Università degli studi di Padova
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Integration of an IEEE802.15.4g compliant transceiver into the Linux-based AMBER platform

Laureando
Matteo Sartori

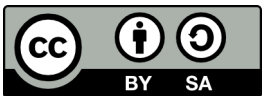
Relatore
Michele Moro

Correlatore
Matteo Petracca
Scuola Superiore Sant'Anna, Pisa

Anno accademico 2015-2016

abstract

Nowadays the world is continuously discovering new strategies and methods to effectively organize the enormous quantity of information that has become accessible to us. Such scenario is very well depicted by a thematic we can see spreading in a lot of different scopes which could entail a deep transformation of our society in the forthcoming future. Internet of Things is considered by many professionals and academics to be the next important breakthrough technology. Radio technologies, protocols development, low-complexity and low-power devices, embedded operating systems, software engineering and security considerations are the main important problems that companies and research laboratories are facing in order for the IoT to become a reality. Texas Instruments, which owns the broadest wireless connectivity portfolio, has developed an entire line of transceiver chips implementing low level protocols, like the popular *IEEE802.15.4g*, thus giving the fundamental communication system upon which to construct modern IoT networks.
As the development process is concerned, it is very important to understand that the right combination of hardware and software tools is necessary in order to solve cutting edge problems in an efficient way. The Amber open-hardware platform is designed to be the perfect tool to deal with IoT embedded applications and lends itself to the development of sensors, protocols, security mechanisms, gateways and filtering questions. Also, the recent progress in facilitating the use of the Linux kernel in the embedded world, and the wide adoption of this operating system as the deployment platform for third-party software business, suggests a possible favorable future for open source and community developed operating systems. In this work we illustrate a whole stack of protocols and software architecture tipically involved in modern IoT systems and report the experience of integrating a capable transceiver device from Texas Instruments into the Amber embedded platform running the Linux operating system.



Contents

List of Figures

Chapter 1

Introduction

1.1 Information age

The *Information Age* is by all means an incredible change impacting on all levels of human society. As for other industrial revolutions in the past, the particular technology which was developed, for example in textile manufacturing, metallurgy, steam power, chemicals, firstly determined improvements in the production methods but in the longer run, entailed a deep transformation concerning economics, politics, society, culture, education etc. The one that contributed to foundations of information age is called *Digital Revolution*.

In this days the world is shifting towards a knowledge based society in which the capability of extrapolating knowledge from flat information has become the main objective of whole economies, and this trend is having large implications also in other fields like, most notably social interactions. Also the fact that a child can access an incredible amount of information only with just a click is an indication of how much different will be forthcoming generations.

For the information age is more difficult to determine exactly what was the principal technology that initiated this evolutionary process, as a lot of different fields had had an incredible development in the 20th century, even though the conception and first studies were just started few years before. We can illustrate and summarize the technological progress of this period, if at all possible, by considering the fact that the main objective of academics and high tech laboratories was to build and efficiently operate new kind of machines. They all wanted systems being able to execute a huge number of small operations, each completely independent and with a well defined behaviour, which could eventually represent some complex automation in the whole, carefully programmed to be very precise and mathematically correct and certainly, as fast as possible. This intent actually originated two different disciplines: digital electronics and computing. An incredible powerful pair where the one cannot exist without the other. Nowadays a widely known expression refers to them as hardware-software symbiosis. Whether the study of electrical digital circuits or the more abstract algorithmic approach, the two disciplines combined have shown the ability to produce amazing complexity systems like

modern web services, operative systems or networking infrastructures.

Another important component in the evolving information technology and a pivotal player for being a key catalyst was Internet. A large globally extended and publicly accessible system which lets machines communicate in a consistent and efficient manner, without requiring a specific underlying machine architecture or operating system due to the employment of well defined suite of protocols. An important historical reason for considering Internet a change in computer networks was the ability to interconnect different local or independent networks without requiring particular assumptions on the inner workings. This simple idea, implemented through the use of the famous TCP/IP suite, was the key in driving the incredibly wide adoption of this technology worldwide. Further, being able to make machines interact with each others, in such a vast scale, was an opening for new concepts. The value a computing machine can have is far amplified by the fact of being simultaneously used for specific tasks and for leveraging on others' capabilities. This introduced the powerful concept of distributed computing or the more recent and business oriented cloud computing.

Without the capability of easily browse information content, Internet by itself would have meant nothing special. The important tool that served in increasing the use of networking technology was the World Wide Web where the content is efficiently interlinked and the user can "surf" an incredible amount of information with only a click. This gave birth to the concept of web services. The user with his/her computer can reach very quickly a lot of different everyday life supporting utilities, for example home-banking, global e-commerce marketplaces (eBay), searching services (Google), video streaming (Netflix), social networking (Facebook) and a lot of ad-hoc web content like Q&A websites, wiki's and forums. New economies, new ways of social interaction, different collaborative communities (Wikipedia, open source) have been emerging since the inception of this new medium. The society proved to be willing to this transformation and new trends show we have just now entered the Information Age, that we should expect to see a lot of innovation happening in the forthcoming future and the world we see today will probably have changed considerably.

1.2 Internet of Things

The concept of computer networking, even though started from locally deployed networks, through the use of the revolutionary idea of packet switching communication, quickly reached its full potential. The principal inclination of automated networks has always been to reach a range as wide as possible, where each machine can access the functionality of another system working wherever on earth. This concept, known as network effect in economy and business, has dictated the evolution of Internet. Another important implication of the fact that society has entered the information age, is the increasing intention of finding business opportunities more and more related on the exchange of information. This is supported by observing that a lot of economics activities has discovered how much value can be obtained from information technologies, for example mobile smart phones are attracting partners over and over again.

A famous trend that is nowadays exploding is the Internet of Things. This technology benefits from advancements in computing power, electronics miniaturization, and network interconnections to offer new capabilities not previously possible, and has the simple target of connecting every kind of automation systems, sensor or actuator to the Internet, although not in a short time at all. The large scale implementation of IoT devices promises to further transform many aspects of the way we live. For consumers, new IoT products like Internet-enabled appliances, home automation components, and energy management devices are moving us toward a vision of the smart home, offering more security and energy-efficiency. Other personal IoT devices like wearable fitness and health monitoring devices and network-enabled medical devices are transforming the way healthcare services are delivered. This technology promises to be beneficial for people with disabilities and the elderly, enabling improved levels of independence and quality of life at a reasonable cost. IoT systems like networked vehicles, intelligent traffic systems, and sensors embedded in roads and bridges move us closer to the idea of smart cities, which help minimize congestion and energy consumption. IoT technology offers the possibility to transform agriculture, industry, and energy production and distribution by increasing the availability of information along the value chain of production using networked sensors. However, such radical change doesn't come for free. Deploying smart object networks can be very challenging due to the engaging of a lot of different technology fields.

1.3 Developing IoT Systems

In this work we present a set of thematics related to IoT, particularly on the development process that is necessary in order to create a network of smart objects. Recent progresses in radio technologies, networking protocols, embedded platforms and operating systems are covered as an high level presentation of new topics that are recently arising in the industry, and a more detailed discussion is provided only on software development in the Linux ecosystem.

As illustrated in Chapter 2, there is no a single valuable framework which can be used for developing IoT systems. There are a lot of problematics, from different fields, that must be well understood, like security, privacy, interoperability and standards. It also obscure to clearly determine properties of objects we want to connect in a smart manner, so the first step is to carefully decide which kind of network our system belong to, otherwise it could be difficult to correctly choose the communication protocols, the development hardware and the operating system to employ. This is why in the Chapter 2 a wide scenario on recent IoT activities and considerations is proposed. Once it is clear which kind of particular networked system we are facing, from the economic and social point of view, we can consider to dive into more technical reasoning. As the Internet of Things is itself born from recent progresses in radio technology is probable that our system will make use of such modern techniques. Important considerations regard the frequency bandwidth that is more appropriate, the modulation scheme for being able to reach a certain throughput and the suite of network protocols that better matches the

characteristics of our future installation. Another important point is the development system to use. Given the embedded nature of common IoT devices, being those sensors or small actuator, is a key factor to use the right development environment. Such hardware has to provide the widest set of configuration options and ports for easily evaluate different digital protocols, power consumptions and for test and debugging purpose as well. The choice of the operating system is also a decisive point. Given the huge amount of abstractions and the incredible stability an operating system can provide, is important to choose the one that better suits our needs. We can decide among more real time oriented systems, closed source or community developed and this decision can often entails subtle considerations like: Which is the most comfortable to develop software with? Which is easier to maintain and to evolve? Is this piece of code used by a wide community of developers around the world? Other points regard supported standards, compatible hardware for which a driver has already been developed and configuration capabilities. Further, it is worth noting that IoT systems' particular characteristics can bring on other relevant problematics like low-level, difficult to write and test code, software upgrade features and maybe the most important: software security considerations. Finally, an important cross thematic that has to be taken into consideration at all levels of the design is the fact that most devices we want to use have to be low-complexity and low-powered. This can have a lot of implications because, even if there are hardware systems with a lot of power related capabilities, for example system-on-chip where nearly all integrated controllers have a low power functioning state, it is hard to find the right software support and integrating such features can be an hassle.

Throughout the document we describe various solutions by reasoning on a certain example of an IoT network. This is to draw attention to a specific practical case study, given that IoT is only a conceptual word for describing a technological trend. We chose smart sensor utility networks, often referred to as SUN, which is receiving enough attention today and is an actual theme when it comes to talk about other famous "smart" trends, like smart city, smart grid and smart agriculture. Special protocols have been developed and are also rather considered in the open source community.

In this thesis we explore the Linux open source kernel and show how this renowned operating system is suited for this matter by integrating a full-featured transceiver device from Texas Instruments into an open hardware platform, known as AMBER, equipped with a Freescale i.MX6 system on chip. This document is organized as follows. The Chapter 2 is dedicated to an overview of IoT, how the actual industry is perceiving it and some design guidelines that could be useful to better approach the development of smart object networks. Chapter 3 explains the IEEE 802.15.4 protocol and why we have chosen it for our purpose. Chapter 4 gives an overview of the Linux operating system and some of its internal subsystems, such as the netlink network interface and Chapter 5 presents the development hardware we utilized to conduct our experiments. The following section, Chapter 6 indeed, explains how we dealt with the driver development, configuration of host and target machines and some experimental outcomes. The intended audience is anybody who wants to learn a bit of IoT concepts, how to manage an embedded operating system and how to write a driver for the Linux network subsystem.

Chapter 2

IoT Background

The heart of this new technology is embodied in a wide spectrum of networked products, systems, and sensors, which take advantage of proceedings in computing power, electronics miniaturization, and network interconnections to offer new capabilities not previously possible. The history of telecommunication systems, which has largely been developing for the last two centuries, has passed through different fundamentals step, for example the invention of the telegraph, the use of electromagnetic waves to convey information and the construction of complex packet-switched networks. The latter, which dates back to the '60s, can be considered the most important because demonstrated the possibility to achieve complex communications nearly without human intervention, so establishing the right scenario for the birth of Internet. The concept of building more and more automated systems led to the development of whole suite of protocols that eventually became capable of running modern life-changing applications like home-banking, cloud computing and an always increasing set of browsable web content. The most incredible fact is that the effort in designing a general system, has shown the incredible general purpose nature of the Internet architecture, which does not place inherent limitations on the applications or services that can make use of the technology, thus making it possible to imagine an hyper connected world.

This chapter tries to present the concept of Internet of Things as a thematic that has important technical questions, difficult industrial implications and issues concerning society in a broader sense. We start from a short historical overview with important definitions invented by famous institutions, then illustrate some design guidelines which can assist every IoT related project and finally some issues concerning security, privacy, interoperability and standardizations, which can also help in approaching and dealing with this new technological trend.

2.1 Origins

The term Internet of Things was first used in 1999 by British technology pioneer Kevin Ashton to describe a system in which objects in the physical world could be connected to the Internet. Ashton coined the term to illustrate the power of connecting radio-

frequency identification tags used in corporate supply chains to the Internet in order to count and track goods without the need for human intervention. The key idea was to imagine the actual Internet infrastructure augmented with "sensing" capabilities towards the physical world, thus opening an astonishing new scenario of applications and possibilities where the concepts of feedback and control are pushed beyond their limits. In the 1990s, advances in wireless technology allowed machine-to-machine (M2M) enterprise and industrial solutions for equipment monitoring and operation to become widespread. Many of these early M2M solutions, however, were based on closed purpose-built networks and proprietary or industry-specific standards, rather than on Internet Protocol IP-based networks and Internet standards. Using IP to connect devices other than computers to the Internet is not a new idea. The first Internet device, an IP enabled toaster that could be turned on and off over the Internet, was featured at an Internet conference in 1990. If the idea of connecting objects to each other and to the Internet is not new, it is reasonable to ask, "Why is the Internet of Things a newly popular topic today?" From a broad perspective, the confluence of several technologies and market trends is making it possible to interconnect more and smaller devices cheaply and easily.

It is also important to understand that the environment in which IoT is growing is not self contained. Instead, it is accompanied by progresses in other, even related, disciplines and a number of technologies have already been established, especially in the industry. The following is a list of macro trends that are becoming more and more consolidated and will certainly favour the diffusion of IoT.

Ubiquitous connectivity Low-cost, high-speed, pervasive network connectivity, especially through licensed and unlicensed wireless services and technology, makes almost everything "connectable".

Widespread adoption of IP based networking IP has become the dominant global standard for networking, providing a well defined and widely implemented platform of software and tools that can be incorporated into a broad range of devices easily and inexpensively.

Computing economics Driven by industry investment in research, development, and manufacturing, Moores law continues to deliver greater computing power at lower price points and lower power consumption.

Miniaturization Manufacturing advances allow cutting-edge computing and communications technology o be incorporated into very small objects. Coupled with greater computing economics, this has fueled the advancement of small and inexpensive sensor devices, which drive many IoT applications.

Advances in Data Analytics New algorithms and rapid increases in computing power, data storage, and cloud services enable the aggregation, correlation, and analysis of vast quantities of data, these large and dynamic datasets provide new opportunities for extracting information and knowledge.

Rise of Cloud Computing Which leverages remote, networked computing resources to process, manage, and store data, allows small and distributed devices to interact with powerful back-end analytic and control capabilities.

From this perspective, the IoT represents the convergence of a variety of computing and connectivity trends that have been evolving for many decades. At present, a wide range of industry sectors including automotive, healthcare, manufacturing, home and consumer electronics, and well beyond, are considering the potential for incorporating IoT technology into their products, services, and operations. The world is definitely ready to accept these new ideas, from an economical point of view as well as cultural.

2.2 Applications

As we are going to see in following sections, another important aspect of IoT is the difficulty in attributing meaning to involved concepts. In fact, when it comes to reason about the applicability of such technology, it is easy to understand that a well defined horizon doesn't exist and the scope of this thematic will probably change considerably. Improvements the technology is undergoing might make us capable of connecting more and more devices in the future, and so it is actually impossible to force any bounds on what we are referring to with the term "Things". These days, the industry is studying the scope of this new technology trying to better understand how the phenomenon behaves and has produced an assortment of various sectors which the IoT will probably have a considerable impact on.

Smart home Internet-enabled appliances, home automation components, and energy management devices. Offering more security and energy-efficiency.

Healthcare services Personal IoT devices like wearable fitness and health monitoring devices and network-enabled medical devices. This technology promises to be beneficial for people with disabilities and the elderly, enabling improved levels of independence and quality of life at a reasonable cost.

Smart cities Systems like networked vehicles, intelligent traffic systems, and sensors embedded in roads and bridges. Minimize congestion and energy consumption.

Agriculture, industry and energy production Increased availability of information along the value chain of production using networked sensors.

It is widely known that industrial and economical systems can shape a lot more than the simple market. The following is a list of locations and spaces where we can expect to see huge transformations caused by advancements in previous industrial sectors.

- **Human devices** attached or inside the human body
- **Home buildings** where people live

- **Retail environments** spaces where consumers engage in commerce
- **Offices** spaces where knowledge workers work
- **Factories** standardized production environments
- **Worksites** custom production environments
- **Vehicles** systems inside moving vehicles
- **Cities** urban environments
- **Outside** between urban environments and outside

2.3 Projections

The incredible amount of popularization material regarding IoT that has been published in all innovation fields has led a lot of famous companies and financial corporations to conduct statistical projections in order to have a measure of how the market will behave in the future. The following are some quotes:

```
Cisco, for example, projects more than 24 billion Internet connected
objects by 2019.

Morgan Stanley, however, projects 75 billion networked devices by 2020.

Huawei forecasts 100 billion IoT connections by 2025.

McKinsey Global Institute suggests that the financial impact of IoT on
the global economy may be as much as $3.9 to $11.1 trillion by 2025.
```

IoT as a revolutionary fully interconnected "smart" world of progress, efficiency, and opportunity, with the potential for adding billions in value to industry and the global economy. Attention-grabbing headlines about the hacking of Internet-connected automobiles, surveillance concerns stemming from voice recognition features in "smart" TVs, and privacy fears stemming from the potential misuse of IoT data have captured public attention. It is very important to understand that such considerations can potentially generate an everlasting debate about pros and cons of the modern Internet and given the immense impact it might have on industry and society it could become a complex topic to understand.

2.4 Definitions

There are different definitions proposed by a multitude of societies, like Internet Engineering Task Force *IETF*, International Telecommunication Union *ITU* and *IEEE*.

This reveals an important point in understanding the concept of IoT: there are different perspectives to be factored into discussions, in order to better understand the whole potential of such technology. The following is simply a list extracted from various documents regarding Internet of Things.

- Internet Architecture Board *IAB* begins RFC 7452 *Architectural Considerations in Smart Object Networking* with this description: The term "Internet of Things" denotes a trend where a large number of embedded devices employ communication services offered by the Internet protocols. Many of these devices, often called "smart objects", are not directly operated by humans, but exist as components in buildings or vehicles, or are spread out in the environment.

- Internet Engineering Task Force *IETF* introduces the term "smart object networking" as a commonly used alias for the Internet of Things. In this context, "smart objects" are devices that typically have significant constraints, such as limited power, memory, and processing resources, or bandwidth. Work in the IETF is organized around specific requirements to achieve network interoperability between several types of smart objects.

- International Telecommunication Union *ITU*, *Overview of the Internet of Things*, discusses the concept of interconnectivity, but does not specifically tie the IoT to the Internet: A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies. **Note1:** Through the exploitation of identification, data capture, processing and communication capabilities, the IoT makes full use of things to offer services to all kinds of applications, whilst ensuring that security and privacy requirements are fulfilled. **Note2:** From a broader perspective, the IoT can be perceived as a vision with technological and societal implications.

- *IEEE Communications Magazine* links the IoT back to cloud services: The Internet of Things is a framework in which all things have a representation and a presence in the Internet. More specifically, the Internet of Things aims at offering new applications and services bridging the physical and virtual worlds, in which Machine-to-Machine communications represents the baseline communication that enables the interactions between "things" and applications in the cloud.

Different perspectives exist because there are many different ways for imaging an hyper connected world where each scenario being equally likely to happen and nobody knows which definition will be more appropriate. However, with the help of radio technology and the possibility to embed relatively small machines in almost every aspect of our life, it is feasible that one day there will be a single well structured information system with the capability of conditioning our life, society and the way we do business.

2.5 Design Guidelines

In this section we concentrate on what kind of difficulties may arise in developing IoT systems, from a more technical point of view. A designer wishing to build a modern style network of small smart objects should consider in advance that such matter is fully made up of issues each pertaining to very different disciplines, like hardware assembled, software design and developing, network topologies and architectures mixing general purpose protocols with ad hoc communication methods. To the extent of presenting such subtleties, in this part we focus on a general overview and a more abstract description of network topics, leaving a more detailed presentation about protocol peculiarities and software developing concerns to following chapters.

As previously described, establishing a plain to see definition for Internet of Things is a difficult task. For this reason, while approaching the design of a system, is even more important to put effort in specifically understand what kind of objects are being connected and what protocols to employ. Reasoning about devices capabilities and design goals should be given a lot of attention, because choosing the wrong hardware component can enormously affect the final outcome. Typical parameters like bandwidth, memory, instructions per second *IPS*, size, energy consumption all participate and, given the attention to low power applications, is not even clear which devices are able to run Internet protocols at all. Another important point regards the network effect that originate. Interconnecting smart objects enables new use cases and products and increasing the number of products that use Internet Protocol Suite on smaller and smaller devices offers the ability to process, visualize, and gain insight from the collected sensor data. Thus the best option is to account for this particular phenomenon from primal design choices, in order to fully govern and take advantage from this effect that will probably expand beyond our imagination.

The following is a short discussion on important topics about developing smart networks. It is largely based on the IAB document *Architectural Considerations in Smart Object Networking RFC 7452*.

2.5.1 Device Characteristics

As announced in Chapter 1 in order to match our considerations with a meaningful practical example, we present our reasoning by analyzing smart metering utility networks. For such systems to express their full power, is probable that a lot of small devices are scattered in a particular territory, as an example agricultural production systems might need to monitor environmental parameters in immense areas. So the first issue is about the radio technology which is able to cover the whole space, with considerations on which radio band is the most effective to use, which modulations scheme will match the particular conditions and how many devices to deploy. Second and nonetheless important is to answer to the following question: “How long is the device designed to operate?” If the particular application doesn’t provide devices with continuous electrical supply, is very likely to have constraints on battery life. This particular intent is a quite hot topic today, because we can see a lot of hardware manufacturers marketing on devices current

consumption characteristics, but this concept is not to be taken for granted. If the particular device can alternate between sleep state and active state, is easy to understand that in order to properly handle this switching configuration is necessary to have the right software support, which could not be an easy task as developing and integrating it in the whole design process may entail significant efforts. A collateral issue regards the maintenance policy of the deployed system. If it is important to easily access devices, for example to replace batteries, is also a first to understand how this affect the security model. Strengthening the system against tampering attacks must be carefully handled. Also, if a particular device is able to act as an actuator is necessary to properly model the network as well, because in the opposite case with only sensing devices a collector approach would serve better. Finally, is important to understand if the deployed system will be connected to Internet, because making the communications to enter and to exit the local area network could also be difficult and implicate particular network topologies.

2.5.2 Communication Models

One of the biggest design challenges in networked product is the right communication model to employ. This can have important implications on overall performances, on expanding capabilities and even on security concerns. Fortunately, a full set of well crafted and long lasting design patterns are available when there are difficulties every designer encounters. This is especially true for network technologies that had to fit in a wide application contexts at the very beginning, like the Internet itself. The IAB selects the following for being particularly suited for smart object networks.

Device-to-device Communications

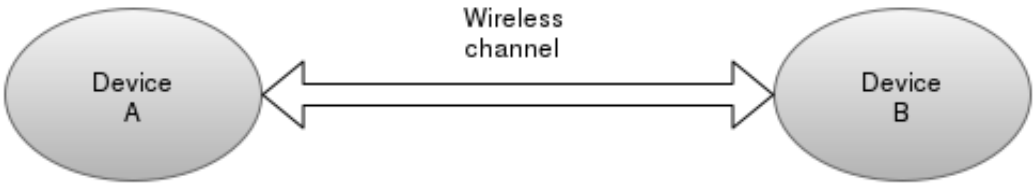


Figure 2.1: Direct communication model

The most simple network scenario, as we see in Figure 2.1, is to have one device speaking directly with another, for example we can consider a light switch that talks to a light bulb in order to activate/deactivate it. Even though such implementations can avoid complex network related topics, like addressing modes, link settings and routing mechanisms, having a device autonomously talk with another peer can be very difficult to design. Paring mechanisms, like that of *Bluetooth*, must be very adaptable to different applicative contexts and implementing intelligent protocol logic may entail the need of powerful computing devices and difficult-to-program software architectures. Achieving

interoperability among different manufacturers can be another trouble, because the standardization process can be remarkably durable and, in fact, choosing and contributing to an open standard can be a good option.

Device-to-cloud Communications

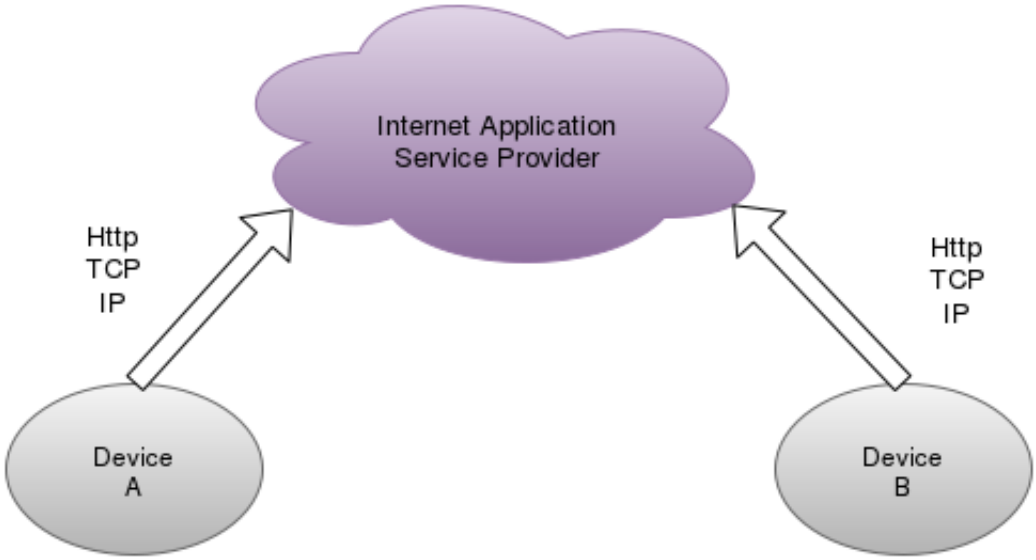


Figure 2.2: Device communicating with cloud services

As an alternative we consider now Figure 2.2, it is feasible to connect devices to the Internet, which can relax some design issues for the network being intrinsically a protocol translator, so questions like devices interoperability can be very well handled. But it is worth noting that a device able to communicate over IP has to support a whole set of protocols and this can be unacceptable, for example attaching a *IEEE 802.11ay* capable device to the light bulb can be considered exaggerated (frequency band: 60 GHz, transmission rate: 20-40 Gbps and range distance: 500 meters).

Device-to-gateway Model

Instead of being connected to the Internet directly, it is possible to place a gateway system between devices and the more global network. The gateway can act as a converter between the IP and a much more simpler network. We can see and example in Figure 2.3. Every device, for example simple sensors producing data in a low rate context, can communicate only with the gateway, thus implementing the essential part of communication and leaving the complex network translations to the gateway. This model is very appropriate for smart sensor networks, because devices can be very simple, low-power and long lasting.

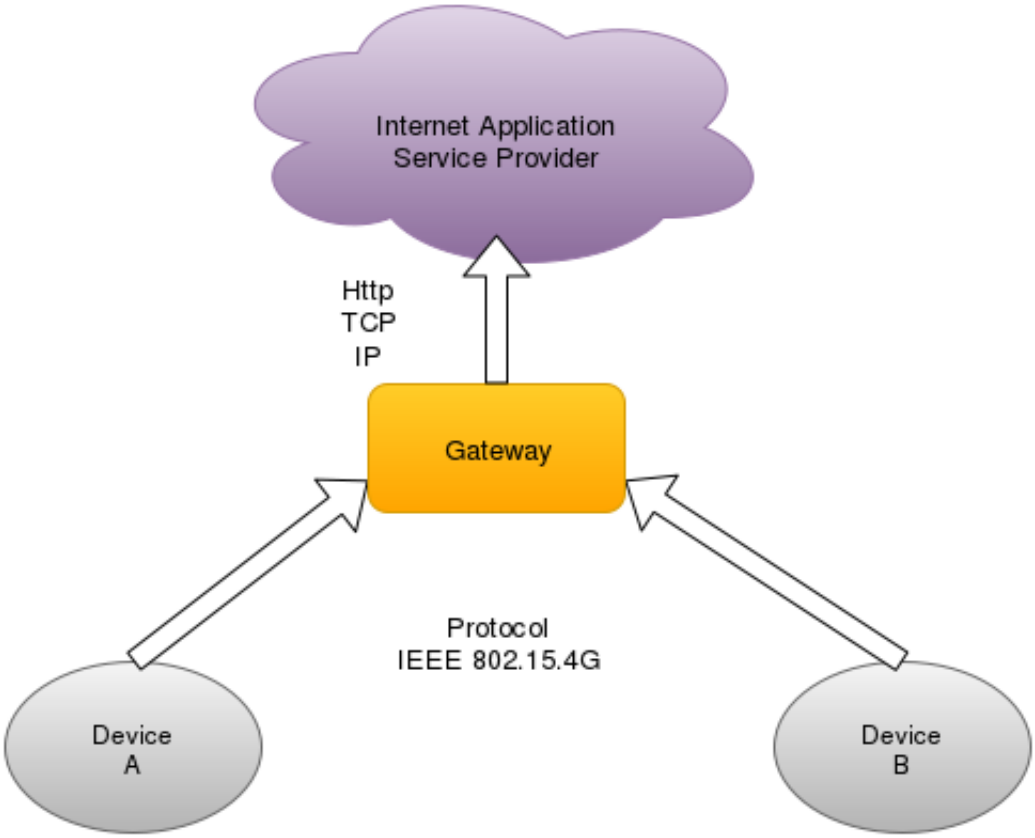


Figure 2.3: Communications passes through the gateway

Back-end Data-sharing Model

The last communication pattern has been devised for better handling data sharing towards third parties services. Data collected from a particular smart sensor installation is not always consumed by a single application service over the Internet, instead is possible to have different more general data collectors which has access to the sensor network. In this case a back-end architecture is more suitable. The "owner" Internet application provider must implement two services: the first is responsible to handle all data gathering and provide this information to the end user, the other is to implement the same functionality to be consumed by other Internet servers. This last capability may entail a different architecture with different protocols and interfaces.

2.5.3 Reuse Internet Protocols

Because building very small, battery-powered devices is challenging, it may be difficult to resist the temptation to build solutions tailored to specific applications, or even to redesign networks from scratch to suit a particular application, in order to gain control

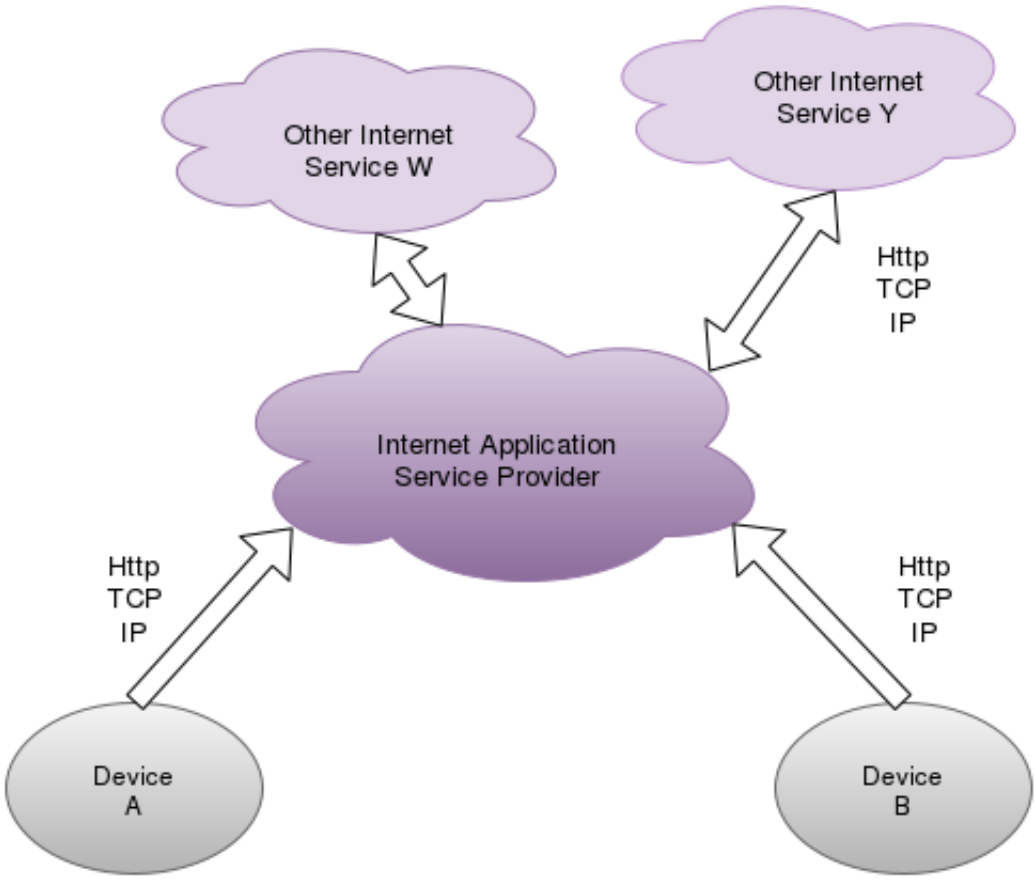


Figure 2.4: The first Internet service provider might be configured to talk to other services

over the whole system behaviour. An alternative to redesigning protocols is to consider the success of the Internet, a set of consensus-based, open and transparent standards which evolved considerably from the beginning and can allegedly last for very long time. The general architecture of Internet has proved to be very flexible and the fact that the world biggest network is already deployed and functioning by the use of IP should convince designer and business to take part in its space of connecting capabilities. But, as the Internet has evolved, is also true that while certain protocols and protocol extensions have become the norm, others have become difficult to use in all circumstances.

Taking into account all these considerations is particularly important for smart objects, as there is often a desire to employ specific features. For instance, from a pure protocol-specification perspective, some transport protocols may be more desirable than others. However, the most important goal is to design applications so that the participating devices can easily interact with multiple other applications and the effort to extend the communication potential should be kept to a minimum.

2.5.4 Design for a Change

Given the fact that a technology can be superseded in few months, embracing rapid innovation in the design process should be a priority. A relevant notion is that of *designing for variation in outcome*. With this goal in mind the designer is able to build systems that can be relatively easy to alter and adapt to particular needs. For example, having a solid software update mechanism is needed not only for dealing with the changing Internet communication environment and for interoperability improvements but also for adding new features and for fixing security bugs.
Common indications for achieving this purpose are breaking complex systems into modular parts, designing for choice to permit the different players to express their preferences.

2.6 Issues

When the world faces the coming of a new technology it is important to understand how it could impact with changes that will probably be introduced, as these can affect enormously how the society interacts with the environment in which this transformation occurs. A worth noting example of a groundbreaking innovation we have seen in the recent past is the Internet itself. An important amount of business activities, how people interact with each other and academic research have been modified considerably. IoT may force a shift in thinking if the most common interaction with the Internet, and the data derived and exchanged from that interaction, comes from passive engagement with connected objects in the broader environment. The potential realization of this outcome, an **Hyper connected world**, is a testament to the general purpose nature of the Internet architecture, which does not constraint application or service data exchange on the properties of the underlying infrastructure, at least from a general point of view.

This section wants to illustrate that a big innovation may entail also significant problems the society will have to deal with, and the deeper the transformation the subtler and the more harmful the impact can be. Professionals, companies and countries have to acquire the necessary knowledge in order to accurately reason and to properly maintain an ever evolving society, which become every day more and more technology based. The IoT designer should also take these considerations into account.

2.6.1 Security

An important popular topic that is invading all information technology related disciplines is that of security. It seems an intrinsic property of a system that when the inner working complexity reaches a big level, it is not only the development process that is affected but also the possibility to include obscure security vulnerabilities terribly grows. Such thematic is more important today than it was some years ago and even if a lot of countermeasures has been developed and wide deployed in the world, like cryptographic systems and strengthen authentication protocols, the complexity which our daily use systems depend on suggests that the battle with cyber attacks is not ended.

As we increasingly connect devices to the Internet, new opportunities to exploit potential security vulnerabilities grow. Poorly secured IoT devices could serve as entry points for cyberattack by allowing malicious individuals to re-program a device or cause it to malfunction. Poorly designed devices can expose user data to theft by leaving data streams inadequately protected. Failing or malfunctioning devices also can create security vulnerabilities. Along with potential security design deficiencies, the sheer increase in the number and nature of IoT devices could increase the opportunities of attack. When coupled with the highly interconnected nature of IoT devices, every poorly secured device that is connected online potentially affects the security and resilience of the Internet globally, not just locally. To complicate matters, our ability to be operative in our daily activities without using devices or systems that are Internet-enabled is likely to decrease in a hyper connected world. This increasing level of dependence on IoT devices and the Internet services they interact with also increases the pathways for criminals to gain access to devices.

IoT devices tend to differ from traditional computers and computing devices in important ways that challenge security:

- Many Internet of Things devices, such as sensors and consumer items, are designed to be deployed at a massive scale that is orders of magnitude beyond that of traditional Internet-connected devices. As a result, the potential quantity of interconnected links between these devices is unprecedented. Further, many of these devices will be able to establish links and communicate with other devices on their own in an unpredictable and dynamic fashion.

- Many IoT deployments will consist of collections of identical or near identical devices. This homogeneity magnifies the potential impact of any single security vulnerability by the sheer number of devices that all have the same characteristics.

- Many Internet of Things devices will be deployed with an anticipated service life many years longer than is typically associated with high-tech equipment. Further, these devices might be deployed in circumstances that make it difficult or impossible to reconfigure or upgrade them, or these devices might outlive the company that created them, leaving orphaned devices with no means of long-term support. These scenarios illustrate that security mechanisms that are adequate at deployment might not be adequate for the full lifespan of the device as security threats evolve. As such, this may create vulnerabilities that could persist for a long time. This is in contrast to the paradigm of traditional computer systems that are normally upgraded with operating system software updates throughout the life of the computer to address security threats.

- Some IoT devices are likely to be deployed in places where physical security is difficult or impossible to achieve. Attackers may have direct physical access to IoT devices. Anti-tamper features and other design innovations will need to be considered to ensure security.

- Some IoT devices, like many environmental sensors, are designed to be unobtrusively embedded in the environment, where a user does not actively notice the device nor monitor its operating status. Additionally, devices may have no clear way to alert the user when a security problem arises, making it difficult for a user to know that a security breach of an IoT device has occurred. A security breach might persist for a long time before being noticed and corrected if correction or mitigation is even possible or practical. Similarly, the user might not be aware that a sensor exists in her surroundings, potentially allowing a security breach to persist for long periods without detection.

In order to oppose this increasing security weaknesses some guidelines has to be followed. We propose here some indications that *The Internet Society* has carefully organized as follows. These are not solutions, instead some questions that can serve in understanding how to reason about the complex thematic of security.

Good design practices What are the set of best practices for engineers and developers to use to design IoT devices to make them more secure? How do lessons learned from Internet of Things security problems get captured and conveyed to development communities to improve future generations of devices? What training and educational resources are available to teach engineers and developers more secure IoT design?

Standards and metrics What is the role of technical and operational standards for the development and deployment of secure, well-behaving IoT devices? How do we effectively identify and measure characteristics of IoT device security? How do we measure the effectiveness of Internet of Things security initiatives and countermeasures? How do we ensure security best practices are implemented?

Data confidentiality, authentication and access control What is the optimal role of data encryption with respect to IoT devices? Which encryption and authentication technologies could be adapted for the Internet of Things, and how could they be implemented within an IoT devices constraints on cost, size, and processing speed? What are the foreseeable management issues that must be addressed as a result of IoT-scale cryptography? Are the end-to-end processes adequately secure and simple enough for typical consumers to use?

Field upgradeability With an extended service life expected for many IoT devices, should devices be designed for maintainability and upgradeability in the field to adapt to evolving security threats? New software and parameter settings could be installed in a fielded IoT device by a centralized security management system if each device had an integrated device management agent. But management systems add cost and complexity, could other approaches to upgrading device software be more compatible with widespread use of IoT devices? Are there any classes of IoT devices that are low-risk and therefore dont warrant these kinds of features? In general, are the user interfaces IoT devices expose (usually intentionally minimal)

being properly scrutinized with consideration for device management (by anyone, including the user)?

Regulation Should device manufacturers be penalized for selling software or hardware with known or unknown security flaws?

Collaborative model Has emerged as an effective approach among industry, governments, and public authorities to help secure the Internet and cyberspace, including the Internet of Things. This model includes a range of practices and tools including bidirectional voluntary information sharing, effective enforcement tools, incident preparedness and cyber exercises, awareness raising and training, agreement on international norms of behavior, and development and recognition of international standards and practices. Continued work is needed to evolve collaborative and shared risk management-based approaches that are well suited to the scale and complexity of IoT device security challenges of the future.

2.6.2 Privacy Considerations

Respect for privacy rights and expectations is integral to ensuring trust in the Internet, and it also impacts the ability of individuals to speak, connect, and choose in meaningful ways. IoT often refers to a large network of sensor-enabled devices designed to collect data about their environment, which frequently includes data related to people. This data presumably provides a benefit to the devices owner, but frequently to the devices manufacturer or supplier as well. IoT data collection and use becomes a privacy consideration when the individuals who are observed by IoT devices have different privacy expectations regarding the scope and use of that data than those of the data collector. When individual data streams are combined or correlated, often a more invasive digital portrait is painted of the individual than can be realized from an individual IoT data stream. This data-aggregation effect can be particularly potent with respect to IoT devices because many produce additional metadata like time stamps and geolocation information, which adds even more specificity about the user. Generally, privacy concerns are amplified by the way in which the Internet of Things expands the feasibility and reach of surveillance and tracking. Characteristics of IoT devices and the ways they are used redefine the debate about privacy issues, because they dramatically change how personal data is collected, analyzed, used, and protected.

2.6.3 Interoperability and Standards

In the traditional Internet, interoperability is the most basic core value, the first requirement of Internet connectivity is that “connected” systems be able to “talk” by the usage of protocols. In practicality, interoperability is more complex. Interoperability among IoT devices and systems happens in varying degrees at different layers within the communications protocol stack between the devices. Furthermore, full interoperability across every aspect of a technical product is not always feasible, necessary, or desirable. Beyond the technical aspects, interoperability has significant influence on the potential

economic impact of IoT. Well-functioning and well-defined device interoperability can encourage innovation and provide efficiencies for IoT device manufacturers, increasing the overall economic value of the market. Also, interoperability is fundamentally valuable from the perspective of both the individual consumer and organizational user of these devices. It facilitates the ability to choose devices with the best features at the best price and integrate them to make them work together.

Chapter 3

Protocols Description

As outlined in previous sections, an important part of the design process in order to implement smart objects networks is to carefully analyze the protocol suite that best fit a particular use. In this chapter we follow our application example of smart sensor networks and expand our considerations in order to choose the right protocol stack. Instead of implementing a specific one for our needs, we decided to concentrate on the selection of an open and wide accepted standard. The famous *Wireless Personal Area Networks WPAN IEEE 802.15* has recently addressed the side use case of smart metering networks. More specifically the committee has completed different projects that can be well matched with our application. The standardization process initially started with the document *802.15.4* which addresses low power applications and successively produced the amendment *802.15.4g* that is instead purposely designed to handle smart utility network. In this chapter we explain the relations among these task groups, give an overview of the standard specification and show why these protocols are suited for smart metering networks.

A standardization process, such as the one conducted by organizations like IEEE Standards Association, International Electrotechnical Commission IEC, Internet Engineering Task Force IETF or International Organization for Standardization ISO, is a well defined process whose aim is to publish documents that establish specifications and procedures designed to maximize the reliability of the materials, products, methods and services that can have impact in the industry sector and indeed on final users. The goal is to guarantee product functionality, compatibility and facilitate interoperability. The industry sector can surely benefit from standards because adopting consistent standardized protocols that are universally understood and adopted leads to simpler product development, speeds time-to-market and eventually eases the international trade. Two important things to note about standards are that the development process is very rigid and goes through important steps in order to assure the quality of the outcome. Secondly and determinant is the fact that the so called working group recruited to perform the job is composed of a wide selection of components from people, companies, organizations, non-profits, government agencies who volunteered to support the development of the standard. This way a lot of different points of view, in the terms of design perspective

and different goals, are sifted and examined to achieve the most general solution.

The content of this chapter is organized as follows. Initially, we present the famous *ISO OSI* communication model that can be very useful to better understand the components involved in a smart objects network, and also for clarifying considerations that are presented in later chapters about the Linux software architecture. Secondly, we illustrate the general working principles of the IEEE 802.15.4 protocol, explaining the amendment 4g that we selected for our application. Finally, we present the *6LoWPAN* technology that can enable IPv6 traffic even on constrained data link layers. The latter, which has not yet been fully completed, may permit one day a powerful network of small devices to be deployed on a vast scale, realizing the full potential of the Internet of Things concept.

3.1 OSI Communication Model

An incredible and extensively used standard for networking is the ISO Open Systems Interconnection model (OSI). It standardizes the communication functions of a telecommunication or computing system without regards to the internal structure or implementing technology. As such, this communications model gives an incredible support for interoperability concerns because it permits to specify a single protocol with the ability of being precise about its scope, thus permitting to ease the development of complex distributed systems for telecommunication.

The ISO model, following the level based abstraction, partitions the communicating facilities of a system into several abstraction layers. The classical organization considers seven layers: physical, data link, networking, transport, session, presentation and application.
The important thing to note is that to each layer corresponds a single protocol which can be defined by a standard. This way the scope of the standard is well contained and understood. As we see in Figure 3.1, each layer utilizes the functionality provided by the underlying layer, implements the specific protocols and offers its services to the upper layer. Two important concepts introduced by the OSI model are the protocol data unit *PDU* and the *encapsulation mechanism*. Each layer exchange the information in terms of packets containing the respective PDU, these are passed down in the abstraction heap and are encapsulated as a service data unit SDU for the lower layer, adding protocols information as header data.
A brief description of each layer follows.

Physical layer It defines the electrical and physical specifications of the data connection and the relationship between a device and a physical transmission medium. It determines the transmission mode: simplex, half duplex and full duplex, and specify the right modulation scheme in order to transmit bits on a physical medium, whether the encoded bits will be transmitted by baseband or broadband signaling.

Data link The data link layer provides node-to-node data transfer. It detects and possibly corrects errors that may occur in the physical layer. It, among other things, defines the protocol to establish and terminate a connection between two

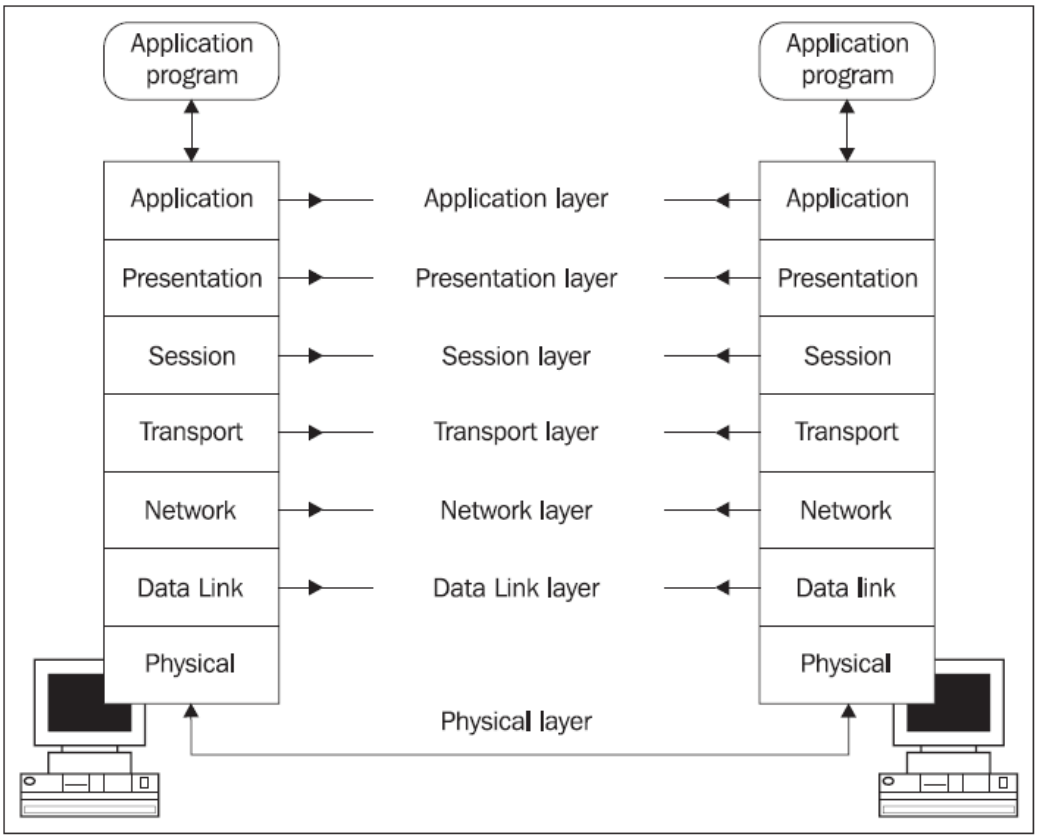


Figure 3.1: OSI communication model

physically connected devices and also defines the protocol for flow control between them. Important functionalities that are commonly seen as two data link sublayers are: medium access control MAC, responsible for controlling how devices access a common medium, and logical link control LLC, for identify network layer protocol and then encapsulating them.

Networking The concept of network can be explained as a medium to which many nodes can be connected, on which every node has an address and which permits nodes connected to it to transfer messages to other nodes connected to it by merely providing the content of a message and the address of the destination node and letting the network find the way to deliver the message to the destination node, possibly routing it through intermediate nodes. The network layer provides the functional and procedural means for transferring variable length data sequences within the same network. It translates logical network address into physical machine address. A number of layer-management protocols are routing protocols, multicast group management, network-layer information and error, and network-layer address assignment.

Transport The transport layer provides the functional and procedural means of transferring variable-length data sequences from a source to a destination host via one or more networks, while maintaining the quality of service functions. The famous TCP pertains to this layer.

Session The session layer controls the connection between computers. It establishes, manages and terminates the connections between the local and the remote application.

Presentation This layer is an adaptation layer responsible for establishing the right context between application layer entities.

Application The application layer interfaces with the user and implement the final information exchange protocol, for example the downloading of an HTML page.

As the objective of this thesis is to understand how to internet-enable an embedded system for the sake of a smart sensor network, in the following chapters we focus only on the physical and on the MAC layer, and give an overview of the problematics to match a data link layer specifically designed for systems with low resources with the network layer IPv6 protocol suit.

3.2 IEEE 802.15.4

The IEEE 802.15 working group's focus is the development of consensus standards for Personal Area Networks or short distance wireless network. The process is overseen by the IEEE Standards Association which in its manifesto shows a list of distinguishing principles that characterize all IEEE standardization activity: openness, consensus, balance and right of appeal. Those guarantee a wide participation of industry partners and especially an open process. Activities of this working group regard wireless networking of portable and mobile computing devices such as PCs, Personal Digital Assistants, peripherals, cell phones, pager and consumer electronics. Main goals concern broad market applicability, coexistence and interoperability with other wired and wireless networking solutions. The portfolio of different standards produced by different Task Groups includes *Bluetooth* technology, high rate WPAN, mesh networking specifications, body area networks and low rate wireless personal area network *LR-WPAN*. The latter, also known as IEEE 802.15.4, deals with low data rate, very long power battery life and very low complexity, thus fitting perfectly for our application.

In this section we present a general overview of the standard and its G amendment, focusing on characteristics that appear more adequate for sensor networks.

3.2.1 Scope

Wireless personal area network WPANs are used to convey information over relatively short distances and in contrast with wireless local area network WLAN, connections effected via WPANs are characterized by having no infrastructure supporting the network.

This way common interactions are mostly run device per device, giving the possibility of being very simple, power-efficient and be extended to an incredible wide range of different types. But, in order to organize efficient communications, the protocol has to be very flexible, extendible and adapts to different scenarios.

The standard defines the physical layer **PHY** and medium access control **MAC** sublayer specifications for low-data-rate wireless connectivity with fixed, portable, and moving devices with no battery or very limited battery consumption requirements typically operating in the operating space of 10 m (the amendment G has the specific purpose of extending this range for outdoor operative conditions). The row data rate is established to be 250 kb/s for satisfying a set of different applications but is also scaleable down to the needs of very low-data traffic instances, for example 20 kb/s or below. The frequency bands are not fixed and a lot of different modulations scheme are provided, a Bluetooth like 2.5 GHz for smaller ranges and an entire set of sub-GigaHertz bands that more appropriate for longer distances and unfavorable environments. In addition, the standard specifies a PHY which is capable of precision ranging that is accurate to one meter and a MAC layer security mechanism that implements **CCM***. It is important to note that although this standard specifies a full physical layer, in respect of the OSI model, only the MAC sublayer of the data link abstraction is elaborated. Designer, in order to realize a full data link layer, must take into consideration that certain aspects, as we explain in the following, have to be completed. Common technologies using IEEE 802.15.4's MAC services that we may found in practice are *ZigBee*, *WirelessHART*, *ISA100.11a* and of course the 6LoWPAN adaptation layer.

3.2.2 General Description

The first argument regards the components which constitute a personal area network **PAN**. In this network there can be two types of devices: a **full-function device** and a **reduced-function device**, respectively **FFD** and **RFD**. An FFD is a device that is capable of serving as a PAN coordinator. An RFD is a device that is not capable of serving as coordinator. An RFD is intended for applications that are extremely simple, such as a light switch or a passive infrared sensor; it does not have the need to send large amounts of data and only associates with a single FFD at a time. Consequently, the RFD can be implemented using minimal resources and memory capacity. Such distinction permits to fully specialize network configurations in the way that is more appropriate for the given application, deploying only the resources that are really necessary.

Two networks topologies are supported. We can see the two different configuration in Figure 3.2. In the star topology, the communication is established between devices and a single central controller, called the **PAN coordinator**. A device typically has some associated application and is either the initiation point or the termination point for network communications. A PAN coordinator can also have a specific application, but it can be used to initiate, terminate, or route communication around the network. The PAN coordinator is the primary controller of the PAN. Applications that benefit from a star topology include home automation, personal computer peripherals, games, and personal health care. The peer-to-peer topology, however, differs from the star topology

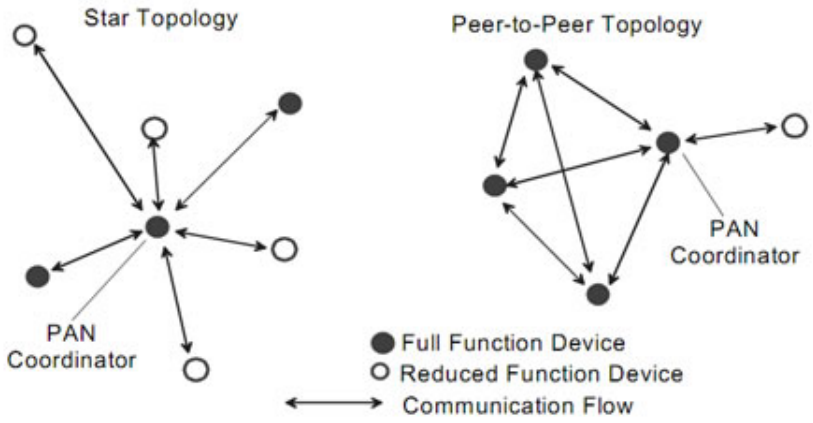


Figure 3.2: Different network topologies

in that any device is able to communicate with any other device as long as they are in range of one another. They also allows more complex network formations to be implemented, such as mesh networking topology and typical applications are industrial control and monitoring, wireless sensor networks, asset and inventory tracking, intelligent agriculture, and security. In a peer-to-peer network multiple hops can route messages from any device to any other device on the network, but this is the task of upper layers. In the network two addresses modes can be use, the extended address which is statically assigned to device and the short address that was allocated by the PAN coordinator during association. As the extended is 64 bits long and the short only 16 bits, this feature allows for traffic optimizations that can be very effective in practice, in that only the necessary payloads are transmitted.

3.2.3 Superframe Structure

Another very useful power optimization is the **superframe structure**. The format of the superframe is defined by the coordinator. The superframe is bounded by network beacons sent by the coordinator, as illustrated in Figure 3.3, and is divided into 16 slots of equal duration. Optionally, the superframe can have an active and an inactive portion. During the inactive portion, the coordinator is able to enter a low-power mode. The beacon frame transmission starts at the beginning of the first slot of each superframe. If a coordinator does not wish to use a superframe structure, it will turn off the beacon transmissions. Anyway, beacons are used to synchronize the attached devices, to identify the PAN, and to describe the structure of the superframes.

This superframe structure is what let the PAN coordinator governs all communications, that is, deciding active and inactive periods of time and allocating **guaranteed time slots GTSs**, each assigned to a specific device. In this manner a single device is able to know when is its transmission time and regulate to perform radio activity only in that period of time. This is incredibly useful because a device can alternate between an

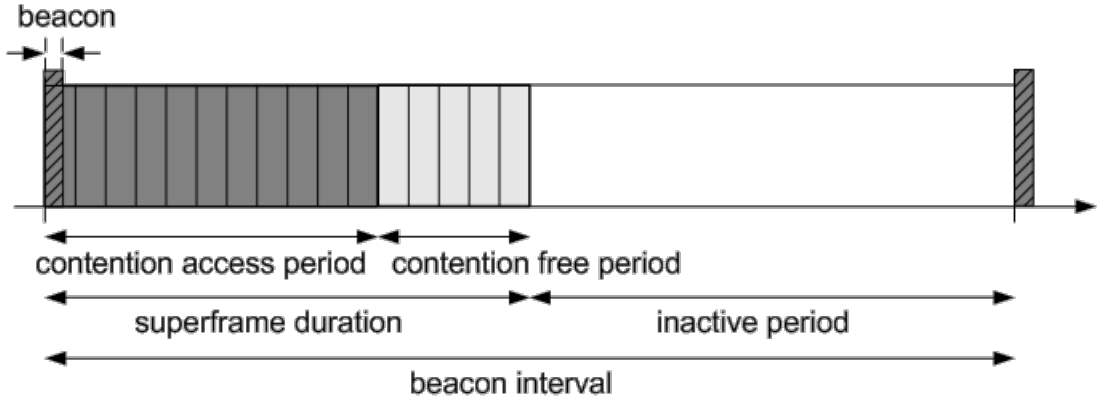


Figure 3.3: Superframe structure imposed by the coordinator

active status and an idle one which can eventually save a lot of resources. As example, battery-powered devices will require duty-cycling to reduce power consumption. These devices will spend most of their operational life in a sleep state and periodically listens to the RF channel in order to determine whether a message is pending. This mechanism allows the application designer to decide on the balance between battery consumption and message latency. Higher powered devices have the option of listening to the RF channel continuously.

3.2.4 Data Transfer Model

In order to completely regulate transfer activities the standard define three type of transactions. The first one is the data transfer to a coordinator in which a device transmits the data. The second transaction is the data transfer from a coordinator and the device receives the data. The third transaction is the data transfer between two peer devices. In star topology, only two of these transactions are used because data is exchanged only between the coordinator and a device. In a peer-to-peer topology, data is exchanged between any two devices on the network, consequently all three transactions are used in this topology.

The IEEE 802.15.4 WPAN employs various mechanisms to improve the probability of successful data transmission.

- CSMA-CA mechanism
- ALOHA mechanism
- optional frame acknowledgment signal
- data verification

3.2.5 Security Mechanisms

The very nature of ad hoc networks and their cost objectives impose additional security constraints, which perhaps make these networks the most difficult environments to secure. Devices are low-cost and have limited capabilities in terms of computing power, available storage, and power drain and it cannot always be assumed they have a trusted computing base nor a high-quality random number generator aboard. Communications cannot rely on the online availability of a fixed infrastructure and might involve short-term relationships between devices that never have communicated before. These constraints severely limit the choice of cryptographic algorithms and protocols and influence the design of the security architecture because the establishment and maintenance of trust relationships between devices need to be addressed with care. The cryptographic mechanism used by the standard is based on symmetric-key cryptography and uses keys that are provided by higher layer processes. The establishment and maintenance of these keys are outside the scope of this standard. The mechanism assumes a secure implementation of cryptographic operations and secure and authentic storage of keying material. As usual the following services are implemented: Data confidentiality, Data authenticity and Replay protection. According to the usual power consumption efficiency the standard permit to adapt the protection on a frame-by-frame basis and allows for varying levels of data authenticity.

The particular technology employed is the **CCM***, a generic combined encryption and authentication block cipher mode. The CCM* mode coincides with the original specification for the combined counter with CBC-MAC for messages that require authentication and, possibly, encryption, but also offers support for messages that require only encryption. Moreover, it can be used in implementation environments for which the use of variable-length authentication tags, rather than fixed-length authentication tags only, is beneficial.

3.2.6 The G Amendment

The *IEEE 802.15.4g* amendment introduces the concept of smart metering utility networks, **SUN**. SUNs enable multiple applications to operate over shared network resources, providing monitoring and control of a utility system. SUN devices are designed to operate in very large-scale, low-power wireless applications and often require using the maximum power available under applicable regulations, in order to provide long-range, point-to-point connections. Frequently, SUNs are required to cover geographically widespread areas containing a large number of outdoor devices. In these cases, SUN devices may employ mesh or peer-to-peer multi-hop techniques to communicate with an access point.

The amendment introduces three new PHYs especially suited for SUN applications, like multi-rate and multi-regional frequency shift keying MR-FSK, the multi-rate and multi-regional offset quadrature phase-shift keying MR-O-QPSK and the multi-rate and multi-regional orthogonal frequency division multiplexing MR-OFDM. In addition particular modifications are made on both PHY level and the MAC level for supporting

a mode **switching mechanisms**. The mode switch mechanism enables a device using the MR-FSK PHY to change its symbol rate and/or modulation scheme on a packet-by-packet basis. This is done as a PHY layer operation, requiring minimal involvement from the MAC layer. A specific mode switch packet is used to inform the receiver of the mode switch and specifies the new PHY mode of the following packets. As an example, this methodology can be used by a device that is configured to operate at the MR-FSK mode with a lower data rate (e.g., 50 kb/s) to enable higher data rate communications when needed. Another mechanisms imported, useful for mitigating interference, is the **multi-PHY management (MPM)**. The MPM scheme facilitates interoperability and negotiation among potential coordinators with different PHYs by permitting a potential coordinator to detect an operating network during its discovery phase using the common signaling mode appropriate to the band being used. Important to note that the 802.15.4g MR-FSK packet definition extends the maximum packet length from 127 bytes to 2047 bytes, thus permitting IP packets without fragmentation.

3.3 6LoWPAN

As we saw in the previous section, services provided by the IEEE 802.15.4 MAC layer are not enough for realizing complex and autonomous networks of smart objects. Even a good medium access control policy, which carefully uses only necessary resources and permit to talk with very simple hardware, has to be extended if we want to build a global public network with sensing and actuating capabilities toward the real world.

It is widely known that the actual IPv4 deployed infrastructure imposes some limitations. In fact, Internet has been an incredible advancement in the last information-centric period of human history and technologies like WWW, email and cloud services could not have existed, but stretching this already loaded infrastructure for connecting a tremendous amount of new heterogeneous devices, has shown some holes in the original design. IPv6, developed by the Internet Engineering Task Force, was born mainly to prevent IPv4 address exhaustion. This new standard represents network addresses with 128 bits, theoretically allowing 3.4×10^{38} different addresses. With 128 bits IPv6 can surpass the IPv4's address capability, of 4.3 billion addresses, by an immense amount. IPv6 provides other technical benefits in addition to a larger addressing space. In particular, it permits hierarchical address allocation methods that facilitate route aggregation across the Internet, and thus limit the expansion of routing tables. The use of multicast addressing is expanded and simplified, and provides additional optimization for the delivery of services. The standards addresses thematics that have pertinance to device mobility, security and configuration aspects. Another more recently development by IEFT is the important 6LoWPAN working group, which concluded in 2014. The specific goal was to propose RFCs about porting the newer IPv6 on low rate personal area networks. Among its publications there are two documents that reason about problematics and give a specification for the introduction of the network concept on Lo-WPAN. Given the fact that its adoption seems imminent, it is certainly a good idea to reason about IoT solutions in terms of an important network protocol update.

In this section we firstly explain what are the benefits of enabling IP, second why it could be problematic to have network protocols running on IEEE 802.15.4 devices, which are usually short range, low bit rate, low power and equipped with very limited computational power. Finally, a short overview of the adaptation layer for IPv6 over IEEE 802.15.4.

3.3.1 Benefits

Configuring and managing computer networks is not a trivial task, especially when this practice regards area networks which are composed of a multitude of heterogeneous devices that can potentially scale to a huge number. Further, reasoning in the direction of connecting this local area network to a global one can only complicate the job. However, extracting more value from an installation must be evaluated as well, since, from the economical point of view, a business may strongly benefit from wider connecting possibilities.
The application of IP technology is assumed to provide the following benefits:

- The pervasive nature of IP networks allows use of existing infrastructure. A lot of indexing capability, for example, are at disposal.
- IP-based technologies already exist, are well-known, and proven to be working.
- IP networking technology is specified in open and freely available specifications, which is favorable or at least able to be better understood by a wider audience than proprietary solutions.
- Tools for diagnostics, management, and commissioning of IP networks already exist.
- IP-based devices can be connected readily to other IP-based networks, without the need for intermediate entities like translation gateways or proxies.
- The many devices in a LoWPAN make network auto configuration and statelessness highly desirable. And for this, IPv6 has ready solutions
- The large number of devices poses the need for a large address space, well met by IPv6.
- Given the limited packet size of LoWPANs, the IPv6 address format allows subsuming of IEEE 802.15.4 addresses if so desired.

3.3.2 Adaptation Layer

Before explaining the adaptation layer we want to reason on several problematics that IEEE 802.15.4 smart object networks have. These can be summarized as follows.

- Topologies considered by the IEEE 802.15.4 protocol are mesh and star. This implies difficulties for the routing protocol since the packet size is very small. The routing overhead should be kept at a minimum, the computing and memory requirements are very constrained and in certain networks there could be devices in a temporary sleep state.
- Limited packet size. Even though applications within LoWPANs are expected to originate small packets, adding all layers for IP connectivity could make impossible the transfer of all the upper layer data into one data link frame. Also, the new amendment IEEE 802.15.4g permits packets with length up to 2047 bytes but for other PHYs that have to remain 127 bytes long, a fragmentation and reassembly layer is probably unavoidable.
- Limited configuration and management. Devices within LoWPANs are expected to be deployed in exceedingly large numbers and the location of some of these devices may be hard to reach. Thus network management should have little overhead, yet be powerful enough to control dense deployment of devices.

The RFC 4944 *Transmission of IPv6 Packets over IEEE 802.15.4 Networks* addresses these problematics.

Transmission Mode

The first thematic regards the transmission mode of IEEE 802.15.4, particularly beacon enabled and non-beacon enabled transmissions. The adaptation layer does not impose any particular requirement on which mode to use, the only services that has to be exported to upper protocols is the capability that beacons can aid in association and disassociation events. This way the IP layer can better handle network attachment.

Addressing Scheme

Also, considerations regulating the addressing scheme must be provided. The two addressing modes natively provided by the MAC layer are both at disposal. Instead, talking of multicast messages, which are not provided by default at the MAC layer, a particular assumption is considering that a single PAN maps directly to a IPv6 link. The correction is simply to translate a multicast on the local link as directed to the 0xffff address, the broadcast address at the MAC layer. This way the network is capable of multicasting on an entire IPv6 link, which is actually a broadcast for the data link layer.

Maximum Transmission Unit

The next important problem is that of the maximum transmission unit. The MTU size for IPv6 packets over IEEE 802.15.4 is 1280 octets. However, a full IPv6 packet does not fit in an IEEE 802.15.4 frame. 802.15.4 protocol data units have different sizes depending on how much overhead is present. Starting from a maximum physical layer packet size of 127 octets and subtracting the header part the resultant maximum frame size at

the media access control layer is 102 octets. Link-layer security may imposes further overhead, which in the maximum case (21 octets of overhead in the AES-CCM-128 case, versus 9 and 13 for AES-CCM-32 and AES-CCM-64, respectively) leaves only 81 octets available. This is obviously far below the minimum IPv6 packet size of 1280 octets and a fragmentation and reassembly adaptation layer must be provided at the layer below IP. Furthermore, since the IPv6 header is 40 octets long, this leaves only 41 octets for upper-layer protocols, like UDP. The latter uses 8 octets in the header which leaves only 33 octets for application data. Without going into details, the adaptation provided is that the particular fragmentation mechanism constructs a particular stack based header which has to prefix each datagrams generated by the network layer. Moreover, the standards defines a header compression mechanism and a format for efficiently obtaining an IPv6 interface identifier from the address setting of the underlying layer, in a stateless manner.

Chapter 4

Linux Architecture

Recently, the operating systems scenario has slightly changed in respect of the embedded world. The kind of software we are able to run on a typical hardware architecture today, some years ago was very difficult to efficiently use or even impossible to execute. The typical machine that was widely used did not implement special interesting and practically useful features, like different execution modes, memory management or full interrupt support, instead commonly found in PC architectures. But, as we explain in Chapter 5, the situation is now completely different and a lot of different *system-on-chip*s are capable of running full featured operating systems. In previous chapters we have already outlined some aspects of using an operating system for developing IoT networks and, besides giving a large suite of abstractions, the operating systems can serve as a fundamental tool for the development process and for the support of a wide set of different devices. The Linux operating system, as an open source software product, is very good in these features, because of the huge community is helping in its development, the system is extremely configurable and support and incredible amount of different machine architectures and devices.

In this chapter we present an overview of some distinctive characteristics of this open source operating system. In particular we illustrate some high level interfaces, the methods the community uses to advance this huge piece of software and some benefits in having the own software included in the main line of the community development process. An important point we want to stress is the support provided for developing driver code. This fact turns out to be very useful for adapting the kernel to own particular hardware. As explained in following chapters, about the driver development, one person can take advantage of a well designed and very general architecture and can make use of an extensive set of different subsystems that handle common hardware characteristics. In this chapter we want to explain the module system and the driver model framework. Given the fact this thesis is about network protocols, we provide a quite detailed explanation of the network subsystem, showing the Linux's capability of configuring the network stack extensively. Aside from a classic BSD-style socket interface, the kernel internally permits to efficiently manage network traffic by routing packets throughout different network protocol driver and exposes an interface for managing network virtual

devices in an advanced manner.

Finally, we expose some considerations regarding the employing of Linux in embedded applications, which is turning into an important reality these days as this extremely configurable operating system permits to implement valuable abstractions even for modern hardware architectures. In fact, the adaptability of Linux for the incredible amount of heterogeneous hardware systems, like system-on-chip and the more recently system-on-module style for embedded platforms, is showing a possible future for this community developed kernel, as it is supporting more and more chips every day.

4.1 Overview and General Concepts

Linux embed in its original philosophy the fact of being a system for writing software, inheriting this attitude from the classical Unix, and has always been oriented toward the principles of multiprogramming: many programs are permitted to run simultaneously, and multiuser: several users can access system resources at the same time. Simplicity, elegance and consistency are some of the qualities of a Linux system and the most important guideline is that every program in the system should implement only one thing and do it well. Technically speaking Linux is a monolithic kernel where the system call interface is quite extended and lot of functionality, such as virtual consoles, network configuration and routing, and much of the inter process communication facilities are built directly into the kernel itself. Talking about Linux as an operating system or a kernel is pretty equivalent, indeed. In the following, after a brief historical overview, we illustrate Linux's features by analyzing principal abstractions an operating system usually provide: processes, memory and I/O.

4.1.1 Origins

Linux started in 1991 as a personal project of Linus Torvalds, a young Finnish researcher, who believed it was the right time to start writing a new operating system from scratch. This new project attracted a lot of people and in few years, when the version reached 1.0, in 1994, the code base was already counting 165000 lines and a lot of interesting features, such as a full new file system, memory mapped files and a networking system with BSD-like socket interface, were already implemented. Linux is widely known as Unix clone. Unix was a new concept of operating system design, developed together with the C programming language at Bell Labs. The first paper about Unix was published by Ritchie and Thompson in 1974. In the following years a lot of new versions were published and more and more departments adopted this software as operating systems for their computers. As in the initial years the license included the source code, different companies and universities modified it. Two famous versions were System V from AT&T and BSD from Berkeley University. What inspired Torvalds in writing his new version was Minix, a small microkernel born in mid-80s with stability and easy to understand characteristics as design goals that tries to respect the Unix design philosophy. One of the famous reasons for the expansion of Linux over Minix was that the former was

willing to accept new features whether the latter was more conservative in order to keep things simple.

4.1.2 Abstractions

When computers reached a point where their hardware turned into a mix of sophisticated components, such as processors, memories, disks, I/O devices, having a manager that simplifies some operations, truly take advantage of underlying features, and provide the user with powerful abstractions became a priority in software systems design. An operating system is a piece of software that can be interpreted as a resource supervisor and regulate all operations happening in the machine. Base concepts that are usually implemented in an operating system includes: processes, virtual memory, files, input/output, protection and the shell. These features are implemented with a powerful mechanism called **system call** that is used to "trap" the guest activity for making the supervisor intervene and regulate the use of resources.

Processes

Processes in Linux follow the Unix specification, with some differences. Through the **fork** system call a process can duplicate its resources and generate a child process. The new process can execute a different executable by issuing the **exec** system call. Inter process communication is achieved with so called **pipe**, a communication channel where a process writes a stream of bytes and the other reads it, and with the concept of **signal**s, that are more massage-passing based. A particular characteristic of Linux is how it organizes the process descriptors information. In fact the *task_struct* represents all kind of executing context and so if a process is multithreaded a single *task_struct* is present for each of them. This peculiarity gives a lot of granularity for the management of resources linked with processes. This structure contains a lot of informations which may fall under this categories: scheduling parameter, pointers to memory image segments, masks for signals, machine registers, system calls state, file descriptor table and general accounting. A mechanism worth noting is the **copy on write** that lets the kernel quickly allocate resources when a new process is started. The allocation actually starts when the process modifies this resources for its needs, for example it writes a new text image upon an **exec**. Another important characteristic concerning process creation is the introduction of the **clone** system call. This function permits to fully specify which resources are shared between a thread and other threads of the same process, so to achieve a fine granularity that allows to share only what is really needed and to be quick in allocation of new threads-related resources. Regarding scheduling of processes, Linux organizes scheduling by partitioning run queues in three classes, two more oriented to real time latencies: **real-time FIFO** not preemptable and **real-time round-robin** instead preemptable. The other class, called **SCHED_OTHER** is handled by the *Completely Fair Scheduler (CFS)* that has recently been introduced. CFS basically models an "ideal, precise multi-tasking CPU" on real hardware, which equally divide the computing power among tasks in execution.

Memory

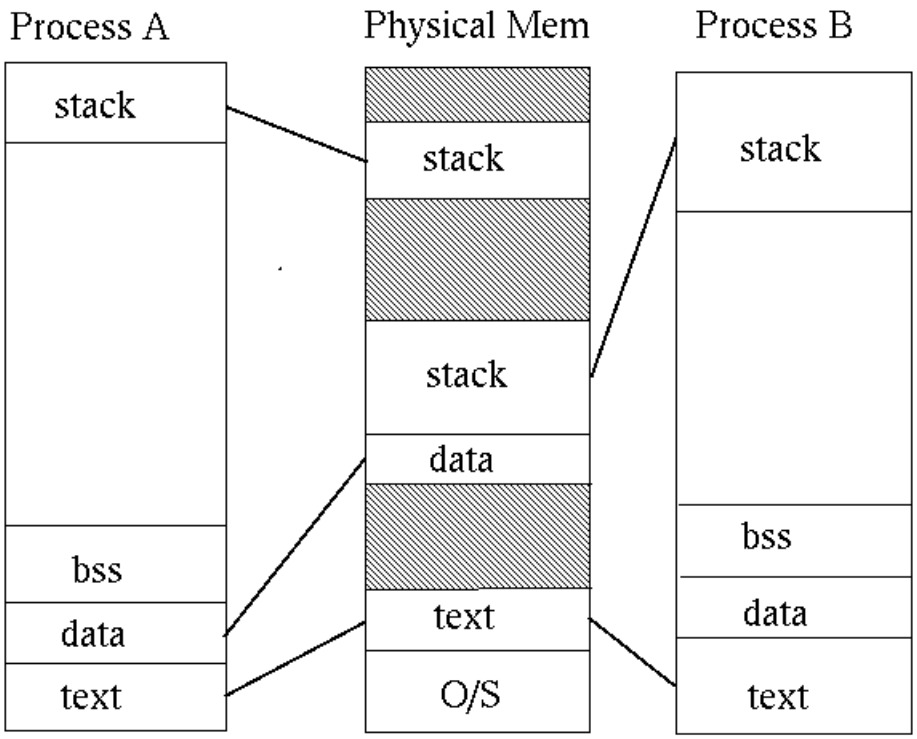


Figure 4.1: Linux memory mapping and organization

Linux organizes process memory, as other Unices, with the concept of **segments**. This way, every memory space is partitioned into three segments, ***text***: holding program instructions, ***data***: that is divided in static data and uninitialized data (**BSS**), ***stack***: which keeps track of procedure contexts. The last two memory segments are organized as in Figure 4.1. Stack and BSS are expected to grow and the kernel manages this by incrementing each segment in the opposite direction, the stack grows downward and the BSS upward. Two important system calls for memory allocation are **brk**, which asks the kernel to augment the data segment and **mmap** that implements the powerful mechanism of *memory mapped files* where the content of a file can be mapped directly in the address space of the process. Linux handles pagination by dividing the memory into three zones: **ZONE_DMA**, **ZONE_NORMAL** and **ZONE_HIGHMEM**. To these different classes corresponds different pagination policies for differentiate memory allocation algorithms, and are respectively: pages for DMA use, normal mapped pages for use by process and memory zones where the mapping is not performed continuously.

Input Output

Beside normal filesystem operation, the I/O management in Linux follows the important concept of representing device operations in the same way of file operations, as other Unix systems do. This way common devices appear in the file system under */dev/* and are of two types: **block devices** and **char devices**. The distinction is made to permit two different access types, one that is more suitable for disk-like devices and the other for streams of bytes. Another important I/O kind of operation is the one given by the **socket** interface for networking operations. Aside from being represented by file descriptors, the inner working of sockets is quite different for the intrinsic asynchronous nature of the network. Sockets can be represent network traffic of different type:

- bytes stream with reliable connection
- packets with reliable connection
- packets without reliable connection

Important input/output related system calls are: **open**, **write**, **read**, **close**, **ioctl**. These permit classic interaction with normal files and special files for device operation. For network operations, instead, one can use **socket**, **listen** and **connect**.

Configuration and Modules

The configuration and build method of Linux is very effective and permits to include in the final image only desired features. This is possible because the build process which is based on the famous *GNU make* utility can understand, as declared in **Kconfig**, what components of the system are necessary for a specific feature and include them in the final binary. This utility regards the choice of the underlying machine architecture, driver support, filesystems, graphics, debug/development facilities and some internal advance feature like *cgroups*. Being able to specify such feature allows to obtain a system that is small and boot fast. Another important characteristic of the Linux kernel is that it supports *loadable kernel modules*. Previously one had to include necessary features during the configuration and build process. This process presents some inflexibility as one have to foresee exactly what features the system has to include and because some machines varies physical configuration dynamically, as for example when an USB stick is plugged in or when a network bootstrap is performed. With this feature Linux can load big piece of code dynamically, for example device driver, entire filesystems, network protocols and auditing and monitoring components. To load a module dynamically the user can use the **modprobe** command that automatically call the right system call Linux made available for this service. The loading operation is quite complex: first the module need to be reallocated on the fly, second the system has to check dependencies and acquire them, then setup various interrupt related entries and allocate device major and minor number.

4.1.3 Community

Given the fact that the Linux kernel has grown incredibly over the years, the community follows some particular methodologies for advancing the code base. The approach of the community is to keep track of different *git* repositories, each maintaining a particular version of the system in a particular state. In order to integrate some changes into the kernel, a person have to fully understand how various repositories are connected, especially because the development process advances at a rapid pace.

mainline This is the process by which new features are introduced into the kernel. As soon as a new kernel is released a two weeks window is open, during this period of time maintainers can submit big diffs directly to Linus in order to integrate changes in various subsystems into the mainline. After two weeks a -rc1 kernel is released it is now possible to push only patches that do not include new features that could affect the stability of the whole kernel. The process finishes when the perceived bug-level status is quite below a given threshold.

stable The stable repositories are usually composed of three numbers, as in $4.x.y$. The x is the major referring to kernel version, while y the others are critical fixes for security problems or regressions discovered after the release. This is the option a user has to choose to get to most stable version of the kernel.

subsystem specific Linux is divided in a lot of subsystems, for example: networking, filesystems, graphics, etc. Each subsystem is associated with a mailing list where discussions about problematics are conducted and patches with code modifications are reviewed.

next This repository gives the opportunity to integrate modifications that will be expected to go into the mainline at the next release, so this is the place where most advanced development occurs.

4.2 Driver Model

In this section we explain one of the most important architectural features of Linux. The *Linux Kernel Driver Model* is a unification of all the disparate driver models that were previously used in the kernel. It is intended to augment the bus-specific drivers for bridges and devices by consolidating a set of data and operations into globally accessible data structures, thus providing a common logic for the management of device resources and for the problem of matching a driver to the specific device found on a generic bus. The current driver model provides a common, uniform data model for describing a bus and the devices that can appear under the bus. The unified bus model includes a set of common attributes which all busses carry, and a set of common callbacks, such as device discovery during bus probing, bus shutdown, bus power management, etc.

To give an indication of how this generalization is achieved in the kernel source code consider the following fragment:

```
struct pci_dev {
    ...

    struct device dev;      /* Generic device interface */
    ...
};
```

This way each device dedicated to a specific purpose has its own data structure but also embed general information with the `struct device dev` inner declaration, thus common code can behave with the same logic by accessing the substructure fields that are present in each device in the kernel. This method resemble a simple object oriented programming style. Important concepts of this architecture are:

bus Each bus has to implement a particular a set of callback to adhere to the model, as matching device to drivers or exporting attributes to other common interfaces, such **sysfs**. These are set up during registration.

binding When a new device is added, the bus's list of drivers is iterated over to find one that supports it. In order to determine that, the device ID of the device must match one of the device IDs that the driver supports. The format and semantics for comparing IDs is bus-specific.

class Each device class defines a set of semantics and a programming interface that devices of that class adhere to. Device drivers are the implementation of that programming interface for a particular device on a particular bus.

devres Devres is a common infrastructure to provide support for handling resources allocated by drivers. This system can automatically understand which features a driver needs and deallocate upon unregistering. Managed resources are clock, dma, gpio, irq and memory allocation.

4.3 Network Subsystem

As previously explained the Linux kernel is a monolithic software system that creates a clear barrier between supervisor code and user code, and a lot of features are provided behind this interface. In order to be so sharply defined the system has to expose good configuration methods that can permit user code to tune some inner working mechanism to his/her needs. This is the case of the networking subsystem, which provides the implementation of a lot of protocols, as TCP/IP for instance, a routing mechanisms with features commonly found in a typical hardware router and a full suite of drivers for network devices. Furthermore, in order to configure this complex stack of software the Linux networking subsystem offers a socket base interface that is very general and powerful, capable of providing the user with access to inner interface configuration and events.

4.3.1 Netlink

Technically speaking **netlink** is a Linux kernel interface used for inter-process communication (IPC) between both the kernel and userspace processes, and between different userspace processes, in a way similar to the Unix domain sockets. Similarly to the Unix domain sockets, and unlike INET sockets, netlink communication cannot traverse host boundaries. However, while the Unix domain sockets use the file system namespace, netlink processes are addressed by process identifiers. This interface is extremely powerful to permit a total configuration of the network stack. Without going into details, it is possible to intercept packets and forward them to specific handlers for implementing filter, firewalls and special signaling protocols. In simple terms netlink permits to manually exchange sockets traffic, bound to specific protocols, between kernel space and user space.
The interface can be used with classic sockets interface

```
#include <asm/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>

netlink_socket = socket(AF_NETLINK, socket_type, netlink_family);
```

The `netlink_family` parameter selects the kernel module or netlink group to communicate with. The following shows some of these modules.

NETLINK_ROUTE Receives routing and link updates and may be used to modify the routing tables (both IPv4 and IPv6), IP addresses, link parameters, neighbor setups, queueing disciplines, traffic classes and packet classifiers.

NETLINK_FIREWALL Transport IPv4 packets from netfilter to user space. This can be used to intercept and filter inbound or outbound traffic.

NETLINK_AUDIT Auditing services.

NETLINK_CONNECTOR Recent netlink simplified user space and kernel space communication module. The Connector driver makes it easy to connect various agents using a netlink based network. One must register a callback and an identifier. When the driver receives a special netlink message with the appropriate identifier, the appropriate callback will be called.

NETLINK_NETFILTER A framework that allows various operations to be implemented in the form of customized handlers. It offers various functions and operations for packet filtering, network address translation, and port translation.

NETLINK_GENERIC Generic netlink family for simplified netlink usage.

4.3.2 Netdevice

Netdevices are another important example of the philosophy of Linux. Being physical network-related devices usually made with similar characteristics, the kernel has maintained this interface which permits to reuse a good amount of code by incorporating common parts into a single layer. There is also a special API to help setting up features a device can switch on when some events occur. Features regards hardware checksumming, TCP related segmentation and scatter-gather accelerations. The following gives an overview of the support provided by the kernel. The subsystem partitions each feature in different classes:

- `netdev->hw_features` set contains features whose state may possibly be changed (enabled or disabled) for a particular device by user's request.
- `netdev->features` set contains features which are currently enabled for a device. This should be changed only by network core.
- `netdev->vlan_features` set contains features whose state is inherited by child VLAN devices. This is currently used for all VLAN devices whether tags are stripped or inserted in hardware or software.
- `netdev->wanted_features` set contains feature set requested by user. This set is filtered by ndo_fix_features callback whenever it or some device-specific conditions change. This set is internal to networking core and should not be referenced in drivers.

4.4 Linux Kernel for Embedded Platforms

Linux is an operating system capable of running on an extensive list of machine architectures. To give a short list, we outline only the most common: Alpha, Atmel AVR32, Texas Instruments TMS320, Qualcomm Hexagon, IBM Power, Intel x86 and x86-64, Sparc and of course ARM. We can think about this good support as one of the benefits of being an open source operating system. In fact, a machine vendor who has already developed a new architecture understand that porting Linux on his/her machine could be a good investment. Given the fact that the whole code base is at disposal and the community is very strong and consolidated, the porting process should not be an expensive task, at least in recent years as many tools are mature and Linux itself has been evolved to be extremely adaptable to a wide scenario. In return, offering a machine with the capability of running an operating system that has become widespread, could only increase the final value. This situation well depicts the scenario of embedded systems where different manufacturers are taking part in the Linux development. Such progress is having an incredible positive effect in developing software for embedded systems, which was previously quite complex as many framework fitted only few machines and a company had to build from scratch many of the tools. With the wide adoption of Linux, making it runs on a new system is no more a matter for only experts. In this section we

want to illustrate some of the concepts that explain why it is possible to run a kernel originally conceived for servers and workstations on a small machine. We introduce the U-Boot bootloader as it is a common choice and because we used it in our development.

4.4.1 Bootloader

One of the most important steps in bootstrapping an operating system on a machine is the loading process. Right after the power on sequence a machine need to be properly configured in order to load a kernel image, which is usually made up of some mega bytes of data and resides in some memory that is not usually directly accessible by the CPU. This is also true in embedded systems where the image is flashed in NAND memories or has to be transferred from the local network. So, there is the need of an initialization step that is capable of setting up the machine right after power on, retrieving the binary image of the kernel, extracting it in the right place and that correctly implements the boot protocol, which is about composed of: passing command line parameters in the right location, organizing the memory layout and setting the right execution context.

A widely used bootloader for embedded application is *U-Boot* which is a very useful tool that runs on PowerPC, ARM and MIPS systems. Besides implementing the Linux booting protocol U-Boot supports a lot of hardware, has a quite comfortable command line, scripting capability, support communications over serial link and is able to boot from different mediums. This environment is very small but invaluable since it aids in the process of porting more complex operating systems. In fact, this reflects a very important difference between embedded systems and PC, where the start up sequence has been dictated by big companies and has remained pretty the same for many years. We list some useful features of this bootloader:

Information that retrieve basic information on the hardware system as discovered by U-Boot

Memory that access the main memory for displaying or modifying

Flash Memory to copy or erase flash memories

Download Command such as `bootp` for booting via network, `loadb` for loading a binary over serial line

Flattened Device Tree Support that manages an important data structure for describing hardware configurations

4.4.2 Device Tree

Device tree is a recent concept that has been diffusing among embedded operating systems during last years. The main objective of the device tree is to represent an hardware configuration. It is a format for describing which device is connected to the CPU, what kind of controller the machines is equipped with and, even more essential, the interrupt disposition and the memory regions, and passing such information to the kernel. This

way the software can be very well parametric and adapt to different configurations. It is derived from the device tree format used by *Open Firmware* to encapsulate platform information. The device tree data is typically created and maintained in a human readable format in *.dts* source files and *.dtsi* source include files, but is then compiled into a binary format contained in a *.dtb* blob file, also called FDT. The Linux operating system uses the device tree data to find and register the devices in the system. The FDT is accessed in the raw form during the very early phases of boot, but is expanded into a kernel internal data structure for more efficient access for later phases of the boot and after the system has completed booting. A device tree is represented as a tree structure where each node possesses a list of attributes. Here is an example

```
soc {
    compatible = "nvidia,tegra20-soc", "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;

    intc: interrupt-controller@50041000 {
        compatible = "nvidia,tegra20-gic";
        interrupt-controller;
        #interrupt-cells = <1>;
        reg = <0x50041000 0x1000>, < 0x50040100 0x0100 >;
    };

    i2c@7000c000 {
        compatible = "nvidia,tegra20-i2c";
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <0x7000c000 0x100>;
        interrupts = <70>;

        wm8903: codec@1a {
            compatible = "wlf,wm8903";
            reg = <0x1a>;
            interrupts = <347>;
        };
    };
};
```

In the listing we can see some features of this language. Each node is declared with the syntax

```
label: name @ address
```

The label is for referencing or alias and the address for specifying the memory location. The kernel during the booting process parses this structure and tries to understand the

content. The attribute `compatible` serves to indicate the model of the device, permitting the kernel to search for drivers supporting that specific device. Other attributes are always parameter to pass to the driver, such as register locations or interrupt values.

4.4.3 Architecture Specifics

The last comment regards machine specific code that is included into the Linux kernel, usually under `arch/arm/mach-*`. If a person is interested in porting to a new machine is probable that he/she will have to write some low level ad-hoc code. Linux, indeed, even if it is very general and a lot of maintainer fight the battle of code reuse and maintainability, some details remain and have to be fixed. A particular machine can have specific instructions, timers, busses, can require cpu and memory initialization, the interrupt handling system may have some peculiarities and, further, there could be specific hardware designed to solve important problems, such as integrated power manager and IO multiplexers. Linux provides the developer with hooks that are called at some stage of the loading process or can establish an interface the kernel uses for certain operations, for example when the system needs to go in idle state the kernel should call the architecture specific hook for that purpose.

Chapter 5

Hardware Description

What is referred to with the term *embedded system* is a computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints. An embedded system is designed to do some specific task, rather than be a general-purpose computer for multiple tasks. This characteristic have made embedded system architectures fundamentally different from personal computer, workstations or server. This is due to the fact that an embedded hardware system had to be designed specifically for the final purpose and, some time ago, putting a lot of complexity and additional features could have caused high development costs, high configuration costs and difficulties in writing software for that specific architecture. But the situation is now different. Advancements in machines development process, as modern tools for designing digital circuits that can automatically optimize low level components configuration or even identify some bugs in the design, has certainly raised the total process performances giving to hardware manufacturers the possibility of shipping full-featured digital components. One trend that supports this evolution is digital circuit integration technology that is frequently producing new solutions, such as *microcontroller*, *system-on-chip* and the more recently *system-on-module*. Those are advancements toward manufacturing costs reduction and the production of smaller systems. Since embedded system are dedicated to specific tasks, designers can optimize it to reduce the size and cost of the product and increase the reliability and performance.

Among wide use architectures, nowadays we have seen the expansion of the ARM technology. This kind of machines has increasingly become very common and has started to compete with PC predominant counterparts like Intel *x86*. And most important, even if ARM systems are almost exclusively used for embedded applications, in general they tend to provide a whole set of modern features: accelerated graphical applications, hardware virtualization and multicores that scales up to 8 core, all of this with low power consumption performances. Those systems range from portable devices such as digital watches and MP3 players, factory controllers, and largely complex systems like hybrid vehicles, MRI, and avionics. It is clear that this technology will play an important role for the expansion of IoT. This chapter is dedicated to illustrate the development hardware we used for practical experiments. We first present the *AMBER* platform and

show some advanced characteristics of this modern-style development system. Second, we give some details on the transceiver CC1200 by Texas Instruments that is very well suited for smart utility networks.

5.1 The AMBER Platform

Amber is a new open platform designed to be a scalable and flexible general purpose gateway and test platform for any new IoT activity: both hardware and/or software. It is an open platform in the sense that all design documents are at disposal and covered by a *Creative Commons* license. The platform is very flexible as it permits to install the SOM of choice with a **SO-DIMM 200-pin** connector. Even if at the moment the *Variscite VAR-SOM-MX6* module is the only fully supported by the board, integrating another 200 pins SOM should be a very small effort. Another important characteristic is that the system make accessible all internal ports on the SOM by exporting them on useful connectors on the board, such as mini-PCIe. This way there is a wide availability of signals: SPI, I2C, UART, USB, GPIO, PWM and CAN. Other features are:

Video Many video outputs can be populated for external video out: *LVDS, HDMI, RGB*. Video-in interface is present on Amber board through an FCI 40 pins connector where both MIPI-CSI and MIPI-DSI are present. When required, Amber can operate with up to 3 displays at the same time.

Connectivity An *USIM* card slot and a *1 Gigabit LAN* connector are on board. Wireless and *Bluetooth / BLE* connectivity are directly integrated in the SOM board. Other wireless connections like *ZigBee, ISA100.11a, WirelessHART, MiWi* or beyond, have to be realized with an extender, as we did for integrating *IEEE 802.15.4g* into this board.

Touch screen Both resistive and capacitive ports are available for a touch panel, SoM must have the feature.

Audio as well is present with two 3.5 mm standard connectors for line-in stereo and line-out stereo. An on board MIC is also present.

5.2 Texas Instruments CC1200

A transceiver is a device comprising both a transmitter and a receiver which are combined and share common circuitry or a single housing. This kind of system has been widely used in embedded applications. The fact of reusing some components permits to integrate advanced functionality, especially in modern integrated radio circuits, where there is the need of low power modes, for instance. Together with a computing processor, capable of driving and taking advantage of all the implemented features, these radio systems are the essential parts for building modern IoT devices. Meeting the most stringent RF requirements in the market, the Texas Instruments RF performance line family has the

most reliable range in the industry. A good range is achieved by high output power and excellent sensitivity. The RF performance line family can be in closer proximity to the other RF systems and potential interfere without any disturbance to the RF link. The RF familys advanced RF channel sniff mode feature ensures quick startup and settling time, and enables a current consumption of sub-3 mA in sniff mode. The sniff mode allows systems to listen for RF packets using very low power consumption while maintaining full RF performance. Ideally suited for low-power, high-performance systems, the CC1200 RF transceiver offers a data rate up to 1 Mbps and years of life for battery-powered applications through low-power operation with sniff modes and fast settling time. The CC1200 supports all the IEEE 802.15.4g FSK modes with hardware packet handling as well as hardware AES security support and all wM-Bus modes with great performance.

5.2.1 Summary of Characteristics

We present a brief list of principal technical features directly extracted from the datasheet in order to give an overview. In the following, we provide an explanation of some radio terminology to help the reader to better understand some concepts, and a deeper explanation of the digital features.

RF Performance and Analog Features

- Excellent Receiver **Sensitivity**: 109 dBm at 50 kbps
- **Blocking Performance**: 86 dB at 10 MHz
- **Adjacent Channel Selectivity**: Up to 60 dB at 12.5-kHz Offset
- Very Low **Phase Noise**: 114 dBc/Hz at 10-kHz Offset (169 MHz)
- Programmable Output Power Up
- Supported Modulation Formats: 2-FSK, 2-GFSK, 4-FSK, 4-GFSK, MSK, OOK
- Supports Data Rate Up to 1.25 Mbps in Transmit and Receive

Low Current Consumption

- Enhanced Wake-On-Radio (eWOR) Functionality for Automatic Low-Power Receive Polling
- Power Down: 0.12 A (0.5 A With eWOR Timer Active)

Digital Features

- WaveMatch: Advanced Digital Signal Processing for Improved Sync Detect Performance
- Security: Hardware AES128 Accelerator
- Data FIFOs: Separate 128-Byte RX and TX

- Includes Functions for Antenna Diversity Support
- Support for Retransmission
- Support for Auto-Acknowledge of Received Packets
- Automatic Clear Channel Assessment for Listen-Before-Talk Systems

Dedicated Packet Handling for 802.15.4g

- CRC 16/32
- FEC, Dual Sync Detection (FEC and non-FEC Packets)
- Whitening

5.2.2 Radio Communication Terminology

Receiver Sensitivity Range is an important requirement for most any RF application. Communication system achieve long range with modulation and demodulation techniques coupled with good receivers sensitivity specifications. Receiver sensitivity is the lowest power level at which the receiver can detect an RF signal and demodulate data. Sensitivity is purely a receiver specification and is independent of the transmitter. As the signal propagates away from the transmitter, the power density of the signal decreases, making it more difficult for a receiver to detect the signal as the distance increases. Improving the sensitivity on the receiver will allow the radio to detect weaker signals, and can dramatically increase the transmission range. Sensitivity is vitally important in the decision making process since even slight differences in sensitivity can account for large variations in the range.

Receiver Blocking Performance When a very strong off channel signal appears at the input to a receiver it is often found that the sensitivity is reduced. The effect arises because the front end amplifiers run into compression as a result of the off channel signal. This often arises when a receiver and transmitter are run from the same site and the transmitter signal is exceedingly strong. When this occurs it has the effect of suppressing all the other signals trying to pass through the amplifier, giving the effect of a reduction in gain. Blocking is generally specified as the level of the unwanted signal at a given offset - often 20 kHz - which will give a 3 dB reduction in gain. Dependent upon the type of receiver, the values for blocking will vary considerably. As a reference point, a good communications style receiver may be able to withstand signals of about 10 dBm before this happens.

Adjacent Channel Selectivity Adjacent Channel Selectivity ACS is a measure of a receivers ability to receive a signal at its assigned channel frequency in the presence of a strong signal in the adjacent channel. ACS is defined as the ratio of the receiver filter attenuation on the assigned channel frequency to the receiver filter attenuation on the adjacent channel frequency.

Phase Noise Phase noise occurs when a system introduces disturbances in the phase of signals produced, implicating a considerable amount of undesired spectral components to arise nearby the main carrier. This phenomenon may involve difficulties in separating the wanted frequency component from signals close to it, so it has to be taken into account especially for narrow band application. It is measured in dBc/Hz, where dBc (decibels relative to the carrier) is given by the power ratio of a signal to a carrier signal, thus providing information on the robustness of the system to interfering frequencies present near the channel.

Link Budget A link budget is accounting of all of the gains and losses from the transmitter, through the medium (free space, cable, waveguide, fiber, etc.) to the receiver in a telecommunication system. It accounts for the attenuation of the transmitted signal due to propagation, as well as the antenna gains, feedline and miscellaneous losses.

A link budget equation including all these effects, expressed logarithmically, might look like this:

$$P_{RX} = P_{TX} + G_{TX} - L_{TX} - L_{FS} - L_M + G_{RX} - L_{RX}$$

where:

$$P_{RX} = \text{received power (dBm)}$$
$$P_{TX} = \text{transmitter output power (dBm)}$$
$$G_{TX} = \text{transmitter antenna gain (dBi)}$$
$$L_{TX} = \text{transmitter losses (coax, connectors...) (dB)}$$
$$L_{FS} = \text{path loss, usually free space loss (dB)}$$
$$L_M = \text{miscellaneous losses (fading margin, body loss, polarization mismatch) (dB)}$$
$$G_{RX} = \text{receiver antenna gain (dBi)}$$
$$L_{RX} = \text{receiver losses (dB)}$$

5.2.3 Digital Features

The device exposes and interface which can be used to control different states the internal state machine can pass through. In this section we give only an high level overview, since in Chapter 6 we have to better explain some details for the driver development.

Command interface Over a *Serial Peripheral Interface SPI* the transceiver provides a programming interface (in Chapter 6 we show some details about it). This way is possible to access the memory of the device for read or write operations, to configure its behaviour and to read informations about the current working status. The memory layout can be seen as a set of registers each with a different purpose,

such as enabling cryptographic operations, setting interrupt signals and conditions, and controlling the radio state machine for switching between transmissions and receptions.

Packet support The device has internally two 128 bytes memories for buffering receive and transmit operations. This are used to unload the processor from some synchronous radio operation that can block it too much frequently. These buffers contains the row data packet being transmitted or received. It is possible to instruct the transceiver to interrupt the CPU when a packet has been received or a transmission has completed. Internally, there is also the possibility to setup an automatic acknowledge response when a correct packet has been fully verified.

Wake On Radio This is an interesting feature as it permits the main processor to go in low power consumption states. The radio systems, itself in low power state, is capable of detecting pertinent radio activity and wake up the CPU with interrupt signals. Being able to use such functionality could really improve battery life.

AES Accelerator This is an embedded accelerator for implementing cryptographic operations. This possibility further offload the CPU from packet related computations.

Chapter 6

Software Development

Another important activity in developing IoT applications is the software development process. Even if the application does not require particular algorithms, efficient data structures or meticulous code optimizations, the act of writing computer code for implementing whatever application logic or functionality is not a trivial task. A discipline which tries to govern the process and identify good methodologies among those that are bad or ineffective is *Software Engineering*. Thematics that are usually questioned regard:

- Keeping the **documentation** in a good state as the software evolve. This is important as the documentation describes the behaviour of the program from an high level point of view, thus helping a lot in reasoning about code properties and because it establishes a communication medium between developers. Other documentation kinds may be more end-user oriented. Maintaining a good documentation is a difficult process because the possibility to go out of synchronization is easy, as in some case the code grows and evolve at a rapid pace.

- The **testing** phase is also an important activity as the system might exhibit some strange behaviour that was not conceived during the writing process. It is very important to execute the product under different conditions in order to see what are its weaknesses and to assure an acceptable user experience.

- Some people comment that the most difficult part is **bug fixing**. Some bugs can emerge during the test phase, in which the software crashes under certain configurations, but also after the latest release where the user directly experiences strange behaviour or anomalies. Other problems regard the vulnerabilities a bug may entail. Those are even more subtle, difficult to find and also to correct.

These thematics are very important and can incisively condition the final product quality and time to market parameters. Almost every software system is affected by these problems but it is also true that some type of systems present their particular difficulties. For this thesis we worked closely with the operating system and we found

that writing code intended to run without the protection of a supervisor, such as the code to implement a driver, may involve some additional issues. The most important of these is when a program running in kernel space executes some harmful operation, the kernel may not be able to recover the situation and can halts abruptly. This circumstance force the developer to restart the system over and over again until the problem is resolved, wasting a lot of valuable time. Another related problem is the difficulty in examine what caused a crash, for instance, because being inside the operating system itself lack the availability of important monitoring tools, such as debuggers.

For embedded software the situation complicates the process a little more. This is due to the embedded systems nature of being almost totally integrated in devices and packages. So, for the developer is more difficult to interact and observe inner behaviours as some special connections have to be prepared specifically for this purpose. Other special requirements regard the toolchain that, when the machine being developed: called **target**, and the machine used for the development: **host** are different, has to be carefully configured. The latter concept is called *cross-compilation*. In following sections we explain further what it means. Also, memory images being produced have to be transferred from the host to the target. This mechanisms, sometimes called *flashing*, is very delicate since the image must be formatted in specific ways and the location is usually machine dependant and has to be carefully determined.

The choice of programming languages, in embedded software and especially for operating systems, is pretty reduced since the lack of a runtime support, always provided by high level languages, implicates the use of low-level and poor-abstraction-mechanisms programming languages. At the time in which Unix was first invented, proceedings in programming language design and theory was quite advanced. The *C Programming Language* was born in this context and determined a vast transformation in developing system programs. Different agencies implemented this language in early moments and a lot of companies are using this language extensively today. Strength of this programming language are about the very well designed abstractions it provides. *Data abstractions* is very simple and elegant and permits to specify memory layout and alignment for objects of every kind. *Procedural abstraction* is clean and fully reflect the underlying stack-based machine usually implemented in modern systems. The language is statically typed with bit arithmetic, common control flow and casting features. The most characteristic, though, is the pointer arithmetic which permits to represent expression whose value is not a direct state (an object), but rather a pointer to some object. This gives the possibility to carefully design powerful **indirections**, thus achieving good performance boosts.

In the following we provide a summary of the experience we had while developing a driver for the Linux kernel.

6.1 Development Setup

In this section we illustrate the configuration we adopted to ease the development process. As previously explained the usual embedded software development style sees two

systems: target and host. There are different methodologies to connect these systems and some are very effective and a consistent time has to be spent in properly configure these two components. As the development process is done through a lot of repetitive tasks, such as re-compile, re-load and re-execute the code to see if it behaves well, speeding up the single phase can really increase the total amount of tests conducted and lines of code written, thus permitting to be more efficient.

6.1.1 Cross Compiling

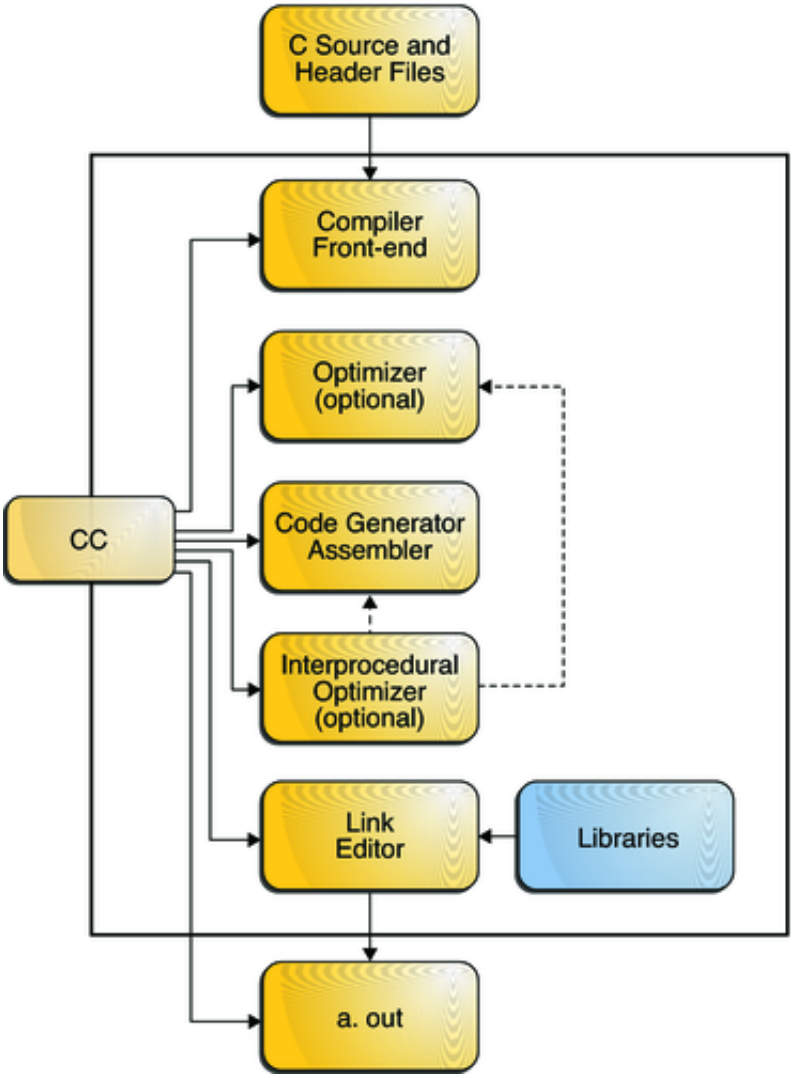


Figure 6.1: Compiler phases for a C program

The term *cross compiling* denote a particular configuration in which binaries are produced in a machine that is architecturally different from the one used to actually run

the software. In Figure 6.1 we can see how a compiler achieve the translation of a C source program into a binary able to run on the given target machine. Among all the phases a compiler performs, we give a short explanation only for what concern the cross compiling thematic.

Front-end This is the part of the compiler that read the source code and interpret the information included. The process is called **parsing** and produces as final output a tree-like data structure suitable for implementing error detection and translation into an assembly-like language used by later phases. This phase can be architectural independent.

Code generation Such phase actually produces the list of instructions that will run on the machine. So, it is machine dependent. At this point a lot of code transformation are performed in order to optimize some resource utilization, such as memory utilization or number of instructions executed.

Linking The process of linking is about actually making the binary run in the final environment, so this phase is not only dependant upon the memory configuration (segments), but also on the operative system that will eventually load and run that code. This is sometimes called **runtime support**. The linker is also responsible for integrating in the final binary information included in libraries. There are basically two types of inclusions: **static** and **dynamic**. The latter is more common because achieves better memory re-utilization and in the Linux context is usually called the *shared libraries (.so)* mechanism.

The toolchain has to be properly configured to produce the right executable. In the recent years a lot of tools have been created in order to ease the development process for embedded applications. For our experiments we used the *Linaro Toolchain*. This is based on famous GNU tools: *gcc, ld, objdump, ar, as* and also the debugger *gdb*. The Linaro Toolchain has shown to be a very useful since it is able to produce binary optimized for a specific ARM machine architecture, such as little-endian or big-endian, and with the right runtime support. The latter phase is rather difficult to implement because the compiler has to know the version of the kernel the binary will run on and has to link with a set of fundamental system library, as *libc* and others.

6.1.2 Serial Line Console

In this section we explain how to interact with the development board. A very common way to achieve simple communications between host and target machine is with a *serial port*. Although this system is classical, it is widely used as it is very simple and adopted by a wide range of embedded system producers. This interface transfers in or out one bit at a time (in contrast to a parallel port) and can transmit up to 115200 bit/s. Such low rate transmission is not suitable for flashing images or loading other binaries on the target, but can really become useful if on the target there is a software component able to implement a console over this communication link. Both U-Boot and Linux can let

the user interact through a console and this is why we used such connection in the first place.

When the machine is powered on, it loads from a given memory location the image of U-Boot and starts executing. The bootloader initializes the hardware needed for talking on the serial port and starts a console-like driver on it. At this point we are able to send command to U-Boot and receive output back. This is really helpful because we can preempt the machine at a very initial step and instruct it how to load the next components in the bootstrap sequence. We explain how this feature decreased development time in the next section.

On the host, we need the respective components for the interaction to have place. Linux has a lot of tools to talk on serial lines and we chose **minicom**, which is one of the most stable serial protocol agents. The command to use on the host machine is

```
$ minicom -D /dev/ttyUSB0
```

The string `-D /dev/ttyUSB0` instructs minicom to use the special device file for the communication. The Linux kernel loads the specific driver once it sees that there is a new link of a certain kind.

6.1.3 System Setup

One really useful feature provided by U-Boot is to permit network boots. Before explaining the boot process, we have to show what particular images are generated by a kernel build and what is the purpose of each of them. The bootloader has to know where to find this data, implement the protocol for transferring binaries, placing them at the right location and start the kernel.

uImage This contains the kernel code which has been statically built into it. In particular, the format of this binary is tailored to U-Boot specific needs.

dtb Device Tree information, as describe in Chapter 4, contains a description of the hardware where the operating systems will run on. Then, the kernel uses such information to load the specific software for driving the given device, for example a power manager could reside on the main bus and the kernel finds the device model and the register location of this device in the device tree. As we saw, such information is specified with a simple programming language. A translation has to be performed also for the device tree data. The result is a binary with extension **.dtb**.

.ko The Linux kernel support dynamic loadable modules. These modules are built separately from the main image and have extension **.ko** (kernel object). Such binaries have to reside on a filesystem to be loaded at the moment in which the kernel needs them for activating particular features.

rootfs Without userspace application a kernel has nothing to perform. Later in the loading process a kernel usually mount a filesystem and automatically executes

the program `/sbin/init`. Without going into details, such program instructs the kernel which other services to load next in order to run an operating system full of useful services. Other components found on a root filesystem are: `/etc` that contains all the configuration file of programs present on the system, `/home/` for user data, `/usr/bin` for user utilities, such as the famous **bash** and others.

Loading Process

We now provide as illustrated in the Figure 6.2 a diagram to show how we organized the host-target configuration to speed up the development process.

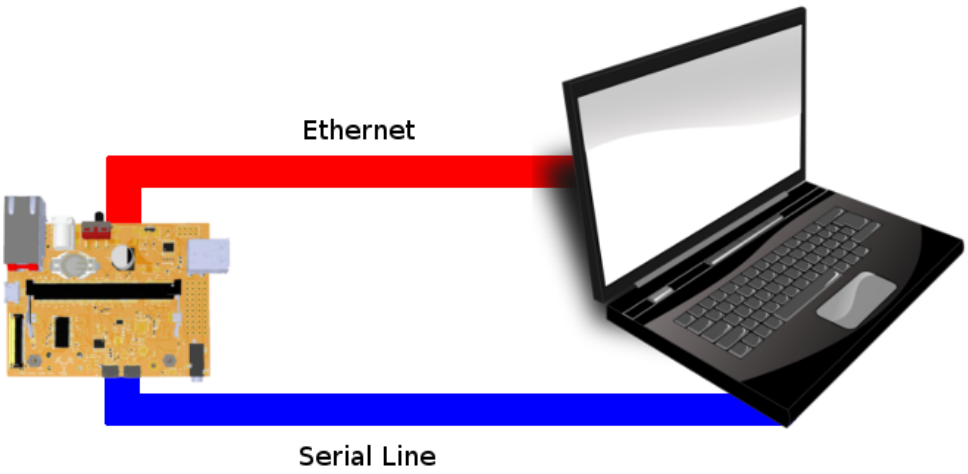


Figure 6.2: Development setup

The steps followed to boot the operating system are listed. During the process the serial line keeps transmitting data related to the console, this way we can receive bootloader and kernel output as they load.

- at the right time the system powers on, the cpu executes the bootloader code that is located in a NAND memory on the board. We had to manual flash this image.
- using the TFTP protocol over the Ethernet link, U-Boot contact the host to retrieve **dtb** files. These files are located on the host's filesystem in a specific shared directory. Data is transferred to the right memory address.
- in the same way the bootloader now receives the kernel image and start the loading procedure specified by Linux.

- when the kernel is ready to start the init process, through the **NFS** protocol in execution on the Ethernet link, it tries to mount the network filesystem and then execute the file `/sbin/init`. On the host computer the NFS daemon is running while this happens.

In order to achieve this, U-Boot has to be instructed to perform the right operations on right file names. This parameters can be specified in a sort of internal environment in which the programmer can set the booting protocol to use when loading images, in our case that is called TFTP, and also the location and names of the file in the shared folder. Given this shell scripting-like feature the user can issue a reset by pressing the reset button and the system execute the points highlighted before automatically. In fact, with this method no image is copied by hand and all the information is located on the host's filesystem where the building process places its output. This way permits to modify the kernel, compile it into an image and, by just pressing the reset button, bootstrap the already built system. Compared to the manual operation this strategy is definitively more efficient.

Another useful feature is to have the filesystem residing on the host machine, as for the kernel image. With the use of the NFS protocol we can share files between the host and the target system. Even though this makes file transfer for development related stuff easy, the most important feature is to have **.ko** file on a filesystem mounted simultaneously on the target and the host. Instead of compiling the driver code statically inside the kernel, we instructed the kernel build process to build it as a loadable module. As this operation is much faster then building the big kernel image, driver modifications can be uploaded to the target kernel with two simple short operations:

```
/path/on/host-$ amb-make
```

and

```
/on/the/target-$ modprobe cc120x
```

The first command, which is a script that simply sequences other commands, build driver's modules using the host machine and move the resulting .ko files in the shared folder, while the second instructs the target system to load them into the kernel. This further accelerates code development, being almost as fast as if the building process addressed the local machine.

6.2 Driver Development

In this section we provide an overview of the driver development process. Since the Linux kernel wants to carefully organize the driver model architecture, for the purpose of code reuse and maintainability, it tends to provide a common interface for module initialization and registration in almost all subsystem. The protocol ideas is illustrated in the following listing (taken from the kernel source code and simplified).

```
/*
 * Represents the operations a driver has to implement
 * for being properly attached to a particular subsystem.
 */
struct ieee802154_ops {
   int             (*start)(struct ieee802154_hw *hw);
   void            (*stop)(struct ieee802154_hw *hw);
   int             (*xmit_sync)(struct ieee802154_hw *hw, struct sk_buff *skb);
   int             (*xmit_async)(struct ieee802154_hw *hw, struct sk_buff *skb);
   int             (*set_channel)(struct ieee802154_hw *hw, u8 page,

   (...)
};

/*
 * Static variable for specifying what function should be
 * called in a particular situation.
 */
static const struct ieee802154_ops mydrv_ops = {
   .start = mydrv_start,
   .stop = mydrv_stop,
   .xmit_sync = mydrv_tx,
   .ed = mydrv_ed,
   .set_channel = mydrv_set_channel,

   (...)
};


/* Example of a subsystem registration */
{
   int ret;
   ieee802154_hw desc;

   // memory allocation
   desc = ieee802154_alloc_hw(sizeof(*priv), &mydrv_ops);
   if (!desc)
      goto error;

   // then the real subsystem registration
   ret = ieee802154_register_hw(desc);
   if (ret)
```

```
        goto error;
}
```

The model suggests that, in order to implement a driver for a device that provides functionality needed by a particular subsystem (in our example is the Linux IEEE 802.15.4 support subsystem), the developer has to define all necessary callbacks and pass them to the registration procedure, contained in a suitable data structure.

6.2.1 Kernel Subsystems

In our development process we used two Linux subsystems. The following gives a small overview, extracted from kernel documentation files.

Serial Peripheral Interface

The *Serial Peripheral Interface SPI* is a synchronous four wire serial link used to connect microcontrollers to sensors, memory, and peripherals. It's a simple "de facto" standard not complicated enough to acquire a standardization body, very general and usable in different contexts. In contrast with classical serial ports, which are asynchronous, SPI uses separate lines for data and clock thus keeping both sides in perfect sync. SPI devices communicate in full duplex mode using a master-slave architecture with a single master originating the frame for reading and writing. Multiple slave devices are supported through selection with individual slave select (SS) lines.
The bus is composed of four logical signals:

CLK Serial Clock (output from master)

MOSI Master Output, Slave Input (output from master)

MISO Master Input, Slave Output (output from slave)

SS Slave Select (active low, output from master)

Linux has a very good support for this low-level communication protocol. For example, it provides full support of master devices: it abstracts common configuration operations, such as setting the *clock frequency* and the *clock modes* which establishes the signal edge to use for syncing transmitted bits. Next in the driver explanation, we show how to send SPI messages to a device.

IEEE 802.15.4 Subsystem

This layer wants to implement the IEEE 802.15.4 standard. The network stack is composed of three main parts:

- **socket API** the generic Linux networking stack to transfer IEEE 802.15.4 messages and a special protocol over *genetlink* for configuration/management

- **MAC** provides access to shared channel and reliable data delivery
- **PHY** represents device drivers

To be more general as possible there are two types of ieee802154 devices. **Hard-MAC**, where the entire MAC layer is implemented in the device itself. The developer has to implement a particular interface for the kernel to properly include the MAC layer inside the network subsystem. Data is exchanged with socket family code via plain *sk_buffs*. The other type is the **SoftMAC**, meaning that the MAC layer is implemented inside the kernel. This is the interface we had to write for, because even though the CC1200 provides support for packet handling and cryptography it does not implement advanced data link features.

6.2.2 Driver Operations

In this section we highlight some important operation the driver code has to perform.

Initialization and Probing The module system has to know what *init* function to call upon module loading. Every module must define how to initialize its resources and how to free them. Following the previous driver model protocol this is done by defining two callbacks

```
static const struct of_device_id cc120x_of_ids[] = {
   {.compatible = "ti,cc120x", },
   {},
};

MODULE_DEVICE_TABLE(of, cc120x_of_ids);

static struct spi_driver cc120x_driver = {
   .driver = {
   .name = "cc120x",
   .bus = &spi_bus_type,
   .owner = THIS_MODULE,
   .of_match_table = of_match_ptr(cc120x_of_ids),
   },
   .probe = cc120x_probe,
   .remove = cc120x_remove,
};

module_spi_driver(cc120x_driver);

MODULE_AUTHOR("MOTT SIRT");
MODULE_DESCRIPTION("CC120X Transceiver Class Driver");
```

`cc120x_probe` and `cc120x_remove` serves this scope. Another important point to note from the listing is that the string `.compatible = "ti,cc120x"` is what is matched when the kernel load a flattened device tree with that device name.

Registration in Ieee802154 Subsystem The registration is quite simple. Plus the usual callback mechanisms the ieee802154 layer permits a lot of configurations to be passed to. Such parameters has to be set prior the registration call and can specify the type of encryption supported, hardware capabilities and channels at disposal.

Interrupts Handling The CC1200 device has the feature of automatically signaling when an internal event requires the intervention of the CPU, such as a packet has been correctly received. Linux permits to request that when a GPIO input pin detects a positive edge the internal CPU interrupt is routed to call the driver specific function. This way it is possible to be preempted by the transceiver. A point worth noting is the fact that the interrupt service function specified to handle the external stimulus, should be as short as possible because the processor while executing such function remains in a non-preemptable state and this period has to last for small times. In Linux this is resolved by writing a very small ISR driver function that actually instructs an internal work queue to perform the actual interrupt handling as soon as possible, thus leaving non-preemptable state in very short time.

SPI Messages The SPI interface is very general and permit to have full control over the content of messages transmitted on the link.

```
/* DATA DECLARATION */
u8 *cmd_buf;  //non initialized
u8 *data_buf; //
struct spi_message msg;
struct spi_transfer cmd_xfer = {
   .len = 1,
   .tx_buf = cmd_buf,
};
struct spi_transfer data_xfer = {
   .len = 4,
   .rx_buf = data_buf,
};

/* MESSAGE CONSTRUCTION */
spi_message_init(&msg);
spi_message_add_tail(&cmd_xfer, &msg);
spi_message_add_tail(&data_xfer, &msg);

/* MESSAGE SUBMIT */
```

```
status = spi_sync(priv->spi, &msg);
```

From the listing is clear that the interface is used following these steps: 1) Data source declarations should be linked within specific structures, 2) The message is allocated and assembled 3) The message is sent over the link.

Debug A useful feature we have used is enabling the driver to send debug messages to special buffers inside the kernel. These are used to inspect system behaviour when something happens. It is sufficient to call the function

```
dev_dbg(&priv->spi->dev, "spi status = %d\n", status);
```

6.3 Tests and Experiments

In order to test the driver, we used the suite *wpan-tools* that provide two tools for interacting with network interfaces and ieee802154 devices.

iwpan uses the netlink interface to access ieee802154 layer specific informations about devices found on the system. Typical uses are: setting PAN identifier, node identifier, list available channels/pages, enable or disable channels, etc.

wpan-ping implements a ping-like loop at MAC layer. We used this command to compute round trip time measures.

Chapter 7

Conclusions

The main thematic of this thesis is to create a link between this new trend of Internet of Things and the development process that is necessary to implement modern smart objects networks. The process is really complex and entail issues from different disciplines. We showed some aspect of protocol design and which data link features and physical layer features are well suited for smart metering utility networks. Devices for IoT system also present peculiar characteristics that most of the time regard embedded system with low complexity and low-power consumption, with the ability to last for decades. Such feature, without a good software support is not easy to obtain, though.

The study has been conducted on the characteristics that an operating system should have in order to be a good candidate for IoT-oriented embedded systems.

- a wide support of different hardware architectures and devices
- good flexibility and configuration capabilities, in term of selected software components that will be part of the final binary
- modern networking subsystem, adaptable to different protocol needs and customizable to implement adhoc signaling and interface management

Such traits are full matched with a community developed software. The wide selection of devices supported is because a lot of different developers around the world want to port a free and open source software on their hardware. The model is pretty simple: a developer can access for free to a code base that is the result of the efforts of thousands of other people and a lot of development time, so it should probably be a useful piece of software. He/she can use such software to achieve his/her needs, by respecting the license, and then return to the community the precious integration of drivers he/she might have developed to complete his/her task, for instance. Those are usually difficult to implement because one person has to lay a hand on the device which sometimes can cost hundreds of dollars. With the same reasoning also the second point is guaranteed, because a multitude of persons often end up having totally different objectives. This assure that the internal code is maintained with a re-usability goal in mind, thus keeping system configuration, building and internal subsystems extremely adaptable. However,

an opposite effect of being involved in a open source community, such as Linux, that some people or even companies may dislike, is the fact of being auto regulated projects. It is true that some people have the power of taking important decision on the design or on the community behaviour, but generally speaking the usual open source software organization is quite loose, from the management point of view. This, for example, can entail the lack of a well structured documentation or even a long learning process for people who wants to be able to understand and to make full use of software features.

Internet of Things is a trend that is expanding at an incredible pace. Such process will likely transform the world we live as this technology will enter every aspects of our life. This is also true from an open source community point of view. The space for implementation and community growth remains vast, as every innovation in the Information Technology sector opens new horizons. The question we leave for further reasoning is how the open source approach will respond to this imminent breakthrough innovation. Is it possible that one day Linux will power every IoT device on earth? How will this impact with the industry? How will the development of IoT systems be, easier or even more complex?

Bibliography

[1] Karen Rose, Scott Eldridge, Lyman Chapin, *The Internet of Things: An Overview Understanding the Issues and Challenges of a More Connected World*, Internet Society, October 2015.
`http://www.internetsociety.org/doc/iot-overview`

[2] H. Tschofenig, J. Arkko, D. Thaler, D. McPherson *Architectural Considerations in Smart Object Networking*, Internet Architecture Board (IAB), March 2015.
`http://www.rfc-editor.org/info/rfc7452`

[3] IEEE Standard for Local and metropolitan area networks, *Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*, IEEE-SA Standards Board September 2011.

[4] IEEE Standard for Local and metropolitan area networks, *Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs), Amendment 3: Physical Layer (PHY) Specifications for Low-Data-Rate, Wireless, Smart Metering Utility Networks*, IEEE-SA Standards Board April 2012.

[5] N. Kushalnagar, G. Montenegro, C. Schumacher, *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals*, IETF Network Working Group August 2007.

[6] G. Montenegro, N. Kushalnagar, J. Hui, D. Culler, *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*, IETF Network Working Group September 2007.

[7] Andrew Tanenbaum, Herbert Bos *Modern Operating Systems*, O'Reilly Media, 4st edition, March 2014.

[8] Greg Kroah-Hartman, *Linux Kernel in a Nutshell*, O'Reilly Media, 1st edition, 2006.
`http://www.kroah.com/lkn/`

[9] linux.org, *The Linux Kernel: The Source Code*, 2013.
`http://www.linux.org/threads/`
`the-linux-kernel-the-source-code.4204/`

[10] tldp.org, *The Linux Kernel*, 1999.
http://en.tldp.org/LDP/tlk/tlk.html

[11] *Kernel source tarball's documentation subdirectory.*
https://www.kernel.org/doc/Documentation/

[12] *AMBER Website.*
http://www.amber-lab.com/

[13] Texas Instruments *CC120X Low-Power High Performance Sub-1 GHz RF Transceivers Users Guide*, September 2014, http://www.ti.com/product/CC1200/technicaldocuments.