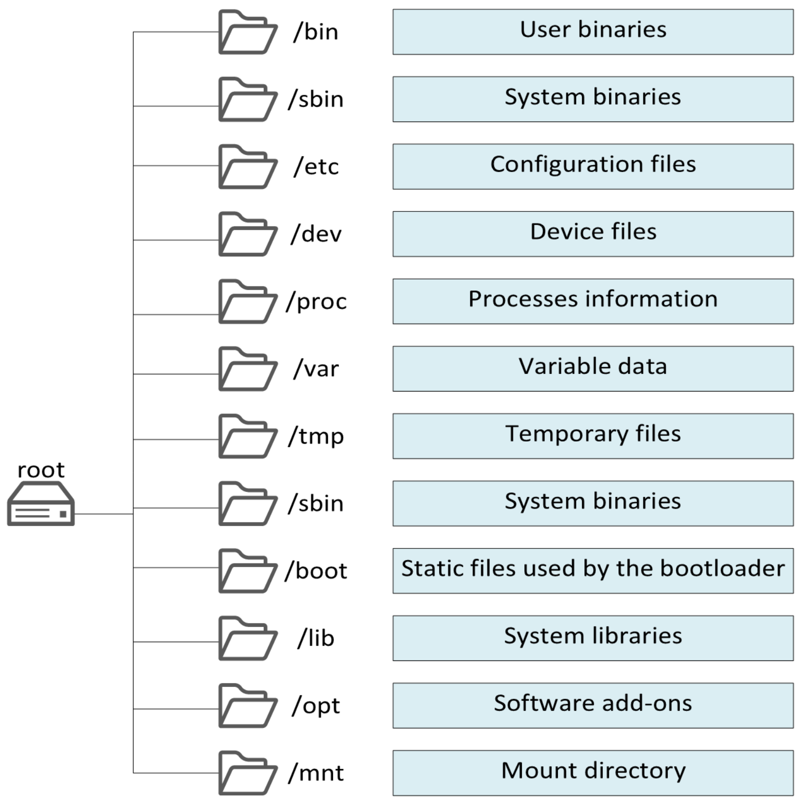


Figure 2.14: Root file system layout

2.3 How to build a custom Linux distribution

Once all the main components of a Linux embedded system have been analysed, it could be interesting to start thinking about from where all these source codes can be downloaded, configured and compiled in order to obtain a working custom Linux distribution.
Thanks to the open-source license, almost the whole Linux system is available in source code and many are the ways to get a Linux distribution, as shown in figure 2.15.

Summarizing the content of figure 2.15: as first option, it is possible to get all the different parts of the Linux system, such as the kernel, the bootloader and all the other components, to construct by hand a Linux distribution. Alternatively, one could decide to go through the route of using a desktop Linux distribution or might follow what the SoC or board vendor tells using commercial components released by software vendors. Or finally, one could use a distribution building system, that as shown, is a method supported also by SoC/Board and software vendors.

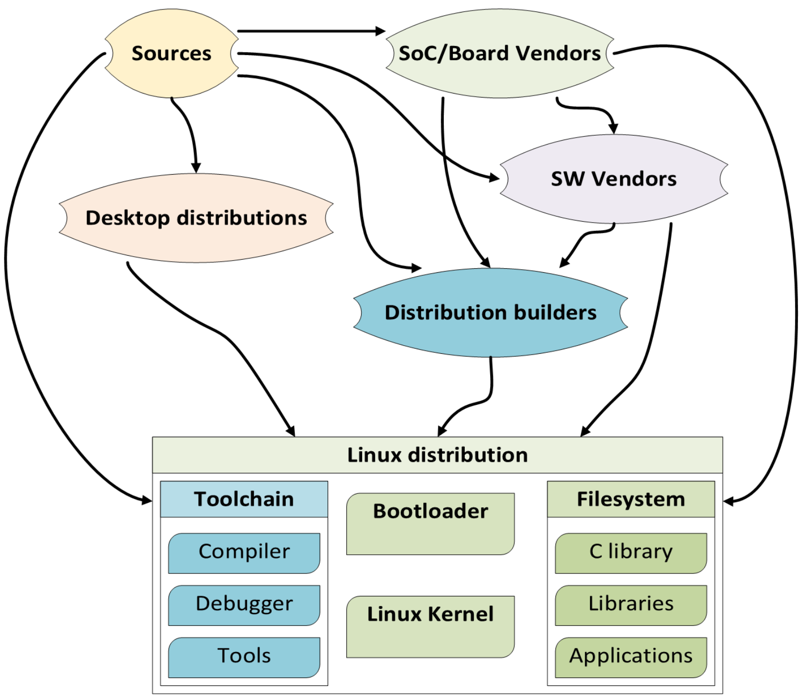


Figure 2.15: How to get a Linux distribution in a nutshell

Looking at the first introduced method, i.e., building a Linux Distribution from sources, it could be very interesting as an academic exercise. Perhaps, kernel and bootloader are the two easiest parts of the system to actually get the source code and to build for the target device. Conversely, the Toolchain along with a working Linux file system, as already said, are not so easy to build from scratch and can represent tricky and time-consuming operations.
The main reason of this complexity comes from the fact that, there are a lot of components with many dependencies to be solved and a lot of configurations to be set.
As an example there is a project called Linux from scratch. Basically, it provides a guide of 334 pages and at the end, the obtained system boots but does not do anything else at demonstration that there is a lot of work involved to obtain a working system starting from scratch.

Analysing the other options, many desktop Linux distributions are available for embedded devices and also SoC/board vendors provide SD card bootable images that contain version of desktop distributions like Ubuntu or Debian.
Of course, this can be an easy and quick way to get started, but the lack of control in the ability of configuring what goes into the file system has to be taken into account since

probably the image will contain a lot of useless software for the actual embedded project. To sum up, Linux allows high flexibility and customization, but, the creation of a custom distribution by hand for a given device can be a tricky operation. On the other hand, using a desktop distribution already built could represent a good alternative, since it could be used quickly and easily, but what is lost following this choice is the long-term control of the Linux system.

Therefore, over the years, many tools have been developed to help developers, during the process of creation of custom Linux distribution. Today, these tools are called distribution builders or distribution management systems, and there are two main alternatives.
The first, that is maybe simpler and easier to get started, is called Buildroot. It supports multiple architectures and C libraries, and basically provides packages that can be put into the final file system. It is based on a set of makefiles that provide lists in which it is possible to select the desired packages to be included into the final system. Because Buildroot is really easy to use, it is not so powerful, and for this reason, it is mainly used when dealing with small or hobby projects.

The alternative to Buildroot, in terms of distribution management systems, is given by the Yocto Project. It is run by the Linux Foundation organization, that essentially represents the industry body for Linux, and its main intention is to provide templates, tools and methods to create custom Linux based systems for embedded products.
Yocto is more complex than Buildroot, but it provides much more flexibility with the ability of adding all the desired software into pre-defined Linux distribution, i.e., with the YP not only Toolchains, bootloaders, kernels, and file systems can be built, but also binary packages, to be installed at runtime into a Linux distribution, can be created and managed.
Therefore, it provides the long term support that is needed for real Linux products in commercial environments, moreover, being widely used, most of the SoC/board vendors provide support in Yocto for their devices.

Summarizing, Yocto is designed to be used in much more complex projects and it will be introduced in a more detailed way in the following of this section.

2.3.1 The Yocto project

As already introduced, the YP is an open source project that helps developers during the creation of custom Linux-based systems for embedded products. It comes out from the

collaboration between the open source community and industry companies allowing the creation of customized distribution in a hardware agnostic way, i.e. developers do not need to take care of the hardware platform they want to use.
Yocto provides all the needed tools to enable the creation of custom Linux distributions for embedded devices, and as affirmed by the community: "The Yocto Project is not an embedded Linux Distribution, It creates a custom one for you" [22].

The Yocto Project focuses on improving the software development process and helps developers in building custom Linux OS for about any kind of computing device, in fact, thanks to the partnership with industries Yocto allows to create Linux images for all the major hardware architectures. Moreover, it makes the creation of custom Linux distribution faster, easier and cheaper.
Yocto allows to create a custom distribution enabling a high degree of flexibility and customization which is a crucial aspect when developing for embedded products where the control of which software to include represents a key aspect.

"The project provides a flexible set of tools and a space where embedded developers worldwide can share technologies, software stacks, configurations, and best practices that can be used to create tailored Linux images for embedded and IoT devices" [22].
Basically, the YP is mostly a group of recipes, written in python and shell scripts, that are managed by a task scheduler called Bitbake. The latter is responsible for producing what has been configured into the various recipes.
Being the Yocto project evolved over the years, it houses multiple projects such as:

- **OpenEmbedded core:** it provides metadata to create binary packages to be installed on the target even at runtime.
- **Bitbake:** it is the tasks scheduler that actually builds the images.
- **Poky:** it represents the reference distribution.
- **Devtool:** it is a command-line tool that helps in build, test and package software.
- **OPKG:** it is a lightweight package management system.
- **CROPS:** it is a cross-platform development framework.
- **Autobuilder:** it is a project that automates build tests and quality assurance.

To sum up, the Yocto project is made up of many other sub-projects and provides a base that can be also extended using new meta layers. It is well-supported also by many SoC vendors, and just for curiosity the name Yocto came out from the SI prefix for 10^{-24} to indicate that it can build very small Linux systems [17].

2.3.1.1 Components and tools

Once got that the YP is made up of different projects and systems, in this section, the contributions given by the main components will be analysed outlining the structure used by Yocto.

OpenEmbedded core

As already stressed, usually, a Linux system is made up of different components, and in order to automate the building of each Linux component three main steps can be identified:

- The first phase to do consists in: fetch, unpack and patch the components source code.
- Then, the source code has to be: configured, compiled and packaged.
- Finally, the produced packages have to be bundled together on a root file system.

The Yocto Project to deal with Linux components uses the OpenEmbedded build system that brakes down these different steps into distinct actions called tasks.
Specifically, there are tasks to be done at each different level of the build process, following specific configuration options, as shown in figure 2.16.

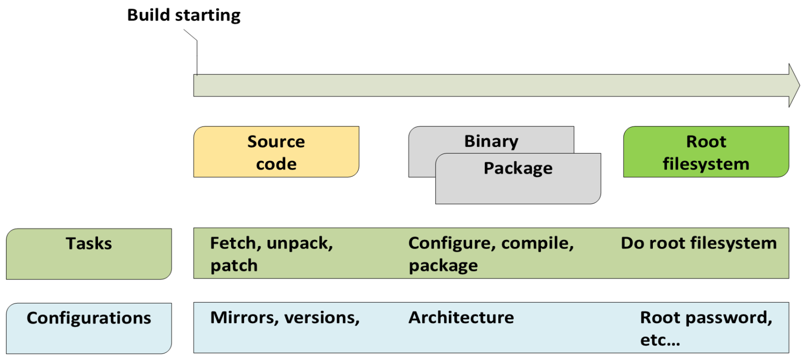


Figure 2.16: The OpenEmbedded build system

Therefore, within the OpenEmbedded project there are defined: tasks to fetch, unpack and patch the software, tasks to configure and build binaries, tasks to create packages, and tasks to produce the final file system.
The configuration options, used to tune the tasks, are generally expressed as a set of variables that can tell to the building system information like: the address of a mirror, the type of architecture to build for, and so forth.
The tasks along with the configuration options make the so called metadata, and dealing with open embedded, three types of metadata can be identified: the recipes, the classes and the configuration options.
The different types of metadata will be analyzed in detail after introducing Bitbake because it can be useful, before going into details, understanding how metadata are used.
To sum up, OpenEmbedded (OE-Core) is set of metadata layers, originally used by the OpenEmbedded system, also adopted by the Yocto Project.

Bitbake

Once the basic concept of metadata has been introduced, another Yocto main component is Bitbake. Basically, it starts parsing metadata, then resolves the dependencies and finally schedules the tasks depending on what has to be serialized or paralysed due to inter-dependencies.
Therefore, Bitbake can be considered as a task execution engine, that as shown in figure 2.17, takes as input metadata (i.e. recipes, classes and configuration files), then analyses the dependencies and finally runs the different tasks in a given order.

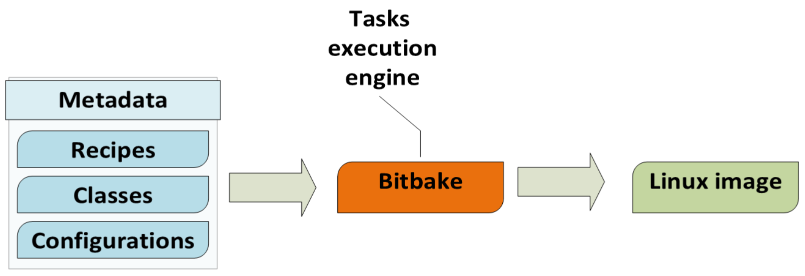


Figure 2.17: Bitbake in a nutshell

2.3.1.2 Main concepts

Recipes

One of the main components of metadata are recipes, they can be seen as a list of tasks to be executed in a given fashion.
For example, as shown in figure 2.18, a recipe used to create a package will contain a given number of tasks: some of them will be involved in getting the source code, others in patching and building the binary, and finally, others will create the package.

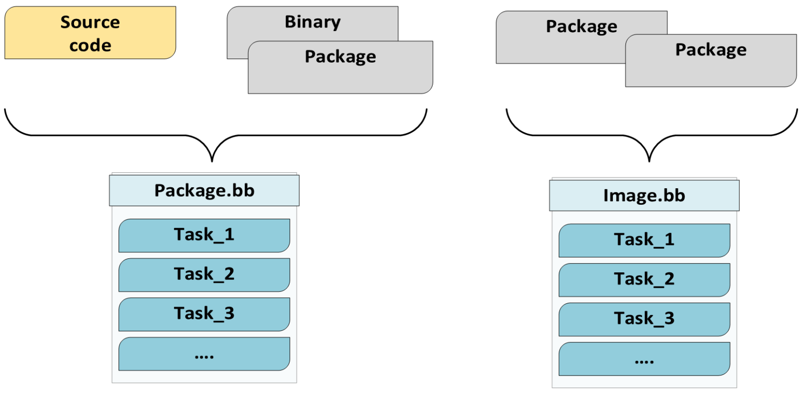


Figure 2.18: The Yocto recipes

Once the recipe is ready, it can be invoked by Bitbake which will resolve all the dependencies and will execute the defined tasks bringing to the final wanted package.
Along with recipes for creating packages, also for images it is possible to create recipes that starting from binary packages generate the root file system following a list of well defined operations.
In order to execute a recipe, the most straightforward command, once initialized the Yocto environment, consists in executing:

```
bitbake recipe-name
```

Listing 2.1: Bitbake command example

Once the command is executed, Bitbake will look for a recipe file named *recipe-name* with the *.bb* extension, then, after parsing the recipe, it will execute the tasks defined by that recipe.

```
SRC_URI = "git://github.com/example.git"
inherit autotools
```

Listing 2.2: Example recipe content

In 2.2 the content of a basic recipe (example.bb) is shown. It is able to download source code from the specified git repository, compile it, and create a package to be installed into a root file system. In that specific case, once the *bitbake example* command is run, the following tasks will be executed:

- **do_fetch:** downloads the source code
- **do_unpack:** extracts the source code
- **do_patch:** patches the sources
- **do_configure:** configures the source code
- **do_compile:** compiles the sources
- **do_install:** installs the binary output
- **do_package:** splits the binaries from documentation, debug symbols, etc
- **do_package_write:** creates the package (RPM, DEB, etc)

At this point, having seen the recipe content, the main question that may arise is: where is the definition of all those tasks?
Looking carefully at the recipe content, the line *inherit autotools* represents the answer to the previous question. In fact, recipes can inherit from the so called base classes which contain a given number of predefined tasks to be executed.

Of course, not always the base class task definitions can be fine for the specific use case: for example when the do_compile task is run, it looks for a makefile but what if there is no makefile within the downloaded sources?
The YP provides an answer also for this, indeed, the recipes can override the default tasks defined into the base classes, in this way, it is possible to adapt a predefined task to the use case requirements.

Classes

As already introduced, usually, recipes use base classes from which they inherit the basic tasks definitions. Within the Yocto Project there are many different base classes used to create recipes for various purposes. Classes use the *.bbclass* file extension and common examples of them are:

- **autotools.bbclass:** used to build a project based on autotools

- **image.bbclass:** used to build a root file system image
- **module.bbclass:** used to compile kernel modules
- **kernel.bbclass** used to build a Kernel

Configuration files

Bitbake, along with recipes, also parses the so called configuration files. They have the *.conf* extension and define configuration variables that control the project's build processes. These files cover different configuration areas such as: machine configuration options, distribution configuration options, compiler options and other general options.
Some of the main configuration files will be presented in section 2.3.2.

Layers

Usually, metadata are organized into layers, they allow to isolate different types of customizations into sort of containers, i.e., they allow to organize the metadata in a modular way making easy future changes and expansions.
The use of a layered approach gives many advantages: for example, thanks to the layered architecture it is possible to easily deal with the customizations required by different machines. In fact, the customization requested to cope with different devices typically reside into special layers that take the name of Board Support Package (BSP) layers.
Therefore, using this approach it is possible to isolate recipes and metadata that support different things, and for example, it is easy to have a separation between the GUI and the BSP metadata.

Append files

Recipes are contained into layers, that for example can be released by board vendors in form of BSP layer, but what happen if one wants to modify an existing recipe?
Also this use case is well known within the Yocto environment and for this reason, it is possible to create the so called append files. Therefore, information inside append files easily extends or overrides the content of recipe files.
Append files use the *.bbappend* extension and have to follow the same file naming of the corresponding recipe they want to extent.

2.3.2 How to start working with Yocto

In figure 2.19 is shown the Yocto Project with its main components in order to summarize how the build system is made up and its working principles.

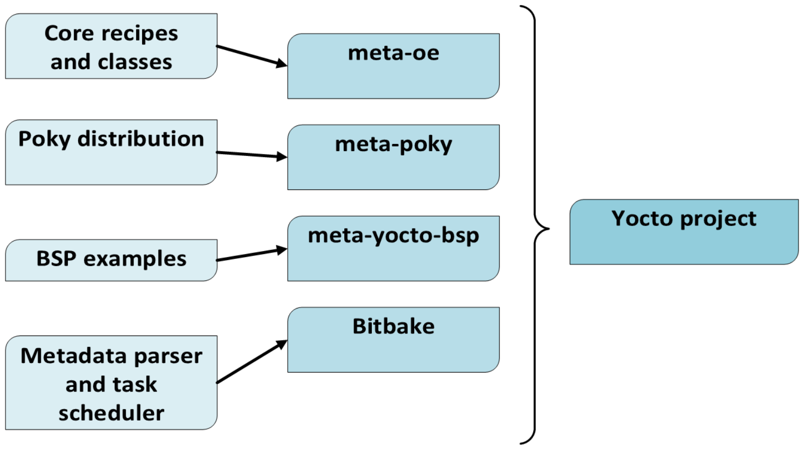


Figure 2.19: Yocto main components

This section will introduce how to download and start working with the Yocto platform from a practical point of view. To get started, the first thing to do, consists in downloading the platform source code from its git repository.
This can be easily done from a Linux shell executing:

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

Listing 2.3: Git clone of the Yocto Project

The build system directories

By default, Yocto comes with few BSP layers and the source code directory has the structure shown in figure 2.20.
Since just a few layers are integrated into the downloaded sources, it is very likely that the specific board BSP layer has to be downloaded and added manually, as well as all the other wished layers.

Environment setup

As preliminary step, to start working with Yocto the *oe-init-build-env* script has to be run. Then, the build directory will be automatically created and the directories layout

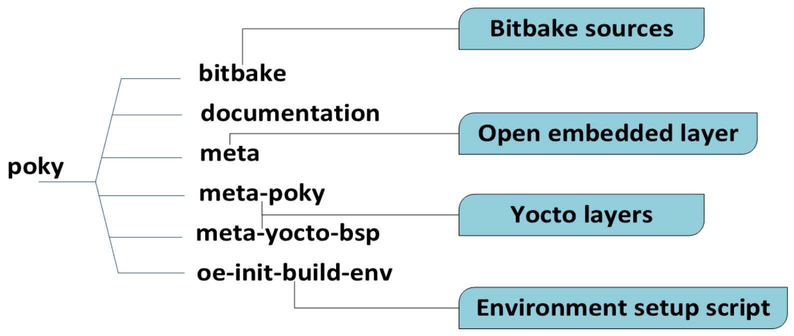


Figure 2.20: Yocto directories layout before running the initialization script

will become something like the one shown in figure 2.21.

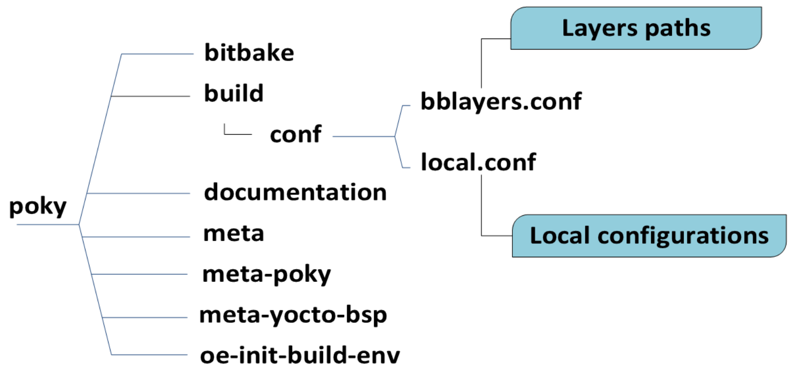


Figure 2.21: Yocto directories layout after running the initialization script

Configuration files

Once the *oe-init-build-env* script is run for the first time, the *conf* directory will be populated by the two configuration files *bblayers.conf* and *local.conf*.
Entering more in details, the first file to analyse is the *bblayers.conf*. It contains all the layers paths to be used by the build system. Therefore, each time a new useful layer is downloaded, it has to be added into the *bblayers.conf* file in order to be recognized by Bitbake. In figure 2.22, the working principle of the *bblayers.conf* file, along with its the default content is shown.
Many BSPs and other Yocto layers can be found at
http://layers.openembedded.org/layerindex/branch/master/layers/.
On the other hand, the *local.conf* file contains all the local configurations for a given build, e.g. inside the *local.conf* file the target machine to be used is defined, along with all the paths, where downloading, staging and compiling sources. The *local.conf* file allows also

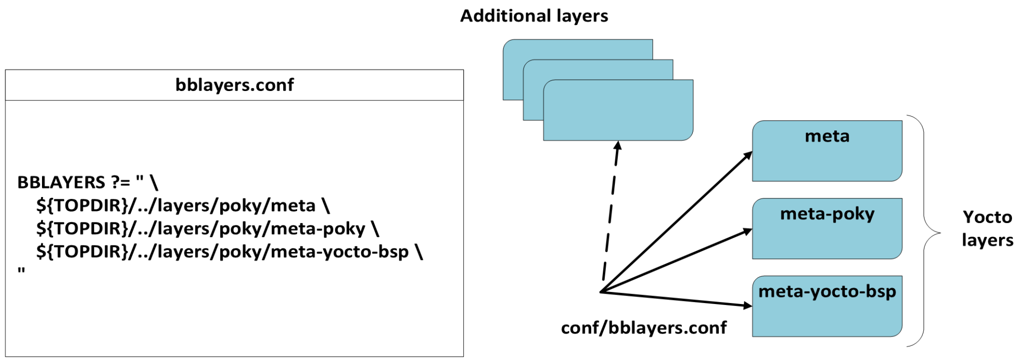

Additional layers
bblayers.conf
BBLAYERS ?= " \
${TOPDIR}/../layers/poky/meta \
${TOPDIR}/../layers/poky/meta-poky \
${TOPDIR}/../layers/poky/meta-yocto-bsp \
"
meta
meta-poky
meta-yocto-bsp
Yocto layers
conf/bblayers.conf

Figure 2.22: *bblayers.conf* file

to customize the image built by the system offering the possibility of: adding packages, selecting the wished kernel version, modifying the packages management system and so on.

Image building

After initializing the Yocto environment, downloading the BSP layer and setting the right machine into the *local.conf* file, it is possible to start building a Linux image simply executing:

```
bitbake image-name
```

Listing 2.4: Bitbake command example for image building

Of course many are the possible images, offered by OpenEmbedded, that can be built. The most common default images are:

- **core-image-minimal:** it is a small console-based system useful for testing purposes as it represents the basic for custom images.
- **core-full-cmdline:** it is a console-based image with a full-featured Linux system.
- **core-image-weston:** it is a basic image based on Wayland with a terminal.
- **core-image-x11:** it is a basic image based on x11 with a terminal.

Obviously, also custom images can be defined creating new recipes that integrate all the desired features and functionalities.
At the end of the compilation process, by default, the image that can be easily deployed into the target device is found in *tmp/deploy/images...*.

Although everything is done by just executing a few commands, behind the scenes different things happen as shown in figure 2.23, where all the main macro steps, from the downloading of the Yocto Project to the obtaining of final image, are summarized.

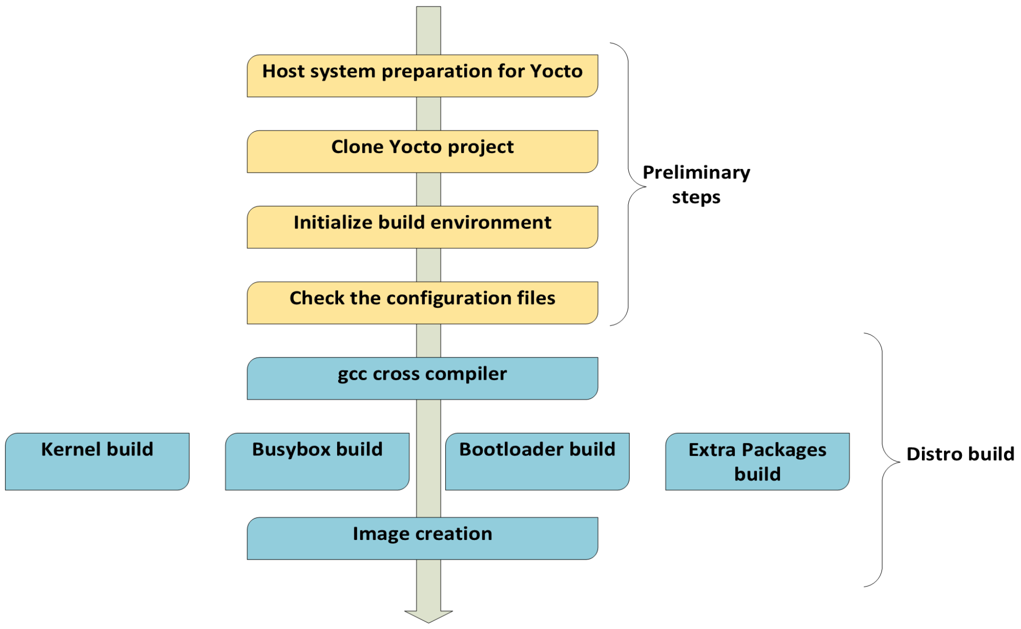


Figure 2.23: Build process steps

SDK creation

Along with having a working Linux distribution image, Yocto allows to generate the corresponding Software Development Kit (SDK) in order to develop additional software for the embedded target device. To start the SDK creation process the command to be executed is:

```
bitbake image-name1 -c populate_sdk
```

Listing 2.5: Bitbake command for SDK creation

Note that the -c option is used to execute a specified task defined inside a recipe.
Once the process ends, a working Toolchain installer, that contains the root file system of the target, is created. In this way, it is also possible to install the Toolchain into other machines.

Creating a new layer and a new recipe

For creating new layers and new recipes, the Yocto Project provides tools aiming at reducing the required effort.
Specifically, to create a new layer containing an example recipe, after having initialized the Yocto environment, some scripts available as part of the Open embedded tools can be used to easily get into the creation of custom layers.
Basically, layers are made up of directories, recipes and configuration files, hence, it is possible to create these directories and meta-data files by hand. However, it is easier to use the script provided in order to create the required structure containing the customized configurations and recipes.
More in details, the *poky/scripts/yocto-layer* script can be efficiently used as reported in 2.6.

```
./poky/scripts/yocto-layer create NEW_LAYER_NAME
```

Listing 2.6: New layer script command

After the running of the previous command, one will be asked to specify a layer priority. The default value will be 6, but depending on the wanted layer's functionalities higher or lower priorities can be chosen. Once a priority value has been chosen, following the script outputs, it is possible to create also an example recipe to be then customized as desired.

Layers priorities

It could be interesting to better understand how the layers priorities work: each layer has a defined priority, which is used by Bitbake to decide which layer comes first when there are recipe files with the same name in multiple layers.
More in details, a higher numeric value represents a higher priority. For example the poky and open embedded layers have a lower priority than those used for the BSP and SDK layers, in fact, the latters sit on top of the formers, following the general layout shown in figure 2.24.
The layer priority plays a key role also when more than one append files for a given recipe exist, in this case, they will be appended in reverse priority, i.e., the bbappend inside the higher priority layer will be added later.

2.3.2.1 The Yocto build system in a picture

In figure 2.25, the Yocto Project is summarized: starting from the top-left, the source code is fetched, then it is patched and configured before being compiled, then a package for the corresponding application can be generated using different formats like rpm, deb,

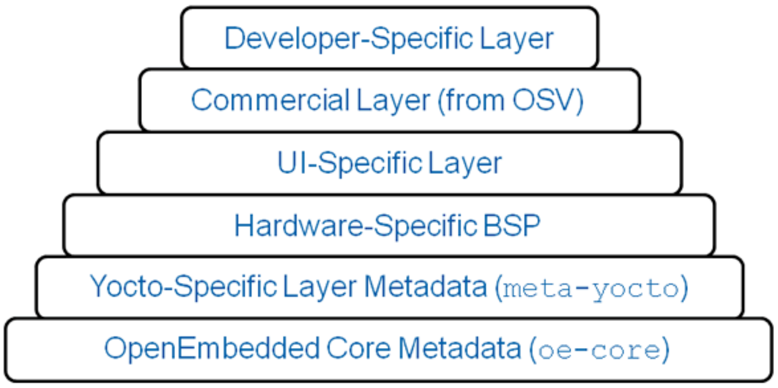

Developer-Specific Layer
Commercial Layer (from OSV)
UI-Specific Layer
Hardware-Specific BSP
Yocto-Specific Layer Metadata (meta-yocto)
OpenEmbedded Core Metadata (oe-core)

Figure 2.24: Yocto layers layout [6]

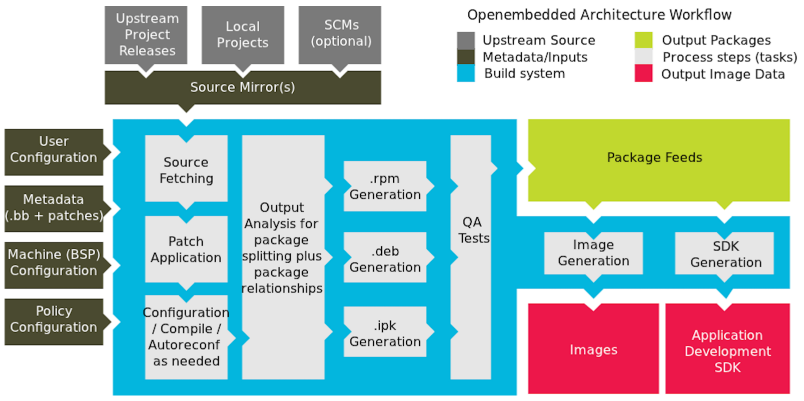


Figure 2.25: The Yocto build system in a picture [21]

etc. Finally, the various packages can be merged to create an image, as shown on the bottom-right.

Chapter 3

Embedded Linux and Real-time

Very often, the interaction between computer systems and the environment happens in real-time, therefore, the discussion about real-time operating systems represents an important topic also for the embedded devices that are part of computer systems.
In this chapter, a discussion about the characteristics of real-time systems will be presented introducing the main ways to achieve real-time behaviour on Linux embedded devices. Specifically, the analysis will put major effort on the so called PREEMPT_RT kernel patch, that can be easily applied to mainline kernels.
Then, the concept of scheduling latency along with the main sources that can bring to increasing latencies will be analysed.
Finally, the closing part will focus on how latencies can be measured and analysed targeting the usage of specific tools and methodologies, such the one offered by Cyclictest, and the more complex ones involving profiling tools.

3.1 Linux and Real-time

Real-time in Linux is a very interesting topic that comes out every time a device, using Linux as operating system, needs to run applications with real-time requirements.
As already stressed, Linux has almost all the required characteristics for being adopted in many systems and also in ICS. However, as anticipated at the end of section 2.1, it suffers when the applications it has to run require real-time behaviours.
The main reason of this problem comes from the fact that Linux was originally meant as a general purpose operating system. Therefore, Linux was not designed to be time deterministic, and as it is, it is not a real-time operating system.
Analysing the way Linux was designed, it was intended to provide the highest level of overall performance. Implicitly this means that things will not happen in a deterministic

time fashion because, everything will be done in such a way to increase the overall level of performance, and, as may be known, these two things go in opposite directions.
Nevertheless, considering that Linux provides many useful features and suffers only this problem, when it has to be used in time critical applications, over the years there have been many attempts to make it working real-time.
Before starting analysing Linux in the real-time context, it could be useful to give a definition of what real-time means.

3.1.1 What does real-time mean?

What does real-time mean is a quite confused concept, and often many contradictory definitions are given by people asked for the definition of a real-time system. Therefore, before going on, it is important to clarify the concept of real-time system.
Some of the most confused definitions involve the following claims:

- Real-time is about fast execution
- Real-time is about high performance
- Real-time is about fast responsiveness

Analysing the previous statements, all of them have some truth but something is still missing.
In fact, when talking about a real-time system: it is not about the fastest execution and the quickest responsiveness time, neither the best performance but it is about timing guarantee. In other words, the definition of a real-time task does not require its execution as fast as possible, but rather as fast as specified by the timing requirements.
A task can be defined real-time, if it has to complete its job before a certain point in time, known as its deadline.
Therefore, when dealing with real-time systems, it is not important to finish before the specified time period, but it is important to guarantee that the task deadline is always met.

Summarizing, when talking about real-time systems, the correctness of the algorithm executed by a task, on a given machine, is not just based on the correctness of the algorithm results themselves, but it is also based on its execution time. This means that if the algorithm is not executed within the specified time slot, the corresponding timing violation will lead to error conditions independently from the algorithm result.
Of course, not all the error conditions are the same. Hence, depending on the consequences caused by the missing of a deadline, systems can be categorized in:

- **Hard real-time:** a deadline missing causes the total system failure.
- **Firm real-time:** infrequent deadline missing can be tolerated, but the service quality may decrease because results produced after the deadline are totally useless.
- **Soft real-time:** the produced results degrade after the deadline is missed, but they are still usable. However, the system quality can degrade after that multiple deadlines are missed.

3.1.2 Make Linux working real-time

Before analysing how it is possible to make Linux working real-time, the question that may arise is: who really needs real-time Linux?
Of course, many are the usages of Linux real-time, and they span in multiple fields, such as:

- Industry
- Automation
- Automotive
- Multimedia systems
- Aerospace
- Financial services

Among the different usages of Linux real-time, to be noticed is the one in financial services. This expresses the need of having real-time capabilities not only on embedded devices but also in more powerful computer systems.
Being the problem of making Linux working real-time well known, over the years, different solutions have been developed.
Traditionally, two main approaches have been used to make Linux real-time capable: the first ones were based on the so called dual-kernel approaches, that were not about making the actual Linux kernel real-time as will be explained later, and the in-kernel approaches which represent the newer trend [18].

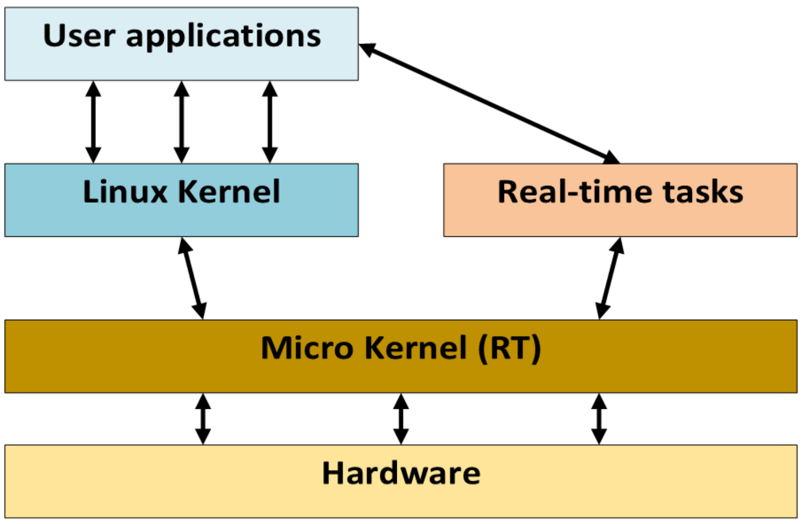


Figure 3.1: Dual Kernel approach

The dual-kernel approaches

The most known dual-kernel approaches, for making Linux real-time, are RTAI and Xenomai: they use a scheme like the one shown in figure 3.1.
In this case, the actual Linux kernel is run over a so called Microkernel that ensures the real-time tasks schedulability preempting the whole Linux kernel every time it is needed. “With dual-kernel, Linux can get some runtime when priority real-time applications are not running on the Microkernel,” said Altenberg [16].

The most evident problem of this approach comes from the fact that someone has to maintain the Microkernel and support its porting on new hardware platforms. Moreover, since Linux does not run directly on the hardware, there is also the need of a hardware abstraction layer to be defined and maintained.
Of course, these problems require huge effort also because the development communities are not so big as the ones that maintain the common Linux Kernel. Therefore, considering the required effort and the various things to maintain, usually, these dual-kernel approaches are some steps behind the actual mainline Linux release.

Entering more in details, RTAI was the first attempt, that was developed by the University of Milano. Using it, real-time applications are written in kernel space, and the interaction between the real-time application and the user space is done through very limited and specific methods. RTAI was meant to obtain the lowest latencies and it is supported by:

x86, x86_64, and a couple of ARM platforms.
"With RTAI, you write a Kernel module that is scheduled by a Microkernel. It's like Kernel development, really hard to get into it and hard to debug."[16]
Moreover, since with RTAI the development is done in Kernel space, due to the Kernel General Public License (GPL), the code has to be released and this can further complicate things because quite often industrial customers want to use closed sources.

The other dual-kernel alternative to RTAI, that nowadays has became dominant, is Xenomai. It supports more hardware platforms than RTAI, but even more important, it offers a sort of solution for doing real-time in user space.
"To do this, they implemented the concept of skins, an emulation layer for the APIs of different Real Time Operatin System (RTOS)s, such as Portable Operating System Interface for Unix (POSIX). This lets you reuse a subset of existing code from some RTOSs."[16]
However, even with Xenomai, it is needed to maintain a separate Microkernel and a hardware abstraction layer. Moreover, standard C libraries are not usable and special tools are needed for applications development.

To sum up, the dual-kernel approaches overcome the limitations of Linux in managing real-time applications. However, both the solutions analysed have some drawbacks that cannot be neglected, especially when dealing with big and complex projects.

The in-kernel approach

Since working with dual-kernel approaches can be complex and this could deter developers to use Linux, the best thing to do could be making the actual Linux Kernel real-time without the help of any external Microkernel, as shown in figure 3.2.
At this point, the question that may arise could be about:

How to get deterministic timing behaviour in a standard Linux kernel?

The first thing that a real-time operating system has to provide, in order to have deterministic timing behaviour, is preemption.
Specifically, it needs to be preemptable most of the time, because higher priority tasks must always preempt other less important things and lower priority tasks. Of course, once added preemption, other well known problems, such the priority inversion, have to be managed and solved.

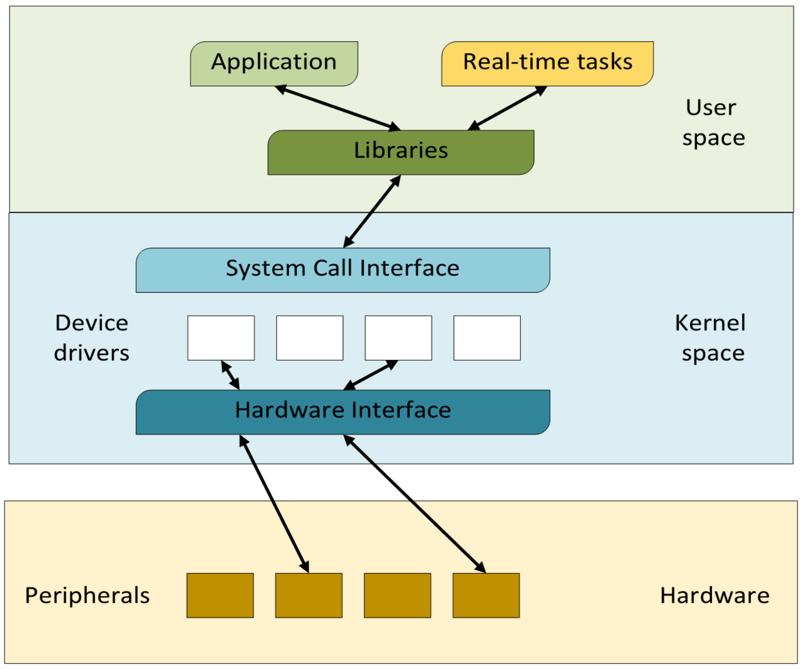


Figure 3.2: Single Kernel approach

Traditionally, all the needed transformations to make the mainline Kernel working real-time have been done though a Kernel patch: the so called PREEMPT_RT.
Nowadays, the PREEMPT_RT patch is widely used and it represents the main way to make Linux working real-time. For this reason, PREEMPT_RT will be detailed and analysed in the following of this thesis.

3.1.3 PREEMPT_RT

As already said, the PREEMPT_RT patch represents the most used in-Kernel approach to make Linux usable in real-time scenarios. Moreover, being PREEMPT_RT officially supported, it allows to use the standard POSIX without the need of special APIs for writing real-time applications.

Therefore, PREEMPT_RT is well supported, and being it highly accepted by the official community, most of its features are already included into the mainline Kernel. Supporting this, Linus Torvalds, in a summit in 2006, said: ”Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with your using PREEMPT_RT”

Once understood that PREEMPT_RT seems to be the best solution for making Linux time deterministic, one question could be:

What does PREEMPT_RT really do?

Before going on and analysing what the PREEMPT_RT patch actually does, it could be useful to have a look at the most common causes of non-determinism in a Linux system.

3.1.3.1 Sources of latency and non-determinism

As already stressed, real-time applications need determinism in order to be sure they complete their jobs before their deadlines. Unfortunately, different reasons can cause real-time applications missing their deadline due to increasing latencies of the system.
Among the most common causes of non-determinism existing in a generic Linux system there are:

- **Scheduling algorithm:** real-time threads need be scheduled before other tasks, therefore, a real-time scheduling policy is needed. Moreover, the policy has to manage the various priorities that have to be assigned depending on the tasks deadlines.
- **Scheduling latency:** the real-time kernel must be able to reschedule as soon as events occur, e.g when an interrupt is triggered, the sooner it is managed the better it is. Reducing scheduling latency represents a key point, hence, this will be detailed in the following of this section.
- **Preemption not enabled:** it can happen that during the execution of critical sections the preemption mechanism is disabled. Of course, this problem can lead to unexpected latencies due to processes that cannot be preempted. Therefore, in real-time scenarios do not disable preemption is a must.
- **Priority inversion:** this is a well known problem that may lead to unbounded latencies due to a higher priority thread blocked on a mutex held by a lower priority task. One of the most used solution to cope with this problem consists in implementing the priority inheritance mechanism. This temporally boosts the lower priority thread, holding the mutex, to have a higher priority in order to do not lock the high priority task for too much time.
- **Accurate timers:** when the deadlines to meet are really small (a few milliseconds or microseconds) high resolution timers are crucial in order to be sensitive to small time quantities.

- **Page faults:** page faults while executing real-time tasks can cause unexpected latencies, therefore, some mechanisms through which memory can be locked are required.
- **Interrupts:** because they can occur with unpredictable time rates, the real-time processes latency may grow significantly, especially when multiple interrupts occur one after the other. To cope with this problem, one solution can be running interrupts as kernel threads, or, in case of multi-core CPUs the interrupt handling could be committed to one core devoted only to this purpose.
- **Processor caches:** caches provide a high speed buffer between CPU and main memory, however, by their nature they are sources of non-determinism especially on multi-core devices.
- **Memory bus contention:** usually, when memory transfers happen directly through DMA peripherals, the channel used to move data is shared with the CPU that, depending on the available bandwidth, can slow down the execution of real-time tasks. Therefore, when DMA channels are used the latencies can increase.
- **Power management:** real-time and power management policies totally go in opposite directions. In fact, power management often leads to higher latencies due to the switching from one sleep state to another that unavoidably takes time. Many are the causes that do not allow instantaneous transitioning: starting from the clock frequency generator that takes time to settle, up to the voltage regulator devices that take time to provide stable outputs. Therefore, for sure, a device exiting from a low power state will not respond immediately to interrupt or other stimuli and the latencies will result increased.

3.1.3.2 The scheduling latency

As anticipated, the scheduling latency represents a key point in real-time systems. Indeed, in order for real-time tasks do not miss their deadlines, they need to be scheduled as soon as they have something to do.
However, in real systems, even when the CPU is idle and there are no other threads of the same or higher priority running, there is always some delay between the time instant in which the wake up event occurs and the time in which the corresponding thread starts executing.
This delay is better known as the scheduling latency, and since it is made up of different contributions it can be broken down into several distinct components, as shown in figure

3.3.

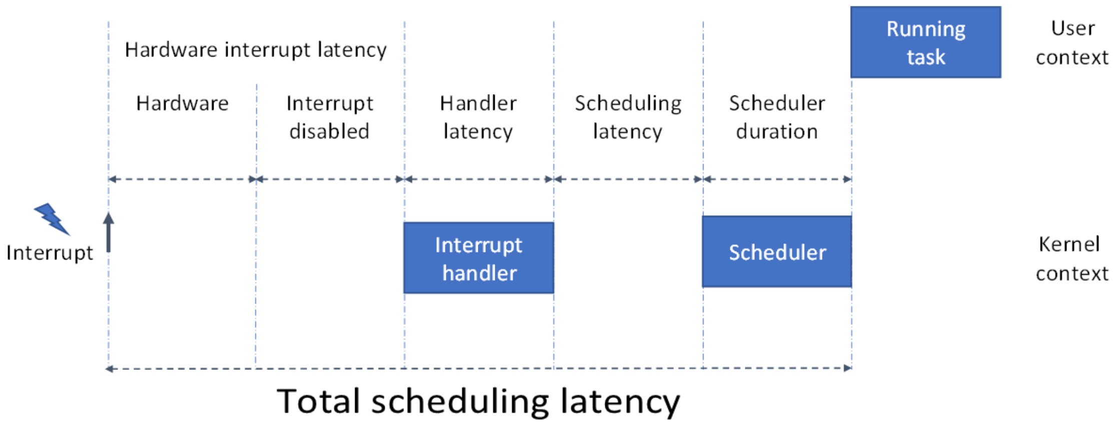


Figure 3.3: Scheduling latency

Analysing the figure 3.3, the first delay contribution is the hardware interrupt latency: it goes from the point at which an interrupt is asserted until the starting of the corresponding Interrupt Service Routine (ISR). In its turn, this latency is made up of two parts: a first, usually small, delay is given by the interrupt hardware itself while, the remaining part is due to interrupts disabled via software. Therefore, minimizing the time in which interrupts are disabled it is really important.
The next delay is the handler latency: it is given by the time the ISR takes to execute, this delay mostly depends on how the routine has been written, and hopefully, it should take just a short amount of time in the order of micro seconds.
Once the ISR is executed, there is the scheduling latency: it goes from the time point in which the Kernel is notified to run the scheduler at the point in which it is actually executed.
The final latency is given by the scheduler duration: it is the amount of time the scheduler algorithm takes to execute for deciding when start running the real-time thread.
Of course, the scheduler latency and its duration depends on whether the Kernel can be preempted or not, in fact, if it is running some piece of code in critical section, the reschedule will be late and the total latency will result increased.

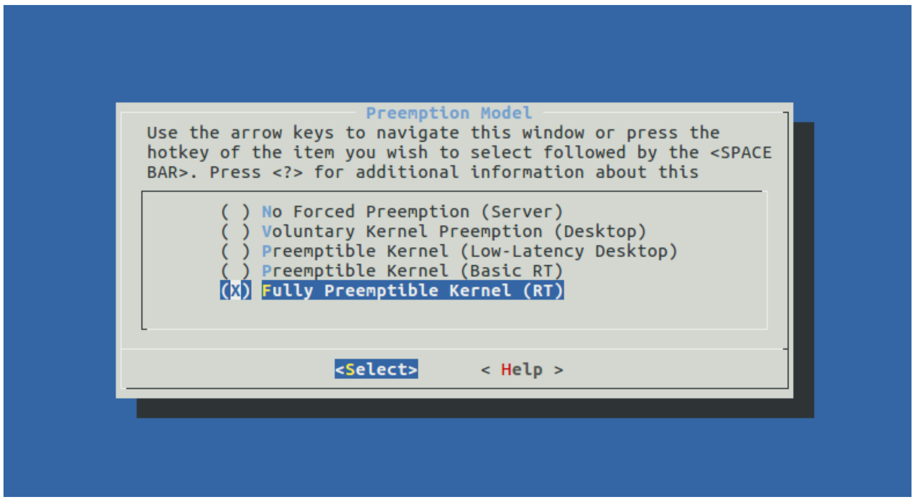
Now that the main causes of non-determinism have been analysed and what the scheduling latency is has been understood, the question "What does PREEMPT_RT really do?" can be answered.
Basically, the main thing done by the PREEMPT_RT patch consists in making the Linux

Kernel fully preemptible while solving the main causes of non-determinism to reduce the latencies.
Entering more in details, these goals are achieved by:

- Transforming the In-kernel locking primitives in preemptible ones, using spinlock and though their reimplementation with real-time mutexes.
- Reimplementing most critical sections with the new preemptible locking primitives
- Solving the priority inversion problem, for intra Kernel spinlocks and semaphores, with the priority inheritance mechanism.
- Managing the interrupt handlers with preemptible Kernel threads, i.e., the PREEMPT_RT patch manages the soft interrupt handlers as Kernel threads.
- Improving the old Linux timer APIs in order to manage high resolution Kernel timers also in user space context.

As said, over the years, many of these features have been incorporated into the mainline Linux, in fact, a standard Linux Kernel already includes high resolution timers, Kernel mutexes, and threaded interrupt handlers.
However, all the Kernel core modifications aiming at reducing the amount of time the Kernel spends running in an atomic context, i.e., the modifications that increase the amount of time in which the Kernel is preemptable, are maintained outside the mainline because they are rather intrusive.
In fact, including these modifications in mainline only a small percentage of the total Linux users could take advantages because the average latencies are made higher but much more deterministic, that is what PREEMPT_RT is for.
Another question that could arise at this point is related on how to apply the PREEMPT_RT modifications to a standard Kernel.

As the name suggest, the PREEMPT_RT patch is a Kernel patch that has to be applied to the source code, hence, to enable the full real-time functionalities the Kernel source code along with the PREEMPT_RT patch are needed.
Besides the Kernel source code patching, the fully preemptable Kernel has to be enabled through the PREEMPT_RT_FULL Kernel configuration option, as shown in figure 3.4, and practically explained in section 4.2.2.



```
                        Preemption Model
 Use the arrow keys to navigate this window or press the
 hotkey of the item you wish to select followed by the <SPACE
 BAR>. Press <?> for additional information about this

          ( ) No Forced Preemption (Server)
          ( ) Voluntary Kernel Preemption (Desktop)
          ( ) Preemptible Kernel (Low-Latency Desktop)
          ( ) Preemptible Kernel (Basic RT)
          (X) Fully Preemptible Kernel (RT)

                  <Select>      < Help >
```

Figure 3.4: Kernel configuration menu

3.2 Real-time benchmarking

As analysed, the PREEMPT_RT patch represents a quite common way to make the Linux Kernel working real-time.
Once the patch is applied and the real-time Kernel has been compiled, the most common problem consists in identifying whether or not the specific hardware and its configuration are able to let the applications running without they miss their deadlines.

To deal with this kind of problem, a mathematical proof saying that the applications will never miss their deadlines would be perfect.
Unfortunately, the Linux systems are so complex that it is basically impossible to demonstrate mathematically that a task will always run without missing its deadline. Therefore, the only way to establish whether or not the application, running on a given hardware, is going to miss its deadline consists in performing practical device measurements.
Entering more in details, different roads can be identified and followed to perform these measurements: maybe, the most straightforward option is to run the actual real-time application in order to get whether its deadlines are met or not. However, for different reasons, unfortunately, not always it is possible to run the actual application, hence, some other methodology is needed.

Specifically, in order to understand whether the device is real-time enough or not, the scheduling latency of the system can be measured and analysed.
In fact, as shown in figure 3.5, the smaller the scheduling latency, i.e., the amount of time

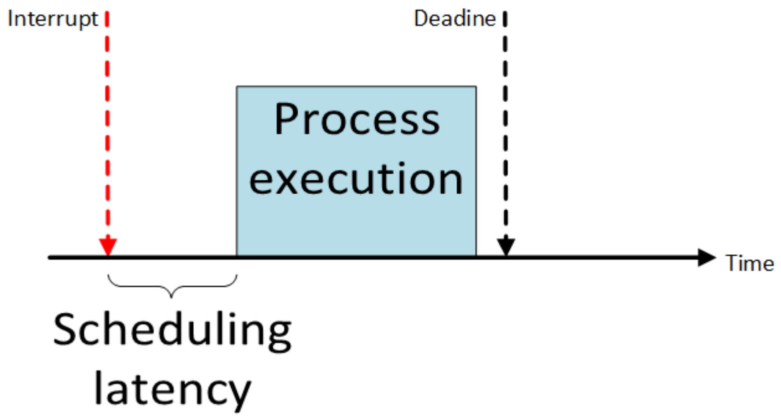


Figure 3.5: Scheduling latency and remaining time

the system needs for responding to the occurrence of an event, the bigger the remaining time available to execute the real-time computations.

To sum up, measuring the scheduling latency can help to get how much real-time a device is. Hence, hereinafter the main tools and options to measure these latencies will be presented and analysed.

3.2.1 RT-tests

The *rt-tests* is a test suite that contains tools to test various real-time Linux features and also the scheduling latency: the main tools taking part of *rt-tests* are:

- cyclictest
- hackbench
- pip_stress
- pi_stress
- pmqtest
- ptsematest
- sigwaittest

3.2.1.1 Cyclictest

Cyclictest is a widely used tool to verify the maximum scheduling latency: it runs a master non real-time thread that in its turn starts a given number of "measuring" threads with a defined real-time priority (scheduled with SCHED_FIFO), then the "measuring" threads are woken up periodically with a defined interval by the expiring of a timer and every time they are woken up the difference between the programmed and the effective wake-up time is calculated and given to the master thread.
Finally, the master thread tracks the latency values and prints minimum, average and maximum value along with other useful information.
To better understand how Cyclictest works its source code can be analysed, and for this reason, in 3.1 and 3.2 a simplified version of its source code is reported.

```
void *timethread(void *par){

        //Thread set up
        clock_gettime(&now);
        next = now + interval;
        while(!shutdown){

                clock_nanosleep(&next);   //Sleep
                clock_gettime(&now);      //Get current time
                diff = now - next;        //Compute the difference
                update_stat(diff);        //Update statistics
                next += interval;         //Compute the new
                   wake-up time
        }
```

Listing 3.1: Cyclictest measuring thread

```
int main(){

        for(i=0; i<num_threads; i++){

                pthread_create(timethread);     //Creates threads
        }
        while(!shutdown){

                for(i=0; i<num_threads; i++){
                        print_stat(stat[i], i);       //Prints
                           statistics
                }
                usleep(10000);
        }
        if(histogram){

                print_hist(parameters, num_threads);
        }
```

Listing 3.2: Cyclictest main thread

Cyclictest analysis

Dealing with Cyclictest some of the most important elements to keep in mind are:

- The main thread runs with a lower priority, therefore, it is like a normal thread.
- The "measuring" threads run at higher priorities, i.e., the one specified by the user.
- Cyclictest does not provide deterministic results.
- Just running Cyclictest gives a lower bound in terms of latency, in fact, it does not give the worst case unless the CPU is under load.
- Running Cyclictest with the CPU idle is almost useless.
- Cyclictest settings have to be tuned accordingly to the system one wants to test.
- Without taking into account the system properties it can happen that some cause of delay are not covered. For example: because Cyclictest wakes up threads with a regular cadence, there may be delay sources that have been already managed before the Cyclictest threads run and in this case it could be helpful to run Cyclictest with different periods.
- Cyclictest does not measure the Interrupt Request (IRQ) handling path of the real-time application, in fact, typically the timers IRQ are fully handled in IRQ context and, in more realistic scenarios, the handler wakes up the corresponding thread that in its turn wakes up the real-time process.

To sum up, among the various elements to keep in mind, as already stressed, Cyclictest must be run under specific load conditions to determine the worst-case latency of the system. Therefore, possible solutions for using Cyclictest in the right way are:

- Do not use Cyclictest embedding the latency measurements into the real-time application: actually this is not a real solution, but, embedding inside the real-time application the measurements in such way that latency can be measured directly by the application represents the best approach even though it is not always possible.
- Run the normal real-time application and non-real-time applications as system load while Cyclictest is running to measure latency.

All the proposed solutions require to have the system real-time application. However, there are situations in which the real application cannot be used and some other alternative has to be found.

Synthetic benchmarks

When the performance of a system has to be analysed and the actual final application of the system is not available, some other benchmarking alternatives are needed. In this case, synthetic benchmarks can come in help: they are test scenarios designed to be easily repeatable, in order to obtain results that can be accurately compared.
Synthetic benchmarks, through isolated investigations, allow to check also the minimal bottlenecks within a system, i.e., they are based on the idea of testing individual parts without the interaction of other factors affecting the results. Therefore, each component of the system is tested alone, like: the Processor before, then the network interface, the disk and so on.
These benchmarks can be really useful when the actual application is not available, but, they have to be defined carefully like will be explained in section 4.2.

3.2.1.2 Hackbench

Hackbench is both a benchmark and a stress test for the Linux kernel scheduler, its main job is to create a specified number of threads or processes which communicates via sockets or pipes. Moreover, it measures how long the communication takes acting as benchmark tool. As already stressed an idle kernel shows lower scheduling latencies, therefore, Hackbench can be used as stress program to obtain a busy kernel in order to get more realistic latencies results.

Pip_stress

Pip_stress creates a priority inversion using three processes. Specifically, the process with the lowest priority holds a mutex and it is preempted by the medium priority process that just runs an infinite loop. Then, the third process, the one with the highest priority, tries to lock the mutex already held by the lowest priority task. Since a priority inheritance mutex is used, the process with the lowest priority is allowed to run at higher priority such that it can preempt the second process with the medium priority. The Pip_stress test program does not take any options as input and if the priority inversion problem is solved it should exit instantly. Thus, Pip_stress is mainly used to test whether or not the priority inheritance mechanism works fine.

Pi_stress

Pi_stress is a program used to stress the priority inheritance mechanism. It runs groups of threads. Specifically, each group of threads causes a priority inversion condition that will deadlock if priority inheritance does not work.

Pmqtest

Pqtest starts pairs of threads and measures the latency of interprocess communication with POSIX messages queues. More in details, it measures the latency between sending and receiving of messages.

Ptsematest

Ptsematest starts two threads and measure the latency of interprocess communication with POSIX mutex. Specifically, it measures the latency between the releasing and the getting of a mutex.

Sigwaittest

Sigwaittest starts two threads that are synchronized via signals and measures the latency between sending a signal and returning from *sigwait()*. Optionally, it can also start two processes rather than threads.

3.3 Profiling Tools

When one wants to understand how a program works, source level debuggers, like the ones described in section 2.2.1.2, can be used. Debuggers allow to get an insight into what the system is doing. However, they constrain the view only to a small portion of code, i.e., the debugging operation is restricted only to a specific application.

Most of times, classic debugging methods can be fine, but, sometimes it is needed a way to get a high level overview of the system in order to see which processes are running, how much CPU and memory resources are using, and so on.

Therefore, some alternatives to the classic debug techniques are needed, such that: larger pictures, saying whether the system is performing as intended or not, can be obtained.
These sophisticated alternative tools are the so called profiling and tracing tools: they aim at identifying where bottlenecks are in order to solve performance issues, while analysing the whole system. In other words, profiling tools offer methodologies to get a high level overview of what the system is doing, analysing how the available resources are managed, hence, when a system has some performance issues it is wise to use these kinds of tools in order to analyse what is going on.
Maybe the most known profiling tool in Linux environments is the so called *Top*, it shows a list of tasks that are currently managed by the Kernel and other system summary information.

Along with *Top*, also other tools exist, for example, the profiling tool *Perf* provides a much more powerful mean to analyse and profile applications. Alternatively, if the problems and the bottlenecks are due to the Linux Kernel, trace tools like *Ftrace* and *LTTng* offer means to gather all the needed information.
To sum up, profiling tools are really useful to dynamically analyse and measure the system status along with the usages of its resources. Moreover, using profiling tools it is also possible to get the complexity of a program, the number of usage of a given instruction, the frequency and duration of system calls, the number of branches correctly predicted and so on. Therefore, very commonly, profiling tools are used to optimize programs executions and to get where problems within a system, in terms of performance, are.

3.3.1 ARM DS-5 Streamline

ARM DS-5 Streamline, by ARM, is one of the various profiling tools available, and as claimed by ARM: "DS-5 Streamline gives you better insight into how your software executes than ever before." [13]
Streamline is a performance analyser, optimized for ARM devices, it represents the ideal tool for developing optimized applications that run on ARM SoCs. It allows to see how the system is performing, and for example, through its time-line view, charts showing the values of performance counters and the processes in execution details are easily readable, as shown in figure 3.6.

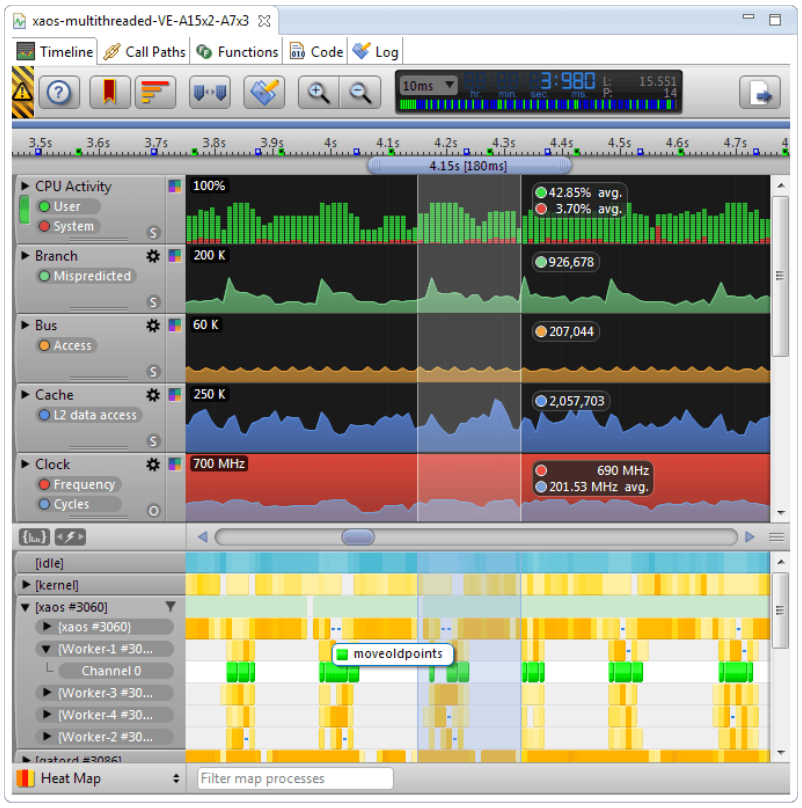


Figure 3.6: Streamline screen-shot

Streamline allows selecting the information to show independently for each process and thread in order to analyse just what is useful for the specific use case. The DS-5 Streamline suite can be easily used through an Ethernet connection and it is made up of three main software components: Streamline running on the development host, the Gator Kernel module which continuously gathers the needed information and the Gator daemon that manages the connection between the host and the target. The working principle of Streamline and its components is summarized in figure 3.7.

3.3.1.1 The Performance Monitoring Unit (PMU)

As described, using Streamline it is possible to analyse the various processes running on the system, obtaining detailed information such as: the number of miss-predicted branches or the number of cache misses.

Streamline gets all the needed information through the usage of the Performance Monitoring Unit (PMU) integrated into the SoC. Indeed, the PMU helps in gathering the useful performance information through the monitoring of the SoC resources.

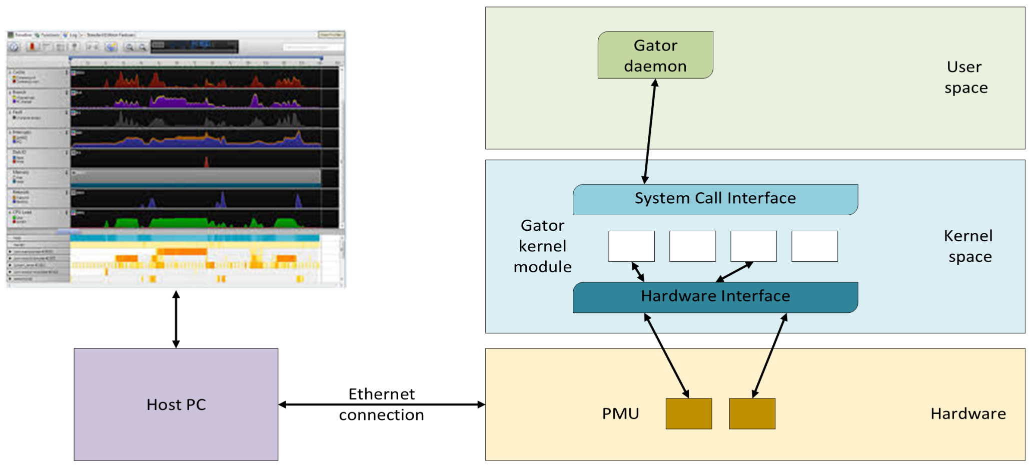


Figure 3.7: Streamline and Gator working principles

Chapter 4

Tests and measurements

This chapter will present the main test scenarios along with their specific configurations and results. In addiction, before introducing the actual measurements, the under test devices characteristics will be summarized when it is possible. Finally, the results will be analysed, and the main bottlenecks in terms of latency, due to problems in cache coherency, will be outlined proposing a solution to reduce the latency and explaining the methodologies used to discovery and mitigate the problem.

4.1 The used devices

Before starting analysing latency results and test scenarios, it could be interesting to get an overview of the used hardware. For this reason, in the following sections, the used devices will be described and their main characteristics will be listed.
However, since during tests also commercial devices have been used, they will be just described in terms of their main characteristics without specifying the actual vendors.

4.1.1 SABRE Board for Smart Devices

The main device, actually involved in all the tests and analysis, has been the NXP Smart Application Blueprint for Rapid Engineering (SABRE) board for smart devices: it is a development board based on i.MX6 Quad core processor widely used for introducing developers to multi-core processors with low-power consumption characteristics, multimedia and graphics capabilities.
In figure 4.1 the iMX6 SoC block diagram is shown, and below its main characteristics are summarized:

- Quad core ARM Cortex-A9

- Multilevel memory system
- Dynamic voltage and frequency scaling
- Powerful graphics acceleration
- Interface flexibility
- Integrated power management throughout the device
- Advanced hardware-enabled security

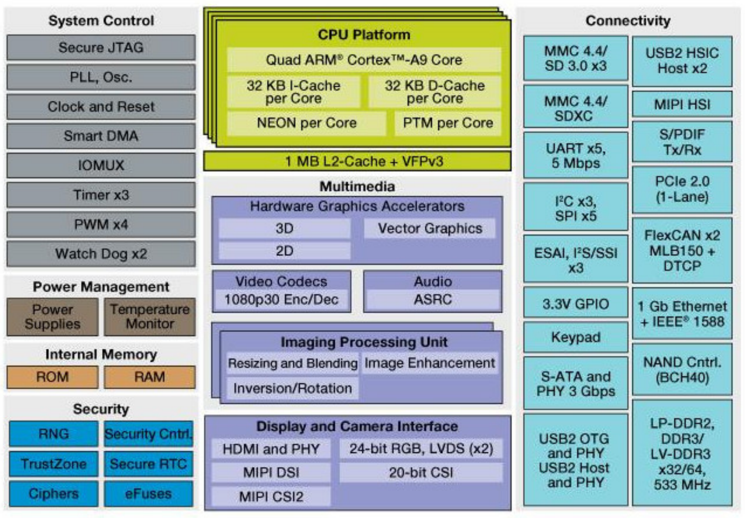


Figure 4.1: iMX6 blocks diagram

The SABRE board provides a low-cost development platform including all primary features of the i.MX6 processor in conjunction with high-speed interfaces and memories. Its hardware design files are easily available along with many accessory boards that can work with the SABRE board to provide additional capabilities such as multi-touch display and Wi-Fi connectivity.

Moreover, being the SABRE board supported by the NXP community, optimized BSPs for Android and Linux along with optimized video, speech and audio codecs are available [8]. Specifically, also BSP layers to be used with the Yocto Project are available and can be used to obtain working and optimized Linux distributions.

Features

The main features offered by the SABRE board for smart devices are:

- i.MX6 QuadPlus 1 GHz applications processor
- 1 GB DDR3, 533 MHz
- 8 GB eMMC NAND
- Two SD card slots
- Serial Advanced Technology Attachment (SATA) 22-pin connector
- HDMI connector
- Two Low-voltage Differential Signaling (LVDS) connectors
- Liquid Crystal Display (LCD) expansion port connector
- Serial camera connector
- Two 3.5 mm audio ports (stereo HP and microphone)
- USB On-The-Go (OTG) connector
- Debug out via USB uAB device connector
- Gigabit Ethernet connector
- Joint Test Action Group (JTAG) 10-pin connector
- mPCIe connector
- 3-axis accelerometer
- Digital compass

In figure 4.2 a picture of the SABRE board for smart devices is shown, and from there, all main components are recognizable and visible.

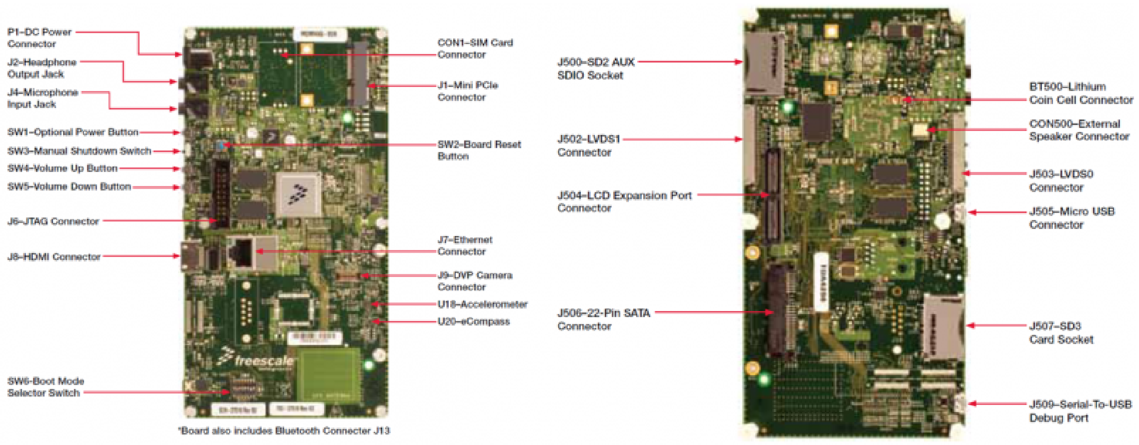


Figure 4.2: SABRE board [15]

4.1.2 Commercial devices

As anticipated, during the tests also commercial devices have been used. Although the vendors' names will not be specified, the main devices characteristics will be listened hereinafter, and starting from now on these two devices, involved into the benchmarks, will be referred as SBC-1 and SBC-2.

- **SBC-1**
  - i.MX6 QuadPlus 800 MHz applications processor
  - 2 GB DDR3
  - 8 GB eMMC NAND
  - SD card slot
  - 2 USB connectors
  - Touch-screen
  - Debug via Universal Asynchronous Receiver-Transmitter (UART)
  - 3 Gigabit Ethernet connectors
- **SBC-2**
  - i.MX6 QuadPlus 1 GHz applications processor
  - 1 GB DDR3
  - 8 GB eMMC NAND
  - SD card slot

- 1 USB connector
- Touch-screen
- Debug via UART
- 2 Gigabit Ethernet connectors

4.2 Test scenarios and results

As anticipated in section 3.2, when the performance of a Linux real-time system have to be measured, different approaches can be followed. The most straightforward method consists in measuring the real-time latencies while the system is running the actual application that will be executed in the final product. Of course, following this approach brings to have the most reliable results expressing how the real system will behave.

However, not always this method can be followed due to different reasons. In fact, it can happen that one does not have the final application or simply the final application cannot be released for testing purposes.

What to do if the final application is not available?

When the final application cannot be used for testing purposes, as anticipated in section 3.2.1.1, synthetic benchmarks can be used to estimate how the system will behave under different load scenarios. Following this method, one of the most important step to be done, in order to test the system real-time performance, consists in defining the benchmarks structures identifying the right test scenarios.
In fact, in absence of the actual final application, identifying the right benchmarks to be executed by the device represents a crucial aspect, because many are the available options in terms of benchmarking suites as described in section 3.2.

Sometimes, if one wants to produce a sort of acceptance test suite with the main goal of evaluating different hardware platforms offered by various vendors, it can be useful to create test cases easy to reproduce in order to let vendors perform the same executed tests when something does not work as expected.

Within this work, repeatability has been a key concept used for developing the various benchmarks. Hence, all the used test programs have been chosen open source, in such a way to let the repetition of all the test cases an easy operation.

Going into details, this section will introduce the main tests scenarios, in terms of used software and their configurations. Then, the obtained results will be presented and analysed.
It is important to notice that, for the sake of simplicity, just the most meaningful results will be reported, nevertheless, this does not imply losing of information thanks to the produced summary graphs that include all the useful data.

4.2.1 The real-time task

Focusing on the main goal, i.e. testing the real-time system performances, to effectively test a device, a real-time task capable to measure the scheduling latency is needed. As anticipated in section 3.2.1.1, one of the most used program to measure the scheduling latency, of a Linux real-time system, is Cyclictest.
Therefore, during the tests executions, in all the cases, the real-time process has been emulated using Cyclictest, in addition to the real-time process emulation, Cyclictest also helped in gathering all the latency information that will be reported in this chapter.
Cyclictest offers many options that can be used to create different test scenarios, however, during almost all test cases its options remained the same. Specifically, the executed command for Cyclictest is reported below:

```
$ cyclictest -S -l "loops" -m -p 90 -i "interval" -q -h400 >
   output
```

Listing 4.1: The cyclictest command

What do the Cyclictest options mean?

- -S, this option:
  - Keeps the specified priority equal across all threads.
  - Sets the number of threads equal to the number of available CPUs.
  - Assigns each thread to a specific processor (affinity).
  - Uses *clock_nanosleep* instead of POSIX interval timers.
- -l "loops" sets the number of loops to be executed.
- -m locks current and future memory allocations to prevent memory being paged to the swap area.
- -p 90 sets the threads priority.

- -i "interval" sets the threads base interval in microseconds.
- -q runs the test quiet and prints a summary just on exit.
- -h 400 gives the latency histogram to *stdout*, 400 is the maximum time to be tracked in microseconds.

Usually, when a real-time application has to run on a target hardware, it is characterized by a well defined period. However, if one is interested in executing synthetic benchmarks that are not fully characterized, like in this case, it can be interesting to know how the system works when the real-time application has different periods.
For this reason, all the executed tests have been repeated multiple times, and every time a different period has been imposed to the real-time task. More in detail, all the tests have been executed three times setting the real-time tasks to have the following periods:

- $100\mu s$
- $1ms$
- $10ms$

Of course, the real-time task period can be easily set acting on the Cyclictest *-i "interval"* option, thus, just running three times the command with different *"interval"* values has been enough.

4.2.2 CPU almost idle

The first test scenario, as the name suggests, has been characterized by simply running the real-time task without any other particular CPU load. Of course, this test case does not provide so much useful information about the system performances, indeed, running Cyclictest without any load just gives a lower bound in terms of latency.
However, it can be useful to run Cyclictest for understanding if everything works fine and if the system has been set correctly. In fact, as will be shown in the following test case, even with a CPU idle if the system under test is not set properly, significant scheduling latency can be observed.

Kernel without PREEMPT_RT

As explained in section 3.1.1, although originally the Linux Kernel was not meant for real-time systems, over the years many attempts to make it working real-time have been done. Nowadays, since applications such as audio or video playback require some real-time capabilities, even the standard Linux Kernel offers some sort of real-time aptitudes, specifically, today a modern Vanilla Linux Kernel offers three main preemption options to improve the system latencies:

- CONFIG_PREEMPT_NONE: No Forced Preemption (Usually used in Server)
- CONFIG_PREEMPT_VOLUNTARY: Voluntary Kernel Preemption (Usually used in Desktop)
- CONFIG_PREEMPT: Preemptible Kernel (Usually used in Low-Latency Desktop)

In addiction to what a standard Vanilla Kernel offers, if the system needs more real-time capabilities, the most popular strategy to improve the already available Linux real-time capability consists in making it more preemptive by applying a patch set known as PREEMPT_RT to the Kernel sources, as explained in section 3.1.3.
Hence, a Kernel patched with PREEMPT_RT adds an additional option to the kernel preemption options, namely the PREEMPT_RT_FULL or Fully Preemptible Kernel (RT).

Just as an academic exercise, in order to have a look at the main improvements offered by a Kernel configured with PREEMPT_RT_FULL, a basic version of the Kernel for the Sabre board has been compiled and the results of Cyclictest, running on that Kernel, have been collected.
Obviously, the obtained results were not the best in terms of latency, in fact, as can be seen in figure 4.3, Cyclictest running on the Kernel without PREEMPT_RT_FULL obtained a maximum latency of $1.2ms$ with respect to the version running on a Fully Preemptible Kernel (RT), visible in figure 4.4, that got a maximum latency of just $26\mu s$.
It is important to notice that the Kernel used during the tests without the PREEMPT_RT_FULL option was already configured to be CONFIG_PREEMPT, i.e. it was set with the most preemptable option offered by a standard Linux Kernel.
In 4.2 are reported the two used Kernel versions, the first one refers to the standard Kernel with CONFIG_PREEMPT enabled, while, the latter is the one of the Kernel patched with PREEMPT_RT and the Fully Preemptible Kernel (RT) option active.

```
root@imx6qdlsabresd:~# uname -a
Linux imx6qdlsabresd 4.1.44-fslc+g6c1ad49 #1 SMP PREEMPT Thu Jul
    19 15:16:14 UTC 2018 armv7l GNU/Linux

root@imx6qdlsabresd:~# uname -a
```

```
Linux imx6qdlsabresd 4.1.38-rt45-fslc+gee67fc7 #1 SMP PREEMPT RT
    Mon Mar 19 22:27:10 CET 2018 armv7l GNU/Linux
```

Listing 4.2: Kernel versions

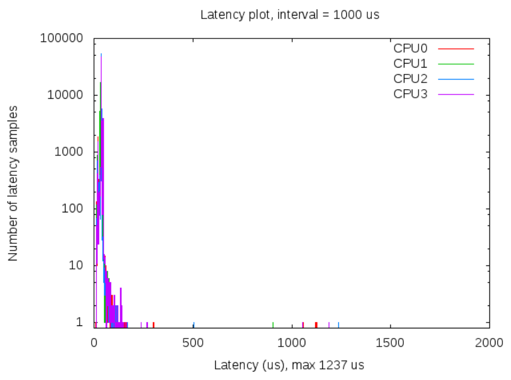


Figure 4.3: Kernel without PREEMPT_RT patch

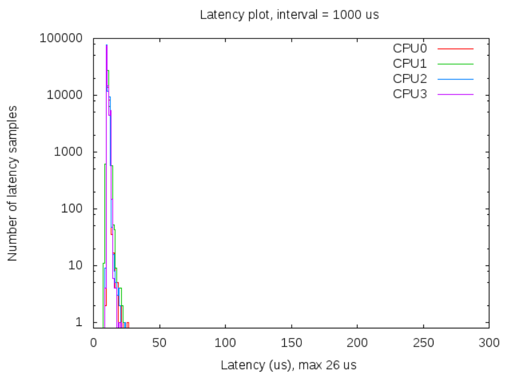


Figure 4.4: Kernel with PREEMPT_RT patch

Coming back to the actual Kernel configuration, namely the one with PREEMPT_RT_FULL enabled, i.e., the one that offers the best real-time capabilities and that is also used by the other devices under tests.

In figure 4.5 the maximum latency values, among all cores, are visible. This bar-graph has been produced starting from latency plots, like the one shown in figure 4.4, and taking into account the maximum value of latency among all the cores. In fact, usually when dealing with real-time systems the maximum value is the one that provides more information, i.e. it says whether the system will miss its deadline or not.

In this first scenario, the Sabre board performed slight better than the other devices. This difference in terms of performances could be attributed to various causes, such as: different hardware configurations, different kernel version and so forth.

Finally, considering the different threads intervals, i.e. the real-time periods ($100\mu s$, $1ms$, $10ms$ and $100ms$), apparently, looking at the Sabre board, better latency results are achieved when the threads interval grows. This last point could be rational since when the threads interval increase, the system should be less stressed.

Additional information about latency average, minimum and variance are provided in table 4.1, for the sake of simplicity they refer just to the Sabre board since it represents the main under test device.

RT task's period	Maximum [us]	Minimum [us]	Average [us]	Variance [us]
100 us	33	8	10	0.25
1 ms	26	8	10.25	0.75
10 ms	22	11	12	0

Table 4.1: Cyclictest results on Sabre board without load

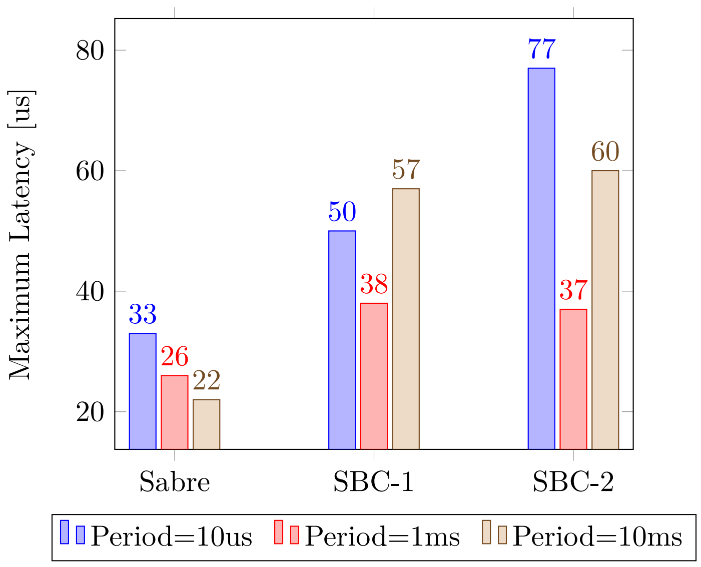


Figure 4.5: CPU almost idle maximum latencies

4.2.3 IPC stress

Measuring real-time latencies with an idle system is quite useless since the CPU has to deal just with the Real-Time (RT) application. Therefore, with this second scenario, the main idea has been to stress the CPUs such that more realistic results were obtained.
As already analysed, the rt-tests suite provides tools also for this use case, specifically, the Hackbench utility (see section 3.2.1.2) can be used as CPU stress source to simulate some sort of system load.
Of course, this scenario follows the idea of synthetic benchmark, creating a more realistic scenario, where Hackbench is used to mimic a system load. Basically, this test case uses: the real-time application, to measure latencies, and Hackbench to stress the Kernel. More in details, the test conditions have been created executing the following commands:

```
$ hackbench -P -g 15 -f 25 -l 200 -s 512 \&
$ cyclictest -S -l "loops" -m -p 90 -i "interval" -q -h400 >
   output
```

Listing 4.3: IPC load command

Regarding the Cyclictest command, it is the same described in section 4.2.1, while the options specified for Hackbench are explained below:

- -P implies to use processes as senders and receivers.
- -g 15 creates 15 groups of senders/receivers.
- -f 25 defines that each sender and each receiver shall use 25 file descriptors.
- -l 200 sets that each sender/receiver pair has to send 200 messages.
- -s 512 sets the amount of data to send in each message equal to 512 bytes.

To sum up, using Hackbench with the described options the Linux Kernel has to manage 15 groups of processes that use $25x2 = 50$ file descriptors sending 200 messages each of 512 bytes. Of course, this test case leads to a high CPU utilization making the system really stressed.
In figure 4.6 the maximum latencies, among all cores and with the different threads' periods, are visible.

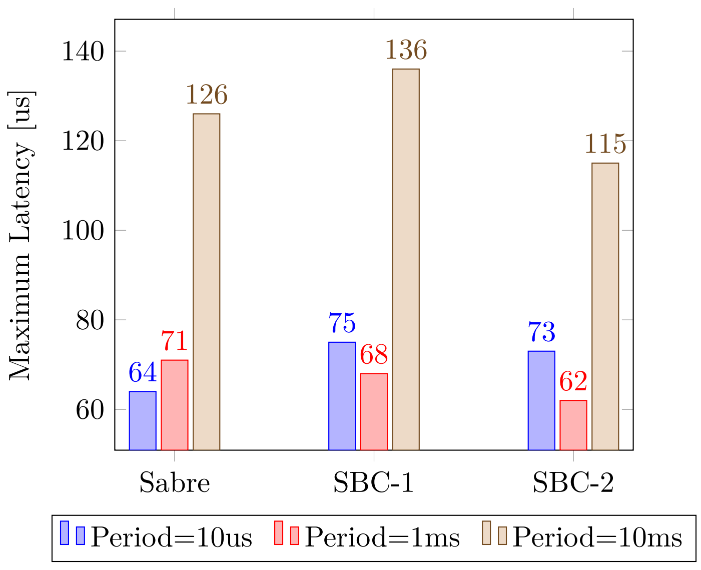
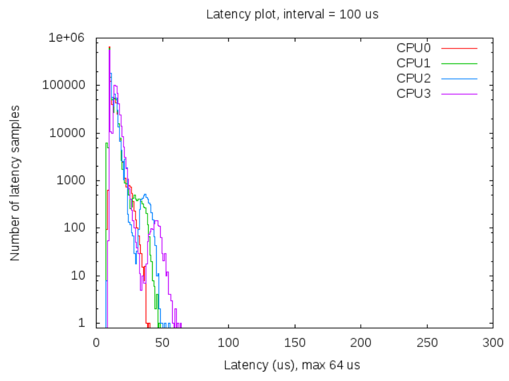


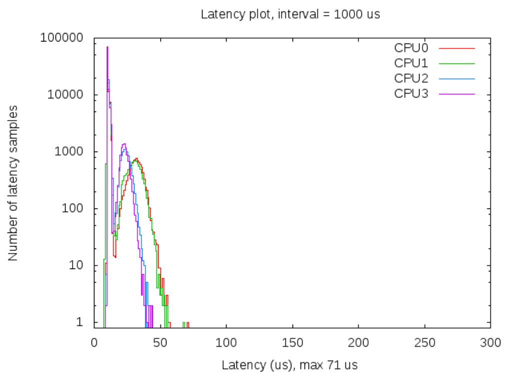
Figure 4.6: IPC load maximum latencies

Also in this scenario, on the average, the Sabre board performed slightly better than the other single board computers. Certainly, this second test case is much more realistic since a system load was emulated using Hackbench and a high CPU utilization is reached.

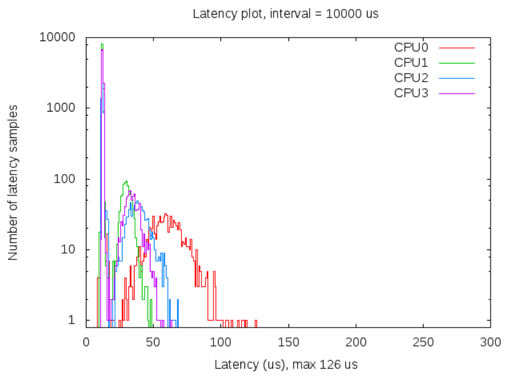
Analysing the obtained results, in this scenario something quite "strange" happened, in fact, threads with lower intervals performed better than threads with bigger period, i.e., the ones that were woken up less frequently got a higher latency. Since this "strange" behaviour came out, in this case, it could be interesting to have a look at the latencies plot reported in figure 4.7, just for the Sabre board.



(a) Period = 100 μs



(b) Period = 1 ms



(c) Period = 10 ms

Figure 4.7: CPU without load

Looking at the differences between figure 4.7b and figure 4.7c, the obtained behaviour, in terms of maximum latency and variability, do not find an easy explanation. Indeed, at first glance one could expect to obtain lower latencies when threads have bigger intervals, i.e., a CPU less stressed, thanks to threads with larger periods, should perform better.

Nevertheless, the obtained results are totally in contrast with what one could have expected, and after some investigation and analysis the cause of this "strange" behaviour was found and the actual motivation will be explained in section 4.3.1.

For now, let's say that a possible cause of this behaviour could be a cache miss problem. In fact, if the real-time threads are executed less frequently and other processes are allowed to execute in the middle (Hackbench or the Kernel itself in this case) it could happen that when the real time thread should be executed it gets a cache miss, because, some cache lines have been replaced due to other processes that have been executed in the middle.

4.2.4 Network stress

Since today almost every device has network connectivity, it could be interesting to have a look at how the system behaves when it is under network load.
Of course, many various scenarios in which the system has to manage network connectivities can be outlined, and for this reason, different test cases have been defined and analysed: specifically, two main benchmarks, each one outlining a different scenario, have been defined and will be presented in the following.

4.2.4.1 Ping

The first outlined scenario, dealing with a network connectivity, uses the well-known Ping network utility tool as background system load.
Ping is a software utility widely used to test the reachability of a host belonging to an Internet Protocol (IP) network.
Basically, ping uses Internet Control Message Protocol (ICMP) echo request to check the reachability of another target host that if it is working properly answers with an ICMP echo reply.
In figure 4.8 the International Standards Organization Open Systems Interconnection (ISO/OSI) model is reported, from there it can be easily seen that ICMP can be considered a third level protocol that uses IP and Ethernet frames, specifically, it has the following properties:

- Is not a transport protocol.
- It is not used to exchange data among devices.
- It produces a lower kernel overhead with respect to transport protocols.
- It is almost the same of using raw IP packets.

To sum up, Ping thanks to its characteristics can in some way be used to emulate raw network traffic such that one created when dealing with network protocols like Ethercat, Modbus, etc.

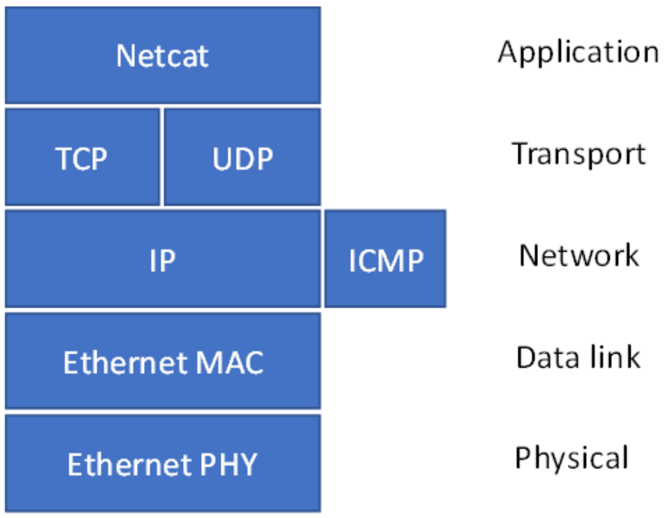


Figure 4.8: Ping ISO/OSI

Coming back to the test scenario, the actual network load was created executing the following commands:

```
$ ping <addr> -l 65500
```

Listing 4.4: Ping load command

Where, the Ping options assume the following meaning:

- <addr> is the device address.
- -l 65500 sets the ICMP message to have the maximum packet size in byte.

Of course, the previous described Ping command was executed along with Cyclictest to measure the real-time latencies, and in figure 4.9 the maximum latency values among all cores are reported.
Analysing the results, apparently, the network load do not affect too much the real-time performance producing a maximum latency of around 70 us considering the SBC-2 device. Instead, the Sabre board even in this case gives the best results with a maximum latency of just 40 us, finally, it is interesting to notice that in this scenario the SBC-2 performed worse than the other devices, however, still acceptable latency values have been obtained.

4.2.4.2 Netcat and disk operations

As anticipated, network capabilities are increasingly required, for this reason it can be useful to evaluate more complex situations such the one involving data transfer over the

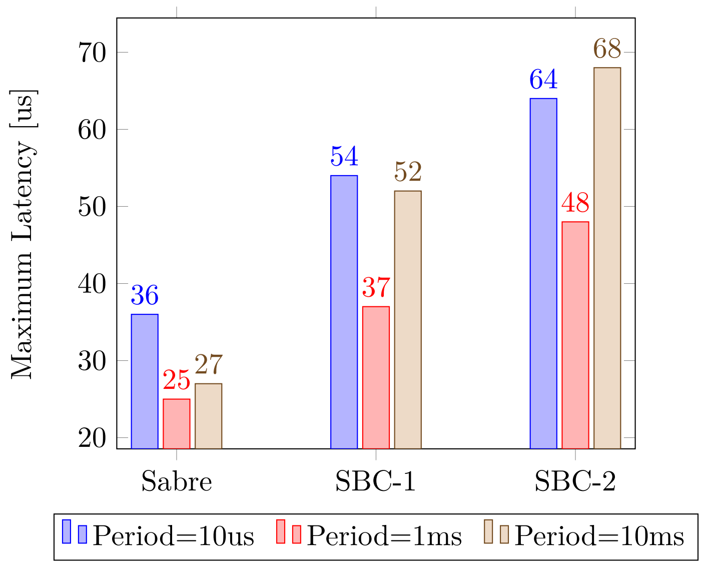


Figure 4.9: Ping load maximum latencies

Transmission Control Protocol (TCP) layer, and to emulate this system load the utility Netcat has been used.

Basically it can read and write over network connections both using TCP or User Datagram Protocol (UDP), the Netcat utility can be also used to transfer files to be saved on storage devices.

Using this feature some other tests have been executed in order to have a look at how the system performs when it has to deal with network load and the additional stress given by disk store operations.

Entering more in details, the embedded device was asked to receive and store 1 Mb of data every second, however, in this case the system was not asked to immediately store the received data leaving the Linux kernel the possibility to cache the coming data and managing the data storage process automatically.

As usually, the maximum latencies among all cores are reported in figure 4.10, from there can be seen that the Sabre board performed better the other devices, and in this case, the worst latency results are given by the SBC-1.

Moreover, without an easy explanation, the SBC-2 device got better results than in the case of the simple Ping load as can be seen looking at the differences between figure 4.9 and figure 4.10.

Of course, many are the parameters that can interact with these kinds of results, however, since latencies did not overcome $100\mu s$ they have been accepted without performing additional analysis.

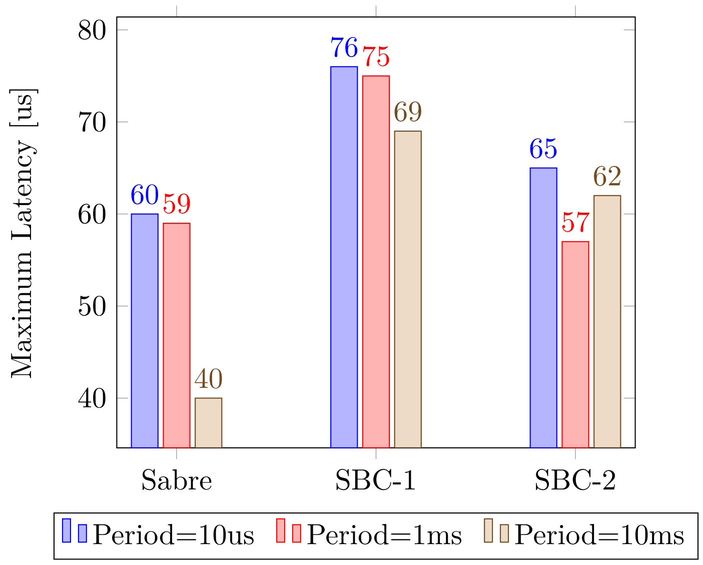


Figure 4.10: Netcat load maximum latencies

4.2.5 Disk stress

Although with the test performed by Netcat disk writings have been done, with this test case the main idea was to intensely stress the storage capabilities.
In fact, after some analysis on how the Linux kernel manages the storage devices, the tests created in this section have been executed by writing blocks of 1 Mb and committing each block to be sure that it was really written on the disk device.
Analysing the obtained results: as usually, the Sabre board got the best results with a maximum latency of around 70 us, while, the worst latency values are given by the SBC-2 that got a maximum latency of 123 us, finally, the SBC-1 did not overcome 100 us of scheduling latency, providing a middle ground between the two previously cited devices.

4.2.6 VLC video

The final test scenario tries to cover the use case in which the device has to manage a video streaming along with the real-time application. As the name suggests, the test conditions have been created reproducing a video file using the well known VLC media player. More in details, the reproduced video file was an MPEG file with a resolution of 720x480 and VLC was executed simultaneously with the real-time process emulated by Cyclictest.
This test scenario has been tested only with the Sabre board, and in figure 4.12 the maximum latencies are shown: as can be seen, the latency increases when the real-time thread's periods become bigger just like happened with Hackbench as system load.

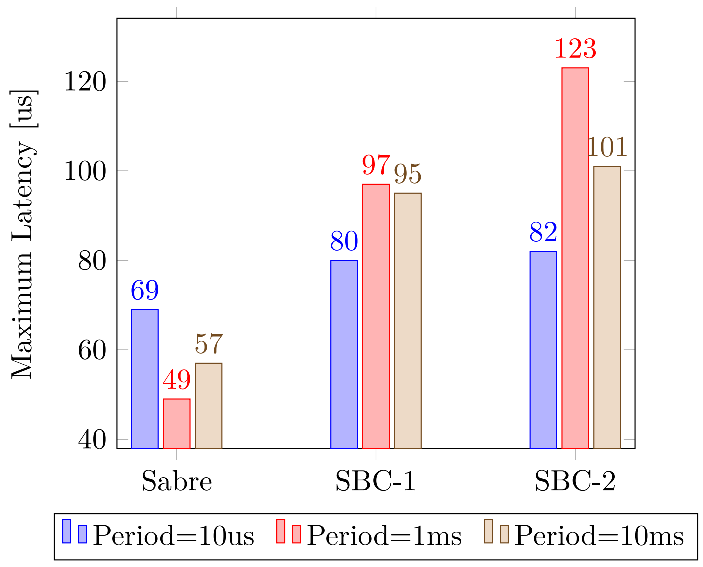


Figure 4.11: Disk load maximum latencies

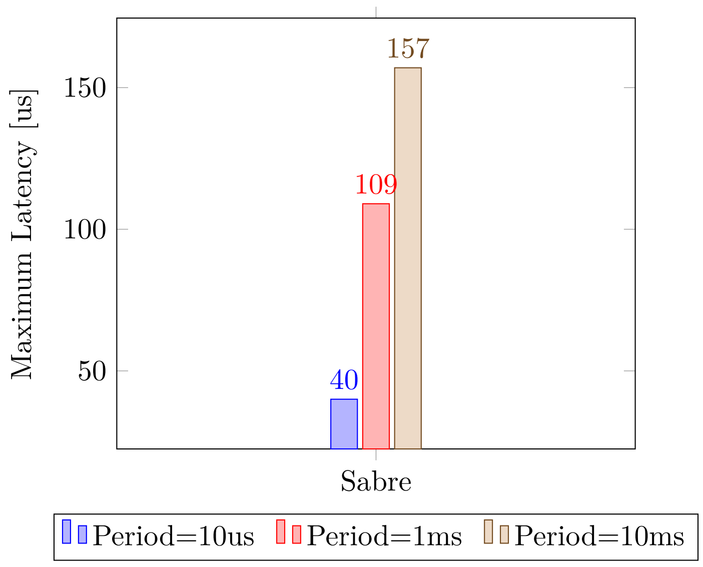


Figure 4.12: VLC load maximum latencies

4.3 Tests analysis

Having seen the various test scenarios and their results, it is possible to say that in almost all situations the analysed systems are able to guarantee a latency that does not overcome 100 us.

However, there are some cases in which the systems presented bigger latencies and "strange" behaviours can be observed. In fact, especially when the real-time applications have bigger periods, latencies overcoming 100us have been registered.

At a first glance, this situation does not find an easy explanation because the system should be less stressed when the real-time task has a bigger period. Nevertheless, having a look on when higher latencies show up, it can be seen that this happens only when the system has to deal also with other tasks, i.e., this "strange" behaviour does not show up when the system has to manage just the real-time application and nothing more.

As anticipated, to better understand what was going on and why these "strange" behaviours were happening, additional investigations have been done using profiling methods such the ones described in section 3.3.

These further analysis will be presented in the following, then, some solutions to that problems will be presented along with the newer results obtained to be compared with the "strange" older ones.

4.3.1 Cache coherency problem

To investigate about the problem that arises when threads have a longer period, most of the previous tests have been re-executed while DS-5 Streamline was running in background.

In this way, accepting some overhead due to the profiling tools, it has been possible to gather all the needed information to better understand what was going on.

Basically, the tests have been run many times while Streamline was gathering profiling information.

Just to have an idea of what has been done, in figure 4.13 Streamline can be seen and from there information related to cache miss of each process can be extracted (see top-right corner).

Running the tests multiple times and gathering information related to cache miss and hit, while considering that independently from the real-time period the number of cache accesses should be almost the same, the plot in figure 4.14 has been obtained: it shows how the latency values are in some way correlated with the cache miss percentage.

To sum up, the graph in figure 4.14 shows that when Cyclictest is run with a larger period, the percentage of cache miss increments and in its turn, also the latency increases. Therefore, when other processes are allowed to run in the middle of two Cyclictest executions the percentage of cache misses increases and bigger latencies are observed.

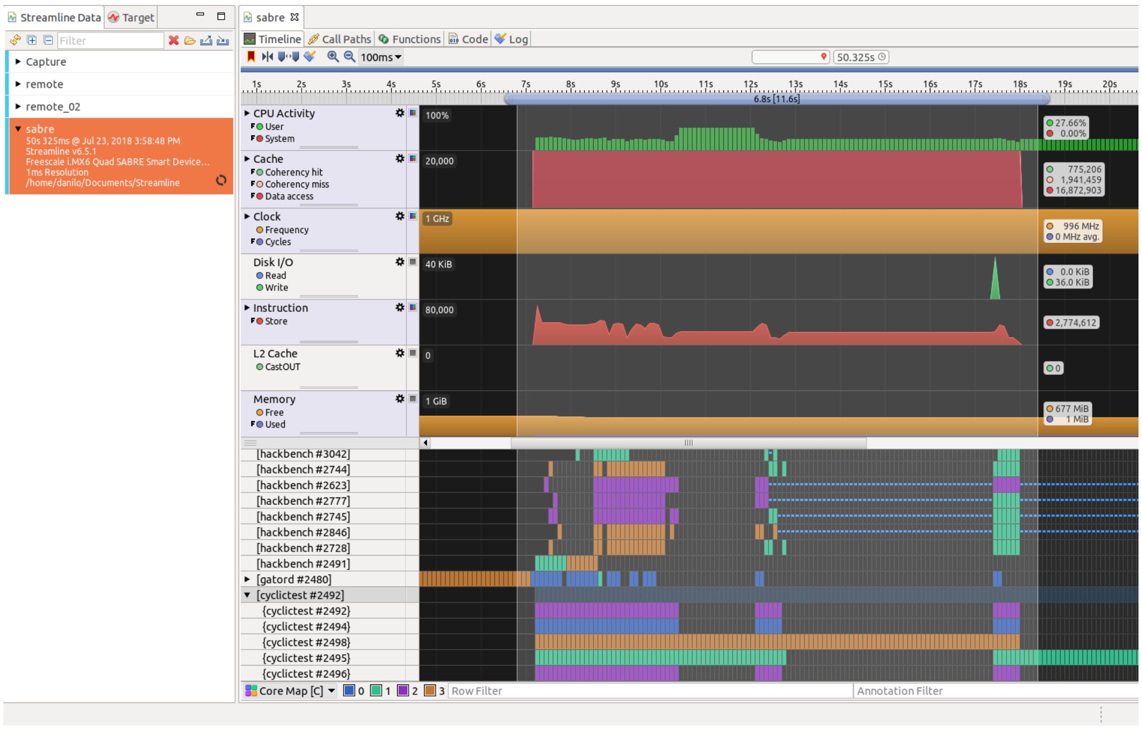


Figure 4.13: Gator cache information

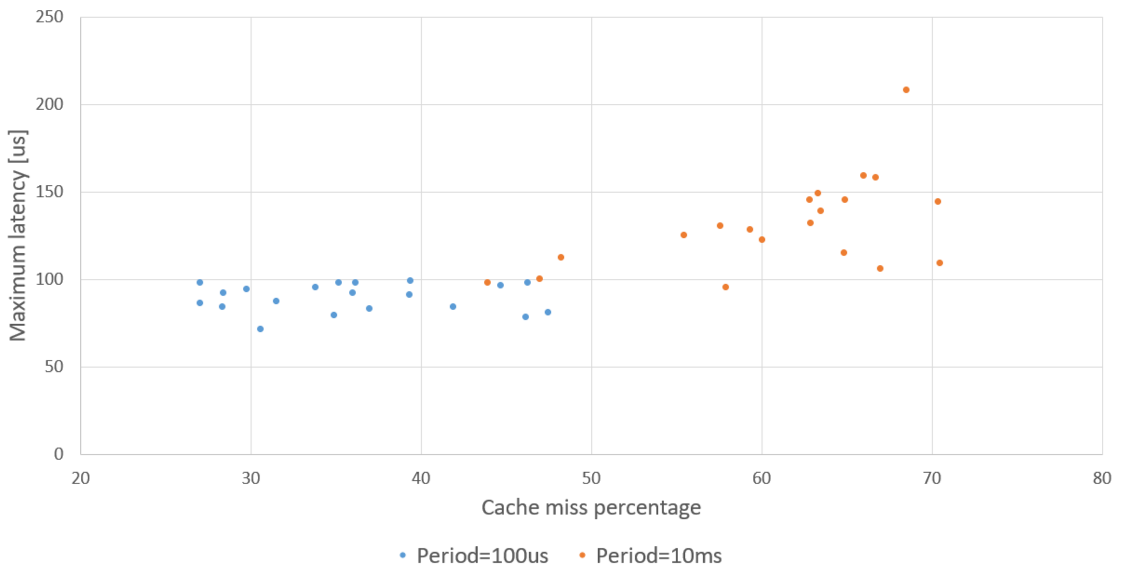


Figure 4.14: cache miss - latency correlation

4.3.2 A possible solution

With the previous tests, an increasing percentage of cache miss has been observed when real-time threads are characterized by longer periods. This happened although all the performed tests have been executed imposing Cyclictest to always run each thread on the same core to avoid further problems with caches.
However, even if the Cyclictest's threads have run always on the same core, the other processes and the Linux Kernel itself have been allowed to run on every core indistinctly. This last point could be used to improve latencies by restricting other processes and Linux itself to run just on some cores, leaving free some other cores that can be uniquely used for real-time processes.

Among the many customization options Linux offers, there is also the possibility to set in which core each process has to run. From a running system it is possible to set the process's affinity using the taskset command and indicating the task's Process Identifier (PID) and an affinity mask, i.e. the process ID and the cores on which that process can run.
Therefore, at a first glance, one could try to create some shell script that uses taskset and assigns all processes to a given number of cores. However, doing this and profiling the system with Streamline it came out that some Kernel threads are not affected by taskset. Thus, after some additional investigations, an alternative way to restrict the entire Linux Kernel to run on a given number of cores has been discovered.
Specifically, to restrict the entire Linux Kernel is such a way that it can run just on the specified cores, the isolcpus boot parameter can be used. The latter is one of the kernel boot parameters that isolates certain cores from Kernel scheduling, i.e., it is useful to leave some cores free from the Linux system.

Hence, understood how to leave some cores free from the Linux system, the tests in section 4.2.3 (the ones reporting latencies grater than 100us and "strange" behaviour) have been re-executed leaving two cores only for real-time tasks.
More in details, from the u-boot command line of the Sabre board, the following kernel parameters have been set:

```
setenv bootargs '....isolcpus=2,3'
```

Listing 4.5: Isolcpu boot argument

The specified parameter allows the Kernel to run only on core 0 and 1 leaving the other cores for real-time tasks.

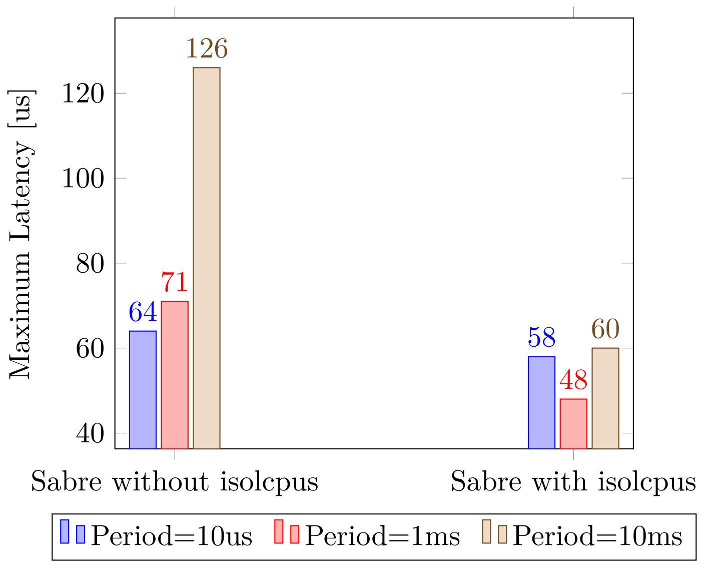


Figure 4.15: Before and after isolcpu comparison

In figure 4.15 the comparison between the case without isolcpus and the one with isolcpus is shown.

From there, it can be seen that the maximum latency values obtained are notably reduced when the Linux Kernel is restricted to run just on core 0 and 1. In fact, also the latencies involving real-time tasks with larger periods (1ms and 10ms) have been definitely reduced.

Chapter 5

Conclusions

This thesis work, that for sure is not fully exhaustive due to the huge number of treated topics, has represented an attempt to cope with the increasing requests of functionalities by the Industry 4.0.
Indeed, with the creation of all-in-one devices, integrating different features, reliability problems can easily arise due to the integration of real-time applications into devices that have to run many functionalities having different urgencies.
Moreover, as explained within the entire thesis, nowadays, the trend goes toward the use of embedded devices using Linux as operating system, and this implies further challenges due to the fact that Linux was not meant as a real-time operating system.
Therefore, some methodologies and tools to check the real-time performances of Linux Embedded systems have been introduced and analysed throughout this thesis work.

Specifically, the main effort has been put in using open-source applications such that tests can be easily integrated into any Linux device (with extreme easiness if the device operating system is built using Yocto Project).
As said, the thesis work has tried to deal with some of the most important related topics: of course, not all arguments have been treated deeply due to their high complexity, like in the case of Linux Kernel and the real-time patch. Moreover, other boundary tools and methodologies, not described within this treaty, have been learned and used to achieve the various goals: for example, the Git version control system played a crucial role for the correct usage of the Yocto Project, but it is not described in this thesis.

To sum up, the presented thesis, developed in collaboration with AROL Group S.p.A., has achieved its main goals, accomplishing the company requests. Specifically, different devices have been used and tested in order to let the company select the right hardware.

More in general, considering the use of Linux for real-time applications, it is possible to claim that the PREEMPT_RT patch offers an effective way to make the Linux Kernel working real-time. In fact, as analysed during the experimental phase of this thesis, the deterministic timing behaviour can be effectively improved a lot when the patch is applied and the Kernel is configured as PREEMPT_RT_FULL.

In particular, it has been observed that even using a common Linux Kernel it is possible to get some real-time capabilities, but in this case, the system has shown latencies greater than $1ms$. However, using the PREEMPT_RT patch the system latencies have been significantly reduced and in all the test cases the latency did not overcome the $200\mu s$.
Therefore, broadly speaking, it is possible to affirm that: using a Linux Kernel patched with PREEMPT_RT, it is possible to run real-time applications having deadlines of some hundreds of microseconds. Nevertheless, the specific system configurations, the real-time application and the other CPU loads influence a lot the device real-time capabilities, hence, each use case has to be analysed carefully, and experimental tests are needed for being sure the application will never miss its deadlines.

Thanks to the Linux complexity and its good malleability capabilities, many customizations can be done to reduce the system latencies, as shown in the latest chapter where the Linux Kernel has been isolated to run just in some cores. Thus, many others customizations are possible to further reduce the latencies depending on the specific use case and some of them will be described in the following section.

5.1 Future developments

As described, many investigations and tests have been done during the thesis development, however, considering the Linux customization capabilities much more can be still done.
For example, it is possible to try setting the interrupt threads affinities such that a given interrupt is always executed on the same CPU core, in this way, possibly depending on the application, some advantages in term of latency can be gained.
Within the thesis work, some profiling tools have been used. However, these used tools introduce overhead and the results could be corrupted. Maybe, it could be interesting to try using less intrusive debugging and profiling techniques in order to get more reliable and precise results.
Other possible roads to be investigated deal with the benchmarking of real-time applications while the system is running graphical operations in background. In fact, through

the usage of the already used profiling tools, like Streamline, it is possible to profile the various applications executions to get aware of unwanted latencies.
Summarizing, different aspects about real-time Linux systems could be still investigated and detailed. Moreover, depending on the actual application requirements some optimizations can be tried to reduce the system latencies.

References

[1] *Arol Closure System*. [Online; accessed 22. Aug. 2018]. 2018. URL: http://www.arol.com/en.

[2] Harry Mulder (Omron Electronics). *The evolution of industrial control systems*. 2017. URL: https://www.processonline.com.au/content/factory-automation/article/the-evolution-of-industrial-control-systems-555674387.

[3] Rolf Ernst and Marco Di Natale. "Mixed Criticality Systems—A History of Misconceptions?" In: *IEEE Des. Test* 33.5 (2016), pp. 65–74. ISSN: 2168-2356. DOI: 10.1109/MDAT.2016.2594790.

[4] "Guide to Industrial Control Systems (ICS) Security ". In: (2013). URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82.pdf.

[5] Mario Hermann et al. "Design Principles for Industrie 4.0 Scenarios". In: *2016 49th Hawaii International Conference on System Sciences (HICSS)* (2016), pp. 3928–3937. ISSN: 1530-1605. DOI: 10.1109/HICSS.2016.488.

[6] *How to add a new layer and a new recipe in Yocto | NXP Community*. [Online; accessed 25. Aug. 2018]. 2018. URL: https://community.nxp.com/docs/DOC-331917.

[7] Darrick Addison (IBM). *Embedded Linux applications: An overview*. 2001. URL: https://www.ibm.com/developerworks/library/l-embl/index.html.

[8] *i.MX 6Quad SABRE Development Board|NXP*. [Online; accessed 21. Jul. 2018]. 2018. URL: https://www.nxp.com/support/developer-resources/hardware-development-tools/sabre-development-system/sabre-board-for-smart-devices-based-on-the-i.mx-6quad-applications-processors:RD-IMX6Q-SABRE.

[9] *Industrial Control System - Definition - Trend Micro USA*. [Online; accessed 10. Aug. 2018]. 2018. URL: https://www.trendmicro.com/vinfo/us/security/definition/industrial-control-system.

[10] *Integrated HMI and PLC.* URL: https://www.isa.org/standards-publications/isa-publications/intech-magazine/2012/august/cover-story-integrated-hmi-plc/.

[11] *Integrated HMI/PLC packages offer convenience, but only in the right applications.* URL: https://www.controleng.com/single-article/integrated-hmiplc-packages-offer-convenience-but-only-in-the-right-applications.

[12] *Linux Crash Top 10 Images.* [Online; accessed 10. Aug. 2018]. 2018. URL: http://www.miguelcarrasco.net/miguelcarrasco/2006/10/linux_crash_top.html.

[13] Arm Ltd. *DS-5 Development Studio | Streamline Performance Analyzer – Arm Developer.* [Online; accessed 10. Aug. 2018]. 2018. URL: https://developer.arm.com/products/software-development-tools/ds-5-development-studio/streamline.

[14] *MPSoC - Wikipedia.* [Online; accessed 22. Aug. 2018]. 2018. URL: https://en.wikipedia.org/wiki/MPSoC.

[15] *NXP - MCIMX6Q-SDB - EVALUATION BOARD, i.MX 6 QUAD SABRE BOARD.* [Online; accessed 22. Aug. 2018]. 2014. URL: https://www.element14.com/community/docs/DOC-67626/l/sabre-board-for-smart-devices-based-on-the-imx-6-series.

[16] *Real-time Linux explained, and contrasted with Xenomai and RTAI.* [Online; accessed 10. Aug. 2018]. 2017. URL: http://linuxgizmos.com/real-time-linux-explained.

[17] Chris Simmonds. *Mastering Embedded Linux Programming.* Packt Publishing, 2015. ISBN: 978-178439253-6. URL: https://www.amazon.com/Mastering-Embedded-Linux-Programming-Simmonds-ebook/dp/B00YSILBY0.

[18] Alexandru Vaduva. *Learning Embedded Linux using the Yocto Project.* Packt Publishing - ebooks Account, 2015. ISBN: 978-178439739-5. URL: https://www.amazon.com/Learning-Embedded-Linux-using-Project/dp/1784397393.

[19] *Why choose Linux for embedded development projects?* [Online; accessed 15. Jul. 2018]. 2018. URL: https://www.eetimes.com/document.asp?doc_id=1277902.

[20] Wikipedia. *Programmable logic controller.* URL: https://en.wikipedia.org/wiki/Programmable_logic_controller.

[21] *Yocto Project Quick Start.* [Online; accessed 10. Aug. 2018]. 2016. URL: https://www.yoctoproject.org/docs/2.1/yocto-project-qs/yocto-project-qs.html.

[22] *Yocto Project web page.* URL: https://www.yoctoproject.org/.

Acronyms

API Application programming interface. 23, 28, 51, 52, 56

ARM Advanced RISC Machine. 20, 51

ASIC Application Specific Integrated Circuit. 8

BSP Board Support Package. 40–43, 45, 67

CPU Central Processing Unit. 12, 14, 21, 26–29, 54, 60, 63, 72, 75

DCS Distributed Control System. 6

DMA Direct Memory Access. 14, 54

DTB Device Tree Blob. vii, 26, 27

DTS Device Tree Source. 26

GCC GNU Compiler Collection. 20

GDB GNU Debugger. 21

GPL General Public License. 51

GUI Graphic User Interface. 9, 13, 40

HDMI High-Definition Multimedia Interface. 25, 68

HMI Human Machine Interface. 9–13

I/O Input/Output. 26, 27

ICMP Internet Control Message Protocol. 78, 79

ICS Industrial Control System. iv, 2, 6–8, 10, 12, 13, 15, 18, 47

IoT Internet of Things. 3, 4, 35

IP Internet Protocol. 25, 78

IPC InterProcess Communication. v, viii, 75, 76

IRQ Interrupt Request. 60

ISO/OSI International Standards Organization Open Systems Interconnection. 78

ISR Interrupt Service Routine. 55

JTAG Joint Test Action Group. 21, 68

KGDB Kernel GNU Debugger. 22

LCD Liquid Crystal Display. 68

LED Light Emitting Diode. 9

LVDS Low-voltage Differential Signaling. 68

MCS Mixed Criticality system. 13, 14

MMU Memory Management Unit. 14

MPEG Moving Picture Experts Group. 81

MPSoC Multiprocessor System-on-chip. iv, 13, 14

OS Operating System. 15, 16, 18, 35

OTG On-The-Go. 68

PC Personal Computer. 8–10, 16, 18, 20

PID Process Identifier. 85

PLC Programmable Logic Controller. iv, 1, 6, 8–12

PMU Performance Monitoring Unit. v, 64

POSIX Portable Operating System Interface for Unix. 51, 52, 62

RT Real-Time. 75

RTOS Real Time Operatin System. 51

SABRE Smart Application Blueprint for Rapid Engineering. 66–68

SATA Serial Advanced Technology Attachment. 68

SBC Single Board Computer. 2, 9, 15

SCADA Supervisory Control and Data Acquisition. 6

SD Secure Digital. 24, 25, 31, 33, 68, 69

SDK Software Development Kit. 44, 45

SI International System of Units. 36

SoC System on chip. 13, 14, 32–34, 36, 63, 64, 66

SRAM Static Random Access Memory. 25

TCP Transmission Control Protocol. 25, 80

TLB Translation Lookaside Buffer. 14

UART Universal Asynchronous Receiver-Transmitter. 69, 70

UDP User Datagram Protocol. 80

USB Universal Serial Bus. 25, 31, 68–70

YP Yocto Project. 2, 19, 34–36, 39