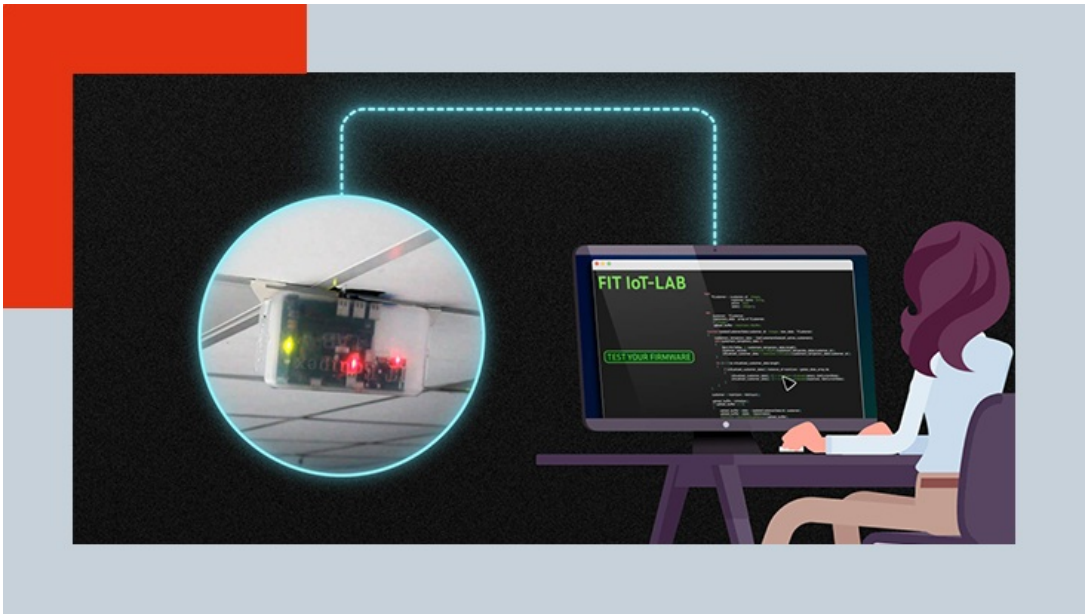


Internet of Things with Microcontrollers: a hands-on course
-
L'Internet des Objets sur microcontrôleurs par la pratique

Session 1: March 30 - May 30, 2020

Alexandre Abadie, *Research engineer, Inria, Saclay*
Emmanuel Baccelli, *Professor, Freie Universität, Berlin*
Antoine Gallais, *Professor, U. Polytechnique Hauts-de-France*
Olivier Gladin, *Research engineer, Inria, Saclay*
Nathalie Mitton, *Researcher, Inria, Lille*
Frédéric Saint-Marcel, *Research engineer, Inria, Grenoble*
Guillaume Schreiner, *Research engineer, CNRS, Strasbourg*
Julien Vandaële, *Research engineer, Inria, Lille*

Course on FUN-MOOC produced by Inria Learning Lab.
The contents of this document are extracted from:
https://www.fun-mooc.fr/courses/course-v1:inria+41020+session01/about and provided under Licence CC-BY-NC.

Module 1. Internet of Things: General Presentation

Objective: At the end of this module you will be able to describe the IoT system from the device to the cloud.

Hands-on activities (TP): TP1 : Welcome to the IoT-LAB testbed training

Contents of Module 1

1.1. Introduction to the Internet of Things

- 1.1.0. Introduction
- 1.1.1. What is the Internet of Things?
- 1.1.2. The architecture of IoT solutions
- 1.1.3. The challenges facing the Internet of Things

1.2. Which Radio Technology for Which Application?

- 1.2.0 Introduction
- 1.2.1 Frequency bands
- 1.2.2 Specifications
- 1.2.3 Technologies
- 1.2.4 Overview and comparative assessment

1.3. Examples of IoT Applications

- 1.3.0. Introduction
- 1.3.1. Smart watches
- 1.3.2. Sensors for connected agriculture
- 1.3.3. Sensor networks in smart buildings

1.4. FIT IoT-Lab Usage

- 1.4.0 Introduction
- 1.4.1 Why have such a platform?
- 1.4.2 What does FIT IoT-LAB offer?
- 1.4.3 How to use FIT IoT-LAB?
- 1.4.4 The use of FIT IoT-LAB in this MOOC
- TP1. Welcome to the IoT-LAB testbed training

1.1. Introduction to the Internet of Things

- 1.1.0. Introduction
- 1.1.1. What is the Internet of Things?
- 1.1.2. The architecture of IoT solutions
- 1.1.3. The challenges facing the Internet of Things

1.1.1. What is the Internet of Things?

The Internet of Things is a revolutionary development in the digital world. But what actually is it? According to the International Telecommunication Union, the Internet of Things (IoT) is a 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." A simpler definition would be to say that it is a set of connected physical objects with digital identities which are capable of communicating with each other. These objects form a gateway between the physical world and the virtual world. The IoT begins in the physical world with sensors used to measure data (temperature, atmospheric pressure, brightness, etc.), which is then transmitted to the internet via the connection and integration of the systems with each other. This data is then processed and stored in order for it to be analysed and used. The IoT is evolving rapidly - forecasts indicate there will be somewhere in the region of 18 billion IoT objects by 2022 (among almost 29 billions of connected objects) [1]. We are witnessing the mass deployment of a set of interconnected objects equipped with communication, detection and activation capacities, which can be used for a range of different applications. IoT is used everywhere, with some examples including:

- smart cities: traffic management and security in public spaces
- connected agriculture: medical surveillance on animals and soil quality monitoring
- industry of the future: equipment surveillance and predictive maintenance
- logistics: container tracking in maritime transport

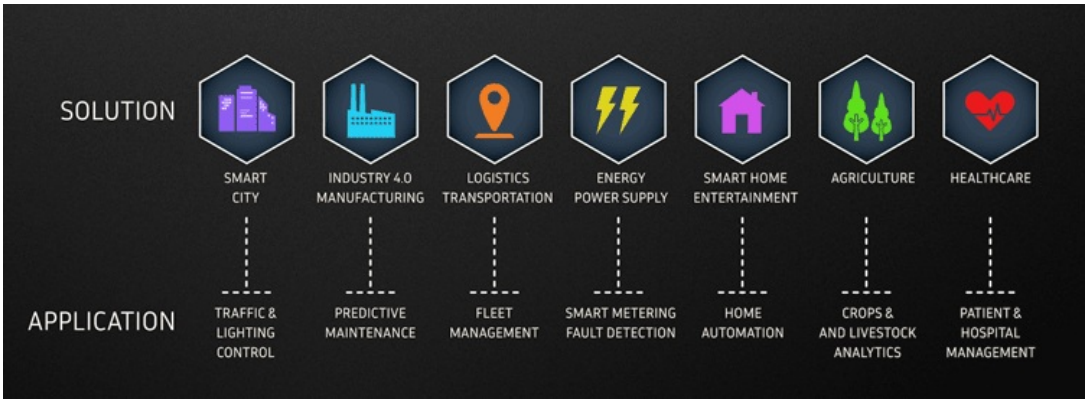


Fig. 1: IoT application domains

[1]https://www.ericsson.com/en/mobility-report/internet-of-things-forecast

1.1.2. The architecture of IoT solutions

Content

1.1.2a Introduction

IoT solutions typically implement **constrained objects or devices** (i.e. things) that pass their data through **gateways** through the network to reach servers that host **IoT platforms** and to which various applications are connected. We will therefore define these 3 layers.

Given the rapid development of the Internet of Things, it soon became vital to determine a standard architecture for IoT solutions. It was with this in mind that the Internet Architecture Board (IAB) published RFC 7542[1]. This document describes four models of interactions common to the architecture of all IoT solutions:

- communication between objects: this model is based on wireless communication between two constrained objects.
- communication between objects and IoT platforms: in this model, data measured by sensors is sent to IoT platforms via a network.
- the communication of objects with gateways: this model enables data to be sent from sensors to IoT platforms via an intermediary.
- back-end data-sharing: this model is used to share data between several IoT solutions. It is based on the "programmable web" concept with APIs.

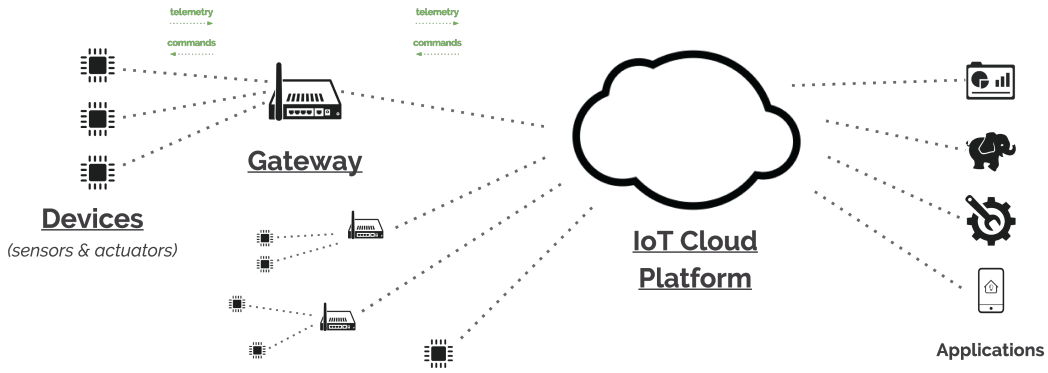


Fig. 1: IoT solutions architecture[2]

1.1.2b Constrained objects

Objects, whether these are sensors used to feed back physical data or actuators to act on the physical environment, are the basis of IoT solutions. These objects are where data come from. Generally speaking, these objects are limited in terms of their size and are battery-operated, reducing their level of autonomy. They operate using microcontrollers (MCUs), which also limits their processing capacities.

The role of these objects is to support specific tasks. They may be comprised of the following in terms of software:

1. An operating system for the IoT, tailored specifically for constrained objects and embedded systems.
2. A hardware abstraction layer (HAL), simplifying access to the features of the MCU: flash memory, GPIOs, serial interface, etc.
3. A communication layer, enabling the object to communicate via a wired or wireless protocol such as Bluetooth, Z-Wave, Thread, CAN bus, MQTT, CoAP, etc.
4. Remote management, which can be used to control objects, to update firmware or to monitor battery levels.

1.1.2c Gateways

A gateway is a relay point between a network of sensors and actuators and an external network. These may rely upon specific materials or extra features added to certain constrained objects.

Gateways can offer capacities for processing data and storage on the periphery of constrained objects (what's known as *Edge Computing*), the goal being to tackle any problems linked to latency or network reliability. When it comes to connectivity between constrained objects, gateways will also be responsible for dealing with issues of interoperability.

1.1.2d IoT platforms

IoT platforms are the software solutions and services required in order to implement IoT solutions. These are run on servers hosted either in the cloud (such as AWS, Google Cloud, Microsoft Azure, etc.) or within the company's own infrastructure. These solutions must be capable of adapting to scaling up at both a horizontal level (the number of connected objects) and a vertical level (the variety of the solutions offered). IoT platforms enable interconnectivity with companies' existing information systems.

The key features of these platforms are as follows:

1. Connectivity and message routing
2. Registering and managing objects
3. Data storage and management
4. Incident management, data analysis and representation
5. Using an API to integrate applications

In this MOOC, we will be focusing on these first two layers: **constrained objects and gateways**.

- From a **hardware** perspective, we will explore the field of embedded systems, explaining what constrained objects are comprised of: microcontrollers, sensors/actuators, data buses and radio chips.
- From a **software** perspective, we will take a look at what solutions there are for programming constrained objects, before taking a closer look at an example with the embedded operating system RIOT.
- From a **network** perspective, we will present the different wireless communication standards and different network layers currently available for the IoT.
- From a **security** perspective, we will focus on the mechanisms that can be employed in order to guard against network or software attacks.

For each of these, you will see how solutions adapt to the limited capacities of the materials and

tackle the issue of energy consumption.
References:

[1] Architectural Considerations in Smart Object Networking:https://tools.ietf.org/html/rfc7452

[2] The illustration comes from a white paper published by the Eclipse IoT Working Group, entitled The Three Software Stacks Required for IoT Architectures Feel free to read this document, which provides a more detailed description of the software architectures in these 3 layers and the existing Open Source solutions._

1.1.3. The challenges facing the Internet of Things

Content

As we have already seen, the Internet of Things can be found in a wide range of applications, whether in industry or in our daily lives, and in order to meet users' expectations, there are a number of challenges it must face.

1.1.3a The interoperability of IoT solutions

The past few years have seen the emergence of both small and large stakeholders in the connected objects sector. The current giants of the internet in particular are engaged in a technological arms race in order to ensure that as many connected objects as possible use their solutions. Examples of this include Amazon's AWS IoT platform, Microsoft's Azure IoT or Google's Google Cloud IoT. Using Google/Amazon/Microsoft cloud technology is practical because all the necessary software building blocks are there in one place, sometimes from the object up to the cloud, enabling a comprehensive IoT application: data management servers, data analysis with machine learning algorithms, hosting for online virtualisation applications, etc.

Unfortunately, this development has seen an increase in IoT solutions operating in *silos*. Essentially, users of a given application find themselves captive, to a certain extent, i.e. the objects and the data which they produce can only be managed through this application. Under normal circumstances, it is not possible to reuse an object that has been programmed for one application with another application.

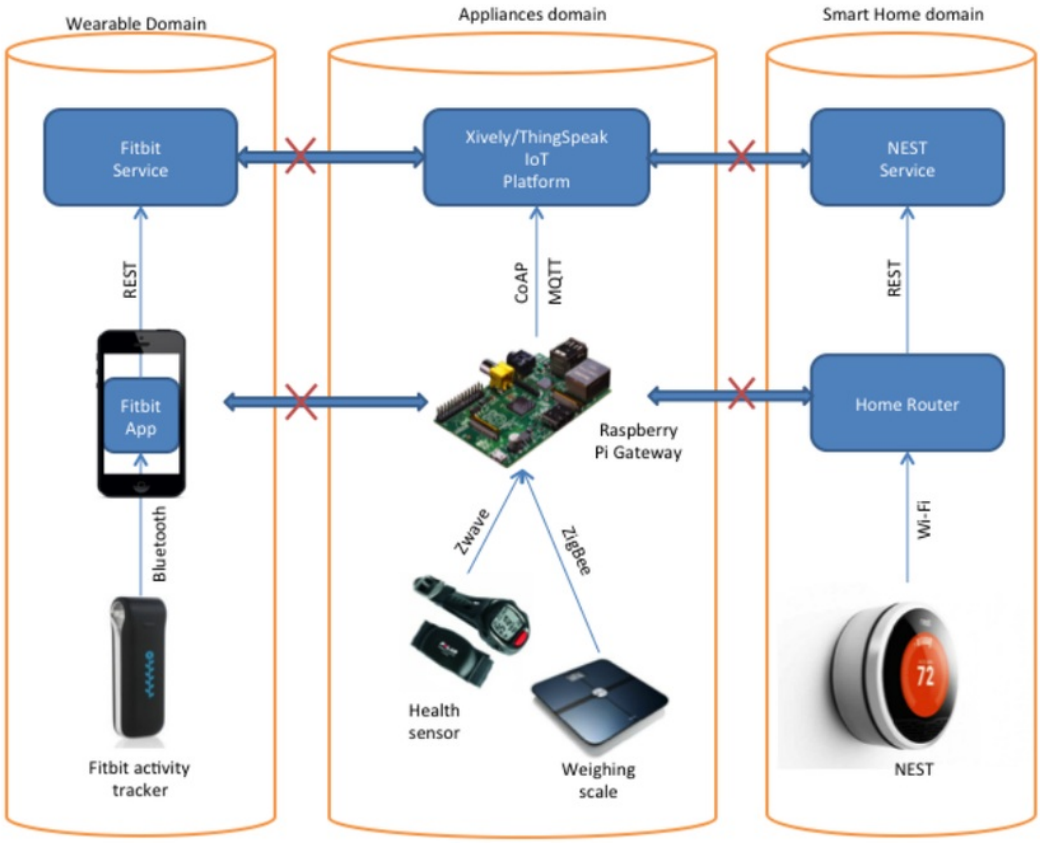


Fig. 1: Vertical silos of IoT service deployment.

Source: P. Desai, A. Sheth and P. Anantharam, "Semantic Gateway as a Service Architecture for IoT Interoperability," 2015 IEEE International Conference on Mobile Services, New York, NY, 2015, pp. 313-319.

Another issue with these silos is the lack of standards used in relation to communication protocols.

When it comes to solving the problem of the interoperability of IoT solutions, one of the major challenges involves finding a way of developing the use of communications standards. A few already exist, the majority of which have been proposed by the IETF (*Internet Engineering Task Force*), an international standards body. These standards are applicable to sensor networks in particular, as we will see in this MOOC.

Other specifications have been pushed by certain companies in order to promote their technology. LoRaWAN, for example, did this with the LoRa Alliance, an organisation made up of various different stakeholders from the IoT world: telecommunications operators, manufacturers of objects, manufacturers of chips, etc.

Another example is the ThreadGroup, which promotes specifications for sensor networks. These ThreadGroup specifications reuse certain IETF standards.

1.1.3b Hardware constraints

Another challenge relates to the hardware aspect of the objects themselves, particularly those

which operate using microcontrollers. The inherent characteristics of these platforms are highly restrictive: a few dozen kilobytes of RAM, flash memory of around one hundred kilobytes. Despite this, they must be capable of operating increasingly complex applications - communication capacities, security, updates - while being increasingly more intelligent. This problem can be solved by using adapted software and multiple optimisations in order to keep these applications going on microcontrollers.

Another aspect of the challenge linked to hardware constraints is how energy is managed on connected objects. Indeed, the majority of objects deployed in IOT applications must be capable of running on batteries for as long as possible. Microcontrollers have a number of different operating levels, meaning they consume very little energy. We generally refer to sleep mode. Limiting consumption and saving as much battery life as possible means getting the most out of this feature. Given that it is first and foremost electronic, it is also vital to ensure that the hardware design does not lead to any current leakage. All this has to be done while ensuring that increasingly complex applications are able to function.

1.1.3c Storing, processing and transporting data

The deployment of a growing number of connected objects (estimates suggest the number will rise to 38 billion connected objects by 2025 1) will produce vast quantities of data (some estimates suggest there could be 175 billion terabytes of data by 2025 2).

Most importantly, it must be possible to store all of this data in a way that is both reliable and sustainable, such as in databases, for example. The experience acquired since the early days of computing on these subjects, particularly on database management systems, should make it possible to tackle this issue. However, the volumes could be so large that it may prove necessary for existing solutions to be improved.

The Internet of Things must meet the need of rapid access to information. This information can't always be taken directly from the data itself: in many cases the data has to be processed and cross-checked in order to extract high-quality, usable information. This processing can sometimes be complex, requiring substantial computational capacities. Today, *cloud computing* platforms make it easy to access these processing resources (processing servers, GPU cards, etc.) and should make it possible to produce processing results in real-time.

Lastly, there are better ways of distributing processing resources among the different components of the IoT chain: the most intensive processing takes place in the *cloud*, less intensive processing can take place at a middle level (on base stations, for example), and the most basic processing can take place on the objects themselves. Distributing resources in this way helps reduce the use of data transport infrastructure (radio, wired technologies), thus increasing the number of objects that can be connected.

1.1.3d The security of connected objects

All sensors deployed in smart buildings, towns or people's homes supply various different types of data (temperature, brightness, pressure, acceleration, proximity, humidity, etc.), which is often highly sensitive (the presence of individuals, for example). If this data is intercepted, it can be exploited by malevolent third parties, causing a whole host of problems: errors with the application or a total shut-down; physical harm caused to individuals, industrial espionage with a view towards gaining a competitive advantage, etc.

Protecting this data is a major concern for Internet of Things applications if its long-term development is to be guaranteed. This protection can take place on a number of levels:

- at the level of objects communicating with each other
- at the level of the applications on the objects themselves
- at the level of the hardware of the object

The complexity of attacks (and the relevant protection) will vary depending on the level of protection (communication, software materials, etc.)

This aspect will be covered in greater detail in module 5, and solutions will be addressed in order to put protective mechanisms in place.

1.1.3e Societal problems

Aside from technical aspects, the Internet of Things is also faced with societal and environmental issues. Chief among these are the consequences of the deployment of IoT solutions on everyday life. Connected objects are becoming increasingly prevalent in all aspects of our everyday lives.

Certain users, for example, through a lack of awareness of how these objects work (voice assistants, home automation systems), don't realise that they are sharing personal and sometimes private information. Meanwhile, the deployment of IoT solutions in urban areas with the goal of making towns and cities smarter can feed back information on the use of infrastructure by residents. These systems are designed to improve the management of transport and infrastructure, to save money and to improve community life. However, even if the initial goals were laudable and worthwhile and the data supposedly anonymous, there is nothing to guarantee that the way in which information is corroborated won't deanonymise this data. There are, therefore, privacy issues linked to the Internet of Things.

From an environmental point of view, meanwhile, the Internet of Things could have a serious negative impact. As we mentioned previously, it is expected that billions of connected objects will be deployed in the near future. What will be the environmental impact of all these objects once they are no longer fit for use? Will it be possible to repurpose them instead of producing new ones?

https://innovationatwork.ieee.org/iot-environmental-impact-ieee-wake-up-radio/
https://www.itproportal.com/features/reducing-the-environmental-impact-of-global-iot/

1.2. Which Radio Technology for Which Application?

- 1.2.0 Introduction
- 1.2.1 Frequency bands
- 1.2.2 Specifications
- 1.2.3 Technologies
- 1.2.4 Overview and comparative assessment

1.2.0. Introduction

There are many different types of radio technology, each with their own advantages and disadvantages and each better suited to specific applications. With that in mind, how can we know which application to choose?

When it comes to selecting the most appropriate type of radio technology, we first need to understand some of the characteristics of radio technology, such as the frequencies they use, the speeds they offer, the ranges they provide, etc. This is what we will try to show in this sequence for different application needs.

1.2.1. Frequency bands

Content

What the different types of radio technology have in common is that they all use radio frequencies. The radio spectrum is highly regulated and the majority of frequency ranges are either reserved (police, army, aeronautics, meteorology, etc.) or licensed (mobile operators, etc.). In France, the ARCEP (the French Electronic Communications and Postal Regulatory Authority) is in charge of these regulations. A breakdown of the radio spectrum frequencies in Franceis available to view on the French National Frequency Agency (ANFR) website. These allocations change from one country to the next. [^1]: Spectrum is allocated in an international agreement. This is discussed during periodic CMR / WRC conferences (the last held in October 2019) in which each country can assign independently frequencies assigned to a usage.

There are, however, a number of frequency ranges which are free to use. These are called ISM bands (Industry, Science and Medical).

1.2.1a ISM bands

There are several ISM frequency bands across the spectrum, the most used being:

- 2.4 GHz (Wi-Fi, Bluetooth, microwaves, etc.)
- 5 GHz (Wi-Fi)
- 433 MHz (baby monitors, alarms, home automation, LoRA, etc.), 868 MHz (home assistants, SigFox, LoRA, etc.)

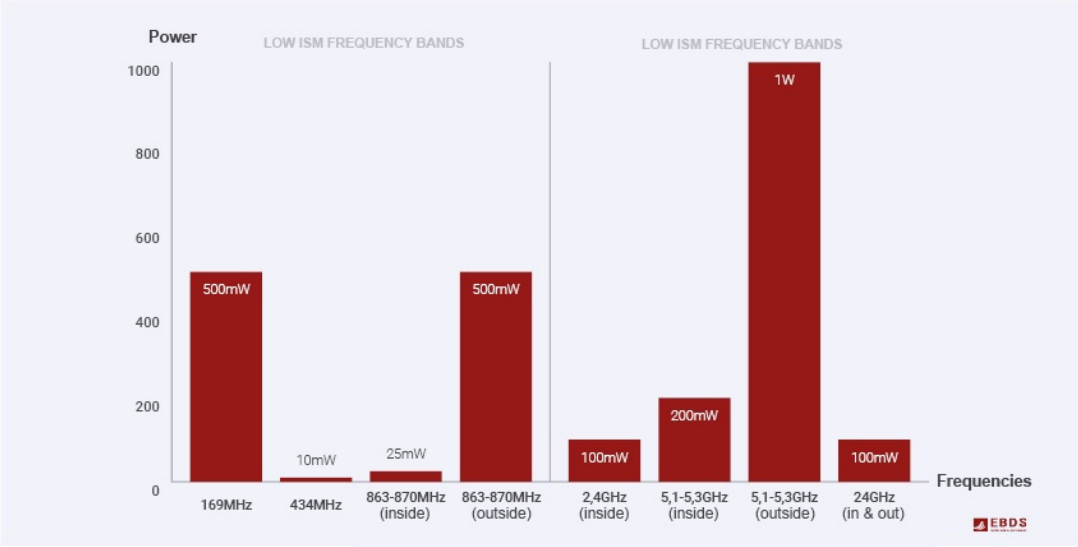


Fig. 1: ISM radio bands (Source : EBDS Wireless & Antennas - http://www.ebds.eu)

The advantage of ISM bands is that they are free to use. They are subject to regulations, however, in terms of their emitting power and usage time, both of which will vary depending on the band being used. For the low band, for example, the emitting time is limited to 1% of the time per hour, or 36 seconds.

The disadvantage of ISM bands is that they are in high demand. Two neighbouring appliances communicating at the same time on the same frequency, even when using different protocols (such as Bluetooth and Wi-Fi), may interfere with each other. Frequencies are a rare shared resource.

1.2.1b Licensed bands

In order to prevent interference, you must be able to control who is authorised to communicate over the same frequencies, which requires the use of licensed frequency bands. This is what telephone operators do: they buy licenses, giving them exclusive rights to certain bandwidths of the radio spectrum. What this means is that only their customers can use these frequencies (via their SIM cards, which control the use of a given frequency)[^2].

[^2]: When changing operator, it is often necessary to unlock your phone's SIM. This is because it is "locked" on to the frequency of a certain operator. Given that this frequency is not the same as that of the new operator, the telephone has to be authorised to connect to a new frequency range. Operators are then responsible for managing the allocation of resources among customers.

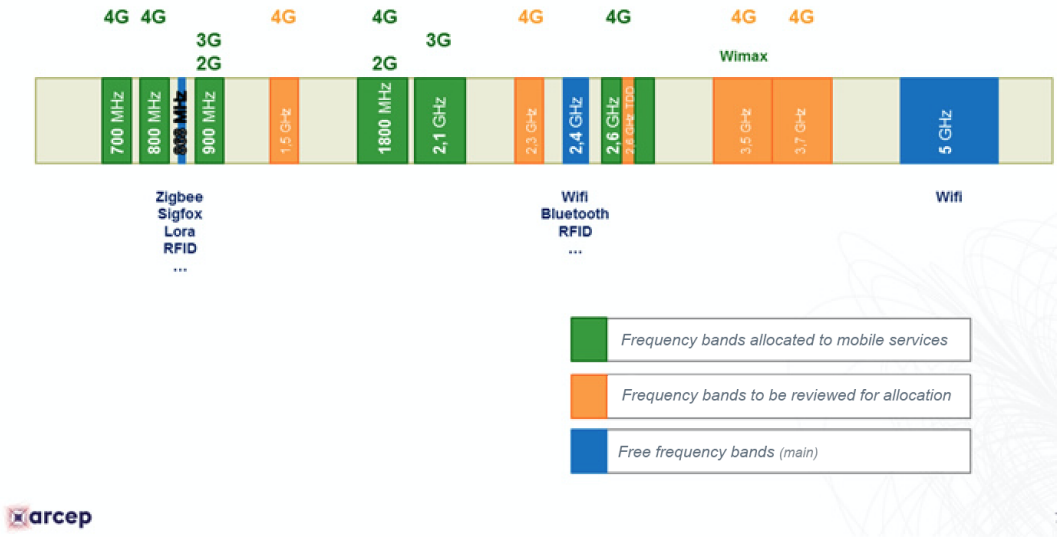


Fig. 2: Frequencies in France

The advantage of going through operators and using licensed frequencies is that these operators are able to guarantee quality of service and a good connection. The disadvantage is that you have to pay tariffs and will be dependent on operators and their infrastructure.

1.2.2. Specifications

Content

The most important aspects to take into consideration when choosing between different types of radio technology are as follows:

- the speed and the quantity of the data to be sent
- the communication range
- energy consumption
- coverage
- reliability

Some of these are linked. Generally speaking, higher frequencies enable greater speeds, but over shorter distances. We detail below these characteristics and cite some technologies that meet them. We will see later in Section 1.3 some applications that use some or other technology.

1.2.2a Speed, data quantity, range, energy

Different types of radio communication technology have different ranges and speeds. These both tend to be linked to the frequency bands they use. The higher the frequency, the shorter the range but the higher the speed. Similarly, the bigger the range or the higher the speed, the more energy will be consumed. The figure below outlines these specifications for the most widely-used types of radio technology.

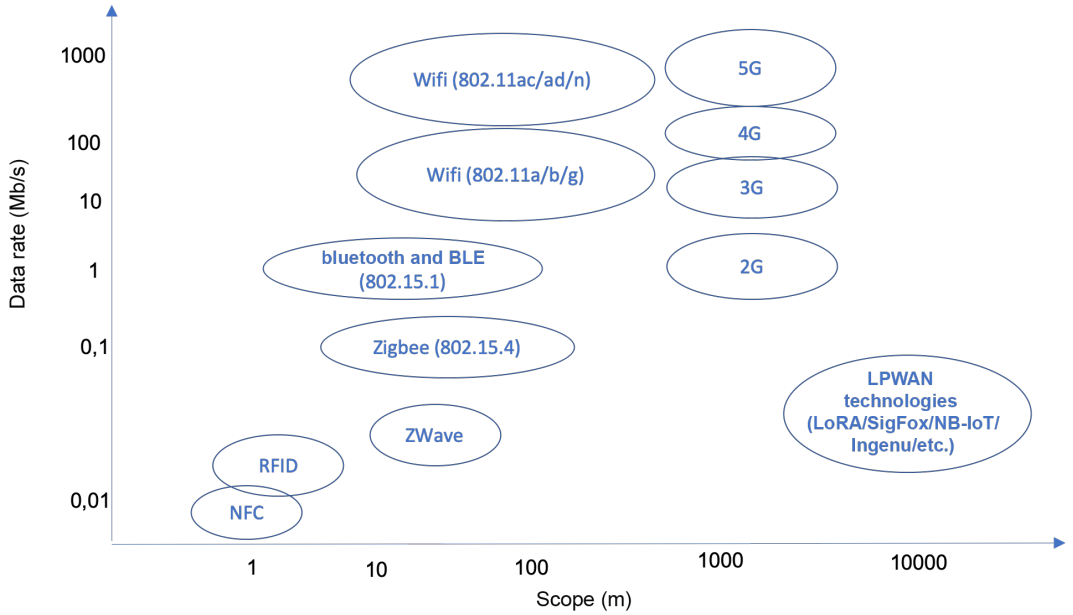


Fig. 1: Speed / Range link

Before making a choice in terms of radio communication technology, you therefore have to consider all these criteria, taking application needs into account. The speed and the range of

communication are both important, but are not the only criteria that should guide your choice. If a communicating appliance is energy self-sufficient (i.e., if it can be recharged often or permanently supplied), then the consumption aspect can be overlooked. On the other hand, if it uses a type of battery that is difficult to recharge or replace, then consumption will be a key factor in determining the choice of technology. Similarly, the quantities of data to be sent must also be taken into account. For high quantities of data (such as video streams, for example), low speed options such as LoRA or SigFox can be ruled out.

1.2.2b Coverage and reliability

As discussed previously, operators control access to communication channels for certain frequency bands, guaranteeing reliable communication. Certain types of technology using ISM bands are also operated. This is the case for SigFox, NB-IoT and, sometimes, Wi-Fi. An operator such as the French operator Orange will also offer you Wi-Fi or LoRA, although individuals are also free to set up their own Wi-Fi or LoRA network.

Going through an operator has considerable advantages, but the zone in which your network is to be deployed must be covered by the operator. This is called coverage. If the cellular infrastructure - LoRA, Wi-Fi, etc. - is not deployed, you will not be able to use its network, and instead will need to consider deploying your own infrastructure or using ad-hoc multi-hop routing.

This will incur an additional expense when it comes to selecting the type of technology. Individual customers will not be able to deploy infrastructure for technology operating on licensed frequencies (like cellular technology). The choice is therefore limited to technology using ISM bands enabling the deployment of new infrastructure, such as Wi-Fi, LoRA or ad-hoc networks (ZigBee, Zwave, etc.).

1.2.3. Technology

Content

There are many different types of radio communication technology. We are not aiming to be exhaustive here, especially as new types are appearing all the time. Instead, we will focus on those most widely used today.

A distinction is made between two different types of radio technology, depending on their range: **short-range** radio technology, which has a range of less than 100 m; and**long-range** radio technology, which has a range in excess of 1 km. Depending on how the technology is to be used, radio range is often a determining factor when it comes to considering either external deployment or deployment in a large building without the need to deploy any heavy infrastructure.

1.2.3a Short-range

Short-range technology is generally used to set up local networks or LANs (Local Area Networks). For the most part, these are associated with high speeds, as is the case with Wi-Fi, for example. Short-range technology is primarily used:

- by internet applications (Wi-Fi)
- for point-to-point information exchange or access authorisation (RFID, NFC, Bluetooth)
- for home automation (ZigBee, ZWave)
- for connected objects (Bluetooth, Wi-Fi, etc.).

Let's take a closer look at some of these technologies.

Wi-Fi

Wi-Fi is a wireless communication protocol governed by the IEEE 802.11 standard. It uses the 2.4 GHz and 5 GHz ISM bands. The first versions, 802.11a and 802.11b, were introduced in the late 1990s, offering theoretical speeds of 11 Mbits/s. The version currently available, 802.11ac, offers theoretical speeds of up to 1.3 Gbits/s. The future version, 802.11ax, will be 20% quicker. Historically linked to laptop computers, Wi-Fi has spread to all communicating mobile devices. Wireless networks operate primarily in stars, linking one station to one point of access. Generally speaking, the point of access is connected to a wired Ethernet backbone for the purposes of routing traffic. Other approaches have involved attempting to build a mesh network type infrastructure on a wireless backbone on a dedicated frequency band (5 GHz) (ref. Roofnet). The main disadvantage of Wi-Fi remains its energy consumption, which means a sizeable battery is required. The 802.11ah standard (2017), which was optimised in order to reduce energy consumption in IoT applications, failed to establish itself in the face of competition from other technology available on the market.

Bluetooth

Like Wi-Fi, Bluetooth also first emerged towards the end of the 1990s. It is overseen by the consortium Bluetooth SIG (Special Interest Group). and was initially designed to link a telephone-type terminal to a computer. 1999 saw the release of version 1.0 and the first compatible phone. In

much the same way as Wi-Fi, Bluetooth uses the 2.4 GHz ISM band, giving it a range of 100 m, but its channels are much narrower, meaning its bandwidth is much lower (less than 1 Mbit/s). There is a potential risk of these two types of radio technology interfering with each other as a result of channel overlap. The standard specifies a number of profiles covering a variety of use cases (modems, file transfer, audio headsets, etc.). Current use of Bluetooth is completely decentralised and does not require any heavy infrastructure. The 4.0 version of Bluetooth, BLE (Bluetooth Low Energy) significantly reduces energy consumption. An increasing number of autonomous connected objects have been adopting this technology in order to be able to interface with mobile terminals. However, connectivity depends on the presence of the radio range between the terminal and the object.

Zigbee

Zigbee is a high level standard based on the IEEE 802.15.4 standard (Low-Rate Wireless PANs), which first appeared in the early 2000s. It was designed to offer a low-speed, low-energy consumption wireless network. In much the same way as Wi-Fi and Bluetooth, Zigbee also operates on the 2.4 GHz ISM band, with a maximum range of 100 m. It is capable of operating on peer-to-peer topologies or star networks. Zigbee is overseen by the Zigbee Alliance, whose role it is to certify that products respect the specifications of the standard. There are a range of different profiles for applications, including home automation (lighting, heating), medical sensors and smoke detectors. The advantage of Zigbee is that it can be used to create a mesh network between objects, extending the surface coverage of the application deployed. Generally speaking, Zigbee/IP gateways can be used for permanent, real-time information exchange between objects and a cloud.

RFID, NFC

RFID (Radio Frequency Identification) is technology used to enable a reader and a tag to exchange information wirelessly. The reader remotely supplies the tag with an electromagnetic wave. The tag then sends a response back, normally in the form of a username stored in a small embedded memory. The reader can also write data into the tag. These tags can have different operating modes - passive, semi-active, or active - and can reach radio ranges between 10 m and 200 m. This technology is mainly used to identify and track objects. The cost of a tag makes it possible to mark objects you wish to follow on a large-scale. The best-known use case is as an anti-theft device attached to items sold in shops. NFC (Near Field Communication) authorises communication over short distances (less than 10 cm). Many smartphones and tablets are equipped with this technology, which can be used for three types of exchange: card emulation (payment, transport), read mode, and peer-to-peer mode. The bandwidth ranges from 106 kbits/s to 424 kbits/s. Secure, short-range exchanges can be used for applications such as bank transactions and for recovering Wi-Fi network keys.

1.2.3b Long-range

Long-range technology is primarily used for two types of application:

- mobile communication and connected services (text messaging, mobile applications, etc.)
- collecting IoT information

These two applications differ chiefly in terms of their speed and QoS requirements (latency, jitter, etc.). Very low latency and jitter[^3] is required for mobile communications, but longer delays are generally tolerated when it comes to relaying environmental data. Some types of LPWAN are

capable of multi-hop routing and setting up mesh networks, while others can only be used to send data to operators' base stations (star networks).

For IoT deployments going beyond building level, long-range radio technology such as NB IoT, Sigfox and LoRA enable scaling-up in terms of radio range, with speeds fast enough for conventional range-finding applications or for remote actuators. Fully-guaranteed radio coverage can only be achieved if the user deploys their own antenna network in order to fill in any potential blank zones. In such cases, only LoRa enables independence by operating its own infrastructure itself.

Let's take a closer look at some of these technologies.

4G / NB IoT

Cellular connections enable radio communication over distances of several kilometres. Operators provide radio coverage almost everywhere in the country. User terminals are hooked up to a cell which shares the bandwidth between two active terminals, e.g. a maximum of 600 Mbits/s is shared for the LTE-A Cat 12 4G standard. The main disadvantage of this type of radio technology is its high levels of energy consumption. In order to compensate for this, Narrowband Internet of Things (NB IoT) provides a way of reducing this energy consumption, to the detriment of the bandwidth. The main disadvantage of cellular connections is how dependent they are on operators, who have operating licenses relating to the frequency bands used. For blank zones, it is not possible to extend coverage using your own means. Lastly, subscription-based economic models can prove very costly in relation to the number of connected objects deployed.

SigFox

Sigfox is an operator which developed and deployed its own type of radio technology using the 868 MHz ISM band. There are usage restrictions on this band in terms of power and time spent using the physical medium: monopolising more than 1% of the radio channel is prohibited. While managing to respect this constraint, the radio modulation used by Sigfox has been able to overcome a whole host of obstacles. However, the size of packets is limited to 12 bytes, capping the number of messages per day at 140. The main advantage is its extremely low energy consumption, but the disadvantages are the same as for cellular technology (subscriptions, blank zones, etc.).

LoRa

Like Sigfox, LoRa (Long-Range) is a type of radio technology which uses the 868 MHz ISM band in Europe. Initially developed by Cycléo in 2009, before being acquired by Semtech in 2012, LoRa operates on an owner radio modulation, enabling it to achieve radio ranges of up to 5 km in dense urban areas and 15 km in open areas. Owner radio chips are sold at a low cost in the hope of attracting large-scale deployment projects with several thousand objects. LoRa modulation offers different emitting powers known as "spreading factors" (SF), ranging from 7 to 12. The stronger the SF, the lower the bandwidth will be. The SF limits the maximum size of packets, which ranges from 51 to 222 bytes. In order to enhance its ability to withstand interference and to protect the integrity of packets, a CRC and a corrector code (Coding Rate) are included in the LoRa frame.

1.2.4 Overview and comparative assessment

To summarise, no technology is intrinsically better than any other. Different criteria have to be taken into consideration depending on application needs, chief among these being speed, range, power consumption and the type of infrastructure.

The table below provides a comparative assessment based on these criteria.

Technology	energy consumption*	speed	range	infrastructure	operator/autonomous
RFID/NFC	****	NFC: 106 kbits/s to 424 kbits/s	between 10 cm and 100m	point-to-point	autonomous
Wi-Fi	*	1.3 Gbits/s	> 100m	star	autonomous
Bluetooth	***	1 Mbit/s	> 100m	point-to-point, mesh	autonomous
Zigbee	****	256 kbit/s	> 100m	point-to-point, star, mesh	autonomous
4G	*	600 Mbits/s	> 30km	cellular	operator
NB IoT	***	250 kbit/s	> 30km	cellular	operator
Sigfox	****	roughly 800 bits/s	> 10km	cellular	operator
LoRa	****	between 250 and 5470 bit/s	> 10km	point-to-point, star	both

*: ***** means the lowest power consumption and * means the highest power consumption*

The following diagram will help you to choose the most appropriate technology for your application. In many cases, more than one solution is possible, even if you don't necessarily select the ideal solution, provided the chosen technology delivers the desired service.

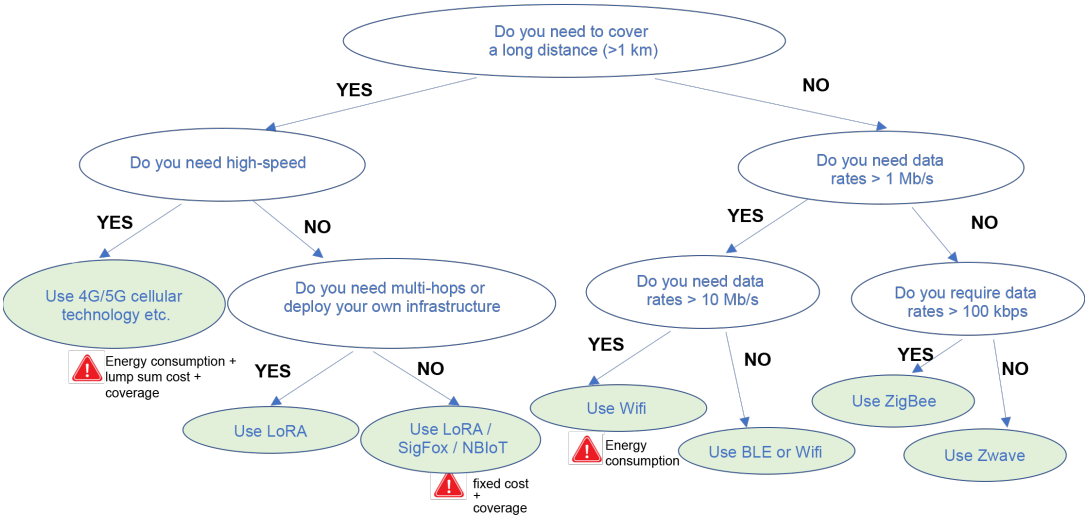


Fig. 1: Technology selection aid

1.3. Examples of IoT Applications

- 1.3.0. Introduction
- 1.3.1. Smart watches
- 1.3.2. Sensors for connected agriculture
- 1.3.3. Sensor networks in smart buildings

1.3.0. Introduction

Video presentation of examples of IoT applications: smart watches, monitoring plant water requirements and managing air quality in buildings.

In this video, we will take a detailed look at 3 concrete examples of IoT applications using different types of radio technology and 3 different ways of communicating with the internet.

1. **Smart watches**, which have become commonplace, are a classic example of **objects communicating with smartphones or computers**, serving as a gateway to the internet.
2. **Temperature and soil moisture sensors** deployed in farming in order to monitor plant water requirements: this presents an example of the use of **low-energy and long-distance cellular networks** for sending information, particularly **LoRaWAN** networks.
3. **Sensors measuring air quality in buildings** and communicating with the building's technical management system in order to control the ventilation system: this example shows how **short-range wireless sensor networks** operate.

1.3.1. Smart watches

Content

1.3.1a Introduction

Smart watches are now widely used by athletes in order to track heart data (heart rate, blood pressure, etc.) during exercise or to count the number of steps taken over the course of a day.

How is this data exchanged between these watches and the internet? Which protocols are used? What constraints are there?

1.3.1b How it works

For this type of Internet of Things application, information acquired by sensors on watches is sent directly via radio, using a *Bluetooth Low Energy* connection (also known as *Bluetooth LE* or *BLE*), to smartphones, users' mobile terminals. This information is then available to consult directly on the mobile terminal via a dedicated application and/or on a website using a computer. In this second example, data is sent by the terminal to an IoT platform through its internet connection: the terminal is connected to the internet via either an operator network (3G, 4G and soon 5G) or a local Wi-Fi network. The protocols used in order to send this data to IoT platforms generally depend on applications: there is no real standard for this part of the chain.

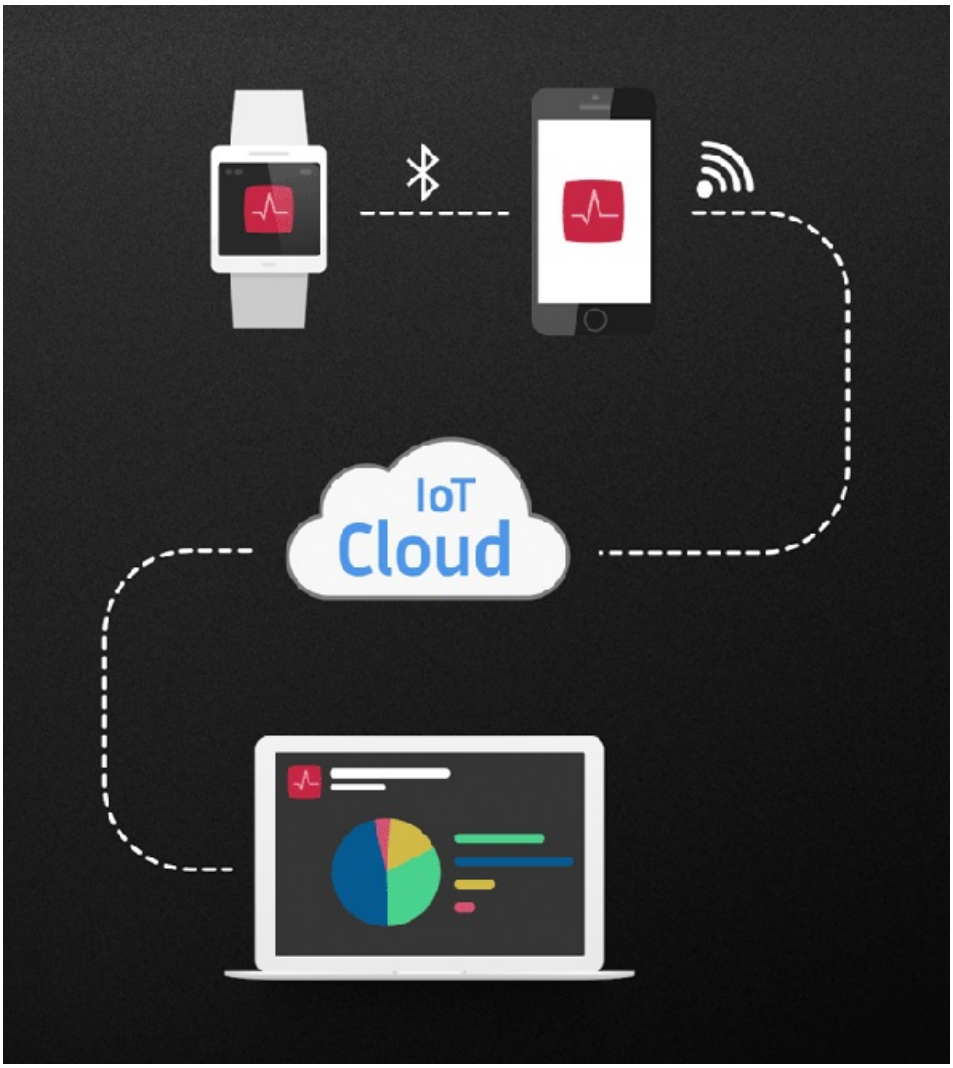


Fig. 1: General overview

1.3.1c BLE protocol

Given that smart watches are targeted at the wider public, they must be easy to use with the terminals already used by potential users, using a radio protocol and applications which are compatible with both the watch and the mobile terminal. For a number of years now, all smartphones have had BLE technology built-in, enabling them to communicate with other terminals or objects. As such, it is this method of communication which is used. For more information on the specifications of the BLE protocol, see https://www.bluetooth.com/

All you really need to know is that Bluetooth LE operates in the 2.4 GHz ISM band. Aside from this physical communication medium, the protocol specifies access profiles in order to ensure interoperability between different classes of objects. A full list of these profiles can be found in the *Generic Access Profile*, from the physical layer up to the*Generic Profile Attribute* (or GATT).

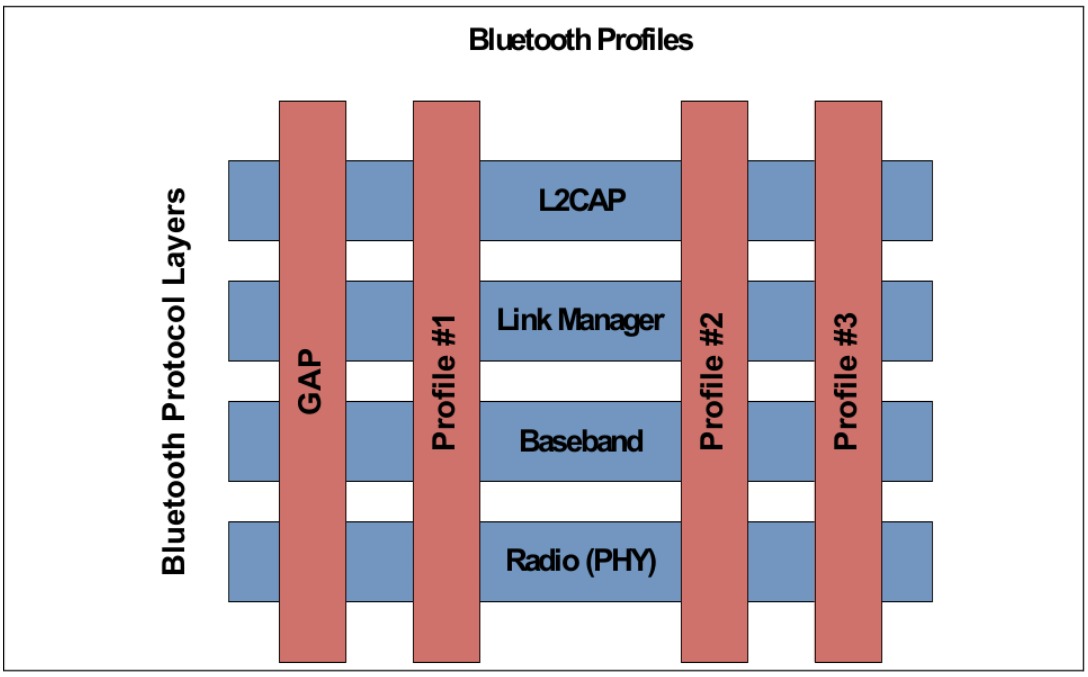


Fig. 2: Bluetooth profiles

Source: https://www.bluetooth.com/specifications/bluetooth-core-specification, Vol 0, Section 6

More specifically, the BLE specification outlines a set of GATT profiles/services designed to ensure interoperability between objects (in this case, between smart watches and smartphones). A GATT profile introduces the concept of clients and servers and what actions are possible between clients and servers: read-only, modification, notification. In order to receive/access data, a client (a smartphone) must establish a connection with the server (the smart watch).

A list of all the GATT profiles and services outlined within the BLE specification can be found on this page: https://www.bluetooth.com/specifications/gatt/.

In order to exchange data on heart rate, for example, the corresponding GATT profile is *HRP* (Heart Rate Profile).

Lastly, data exposed by the server are known as *attributes*. An *attribute* is either a service (listed in the same location as GATT profiles) or a characteristic. Each characteristic contains a description, a unique value and descriptors used to qualify this value.

In our example, heart rate is exposed with the characteristic *Heart Rate Measurement* (code *0x2A37*).

A full list of characteristics can be found on this page:
https://www.bluetooth.com/specifications/gatt/characteristics/

This feature enables any application operating on the smartphone and designed to display the heart rate supplied by a BLE sensor to process the data obtained from the smart watch via BLE by implementing the compatible profile GATT HRP, which will contain a characteristic such as *Heart Rate Measurement*.

Other documentation resources for BLE:https://blog.groupe-sii.com/le-ble-bluetooth-low-energy/

1.3.1d Watch battery life

Given that watches normally have an ergonomic interface and a screen for their primary function (which is to tell the time), it is not really possible to obtain consumption as low as it is for other classes of objects with better self-sufficiency, but it is reasonable to expect self-sufficiency of at least 24 hours, or even several days. In all cases, users of these watches will employ them on a sufficiently regular basis in order to be able to recharge them, as needed.

1.3.2. Sensors for connected agriculture

Content

1.3.2a Introduction

One of the most common uses for connected objects is for controlling the level of soil moisture in gardens or, on a larger scale, on farms. When used in this way, the data fed back by the sensors is then used to trigger an alert in the event of the soil becoming too dry and/or to activate automatic sprinkler systems. Because the data is available in real-time, this type of system can be used to optimise the way in which certain crops are watered (drip irrigation) and to make savings not only financially, but also in terms of the amount of water used, which will be of particular benefit in regions where the resource is limited (e.g. drought in the central valleys of California).

There are a number of questions when it comes to deploying this type of system: what can be done to source data from remote locations, i.e. over large distances? What protocols are used? What constraints must objects respect for this type of system?

1.3.2b How it works

For this type of application, the current trend is to use cellular technology, whereby objects communicate with one or more gateways, which are themselves connected to the internet. Objects and links communicate with each other over frequency bands enabling data to be sent over distances of several kilometres. There are two different scenarios for cellular networks:

- **When the frequency band belongs to an operator:** in such cases, signals can only be emitted in a way that respects the communication protocol the band was set up for, and there must be a commercial agreement in place with the operator. This is the same as for mobile phones. The 900 MHz frequency band, initially allocated for GSM (2G) and now used less and less for mobile communication, has found a new use in the Internet of Things. The NB-IoT (Narrow Band IoT) protocol uses this band, which is shared between a number of operators, to send IoT data by means of a subscription.

- **When the frequency band is open:** this is the case for the ISM band, which can be used by anyone, provided they comply with certain restrictions outlined by a public regulatory body. In Europe, this body is called the *ETSI* (European Telecommunications Standards Institute). For Europe, the ISM bands generally used in connected agriculture are the 433 MHz and 868 MHz bands. In order to be able to emit over these frequency bands, it is necessary to respect 2 constraints outlined by the ETSI, which vary depending on the bands (and sub-bands) used:

  > the emitting power must not exceed a certain threshold
  > the hourly emission length, also called the*duty cycle*, must remain below a certain threshold. This limit is expressed as a % of one hour.

  Two main types of cellular technology use these ISM bands to transmit IoT data**SigFox** and **LoRaWAN**.

In all cases, objects communicate over radio frequencies with base stations (or gateways), via a given frequency band and using a communication protocol linked to the technology being used. The base station then relays this data to a server of the operator it belongs to. Base stations and servers communicate with each other via the internet's standard communication protocols (IP networks). Lastly, these servers are responsible for sending data to their recipients via a pre-determined application protocol. These protocols generally ensure a level of confidentiality for the data, in addition to ensuring that the data is easy to access (like for the internet via HTTP, for example, but there are many other protocols which we won't go into here, such as MQTT).

At an application level, for this type of network, there are no particular specifications for data formatting, meaning 2 different applications will not be compatible unless the format of the data generated by the object is known in advance.

1.3.2c LoRaWAN networks

LoRaWAN networks are cellular networks which operate on the 433 MHz or 868 MHz ISM bands in Europe. In Europe, when using these bands, the *duty cycle* tends to be 1%, but it can be higher or lower on certain sub-bands, while the emitting power may not exceed 14 dBm for 868 MHz or 20 dBm for 433 MHz. The radio technology (determining the modulation and the coding of information) used to enable objects/sensors and base stations to communicate with each other is called LoRa (for *Long Range*) and is capable of sending small data at low data speeds ranging from 292 bits per second to 50 kilobits per second. The LoRaWAN protocol then determines the specifications above LoRa technology in order to connect objects to cellular networks and to enable them to send and receive data.

Figure 1 shows the architecture of a LoRaWAN network:

- The objects which communicate via LoRa with base stations or gateways.
- The gateways which are connected via the standard internet network to a network server. These servers will be located *in the cloud* and will be managed by the operator of the LoRaWAN network.
- The applications are connected to the network server, enabling users to access data, either via the web, or via other application protocols, such as MQTT.

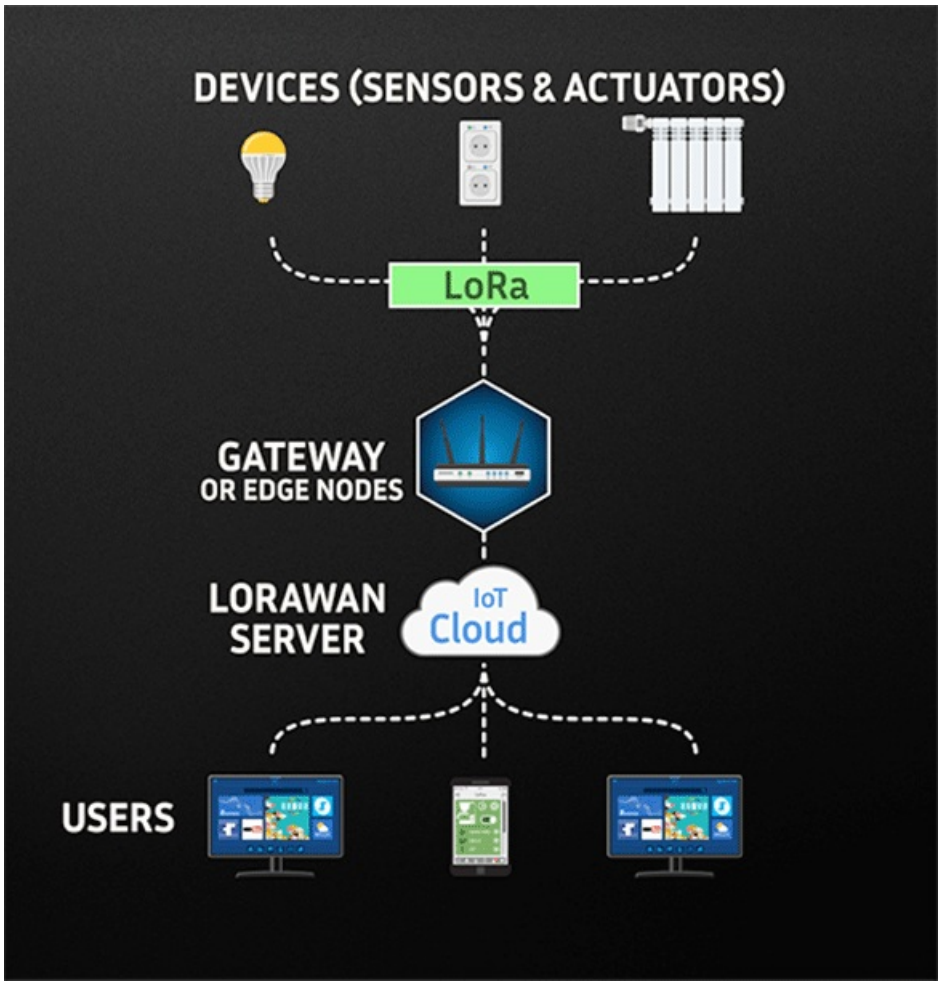


Fig. 1 : Overview of a LoRaWAN network

In LoRaWAN networks, lists of usernames and keys are programmed on objects, enabling them to authenticate themselves with the network server. This aspect is important in enabling the network server to determine if an object does indeed belong to an application managed by the operator of the network. Any piece of data received by the server which does not belong to an application by the operator will simply be rejected.

Let's take a brief look at network operators: there are many, and so we won't list them all here. In France, for example, certain operators such as Orange or Objenious (Bouygues Télécom) are subsidiaries of major mobile operators. They offer pretty extensive coverage at a national level, with billing based either on the number of objects, or on the number of messages. When starting out, it can sometimes be simpler to use operators with open network access, such as Loriot or The Things Network. For these types of operator, volunteers use their own gateways to ensure coverage, meaning that the service is not always continuous. That said, access to the network is free, and far less restrictive in terms of the number of messages or objects.

1.3.2d Sensor autonomy

There are 3 different classes of objects in LoRaWAN networks:

- **class A objects** are the objects with the lowest energy consumption. These objects spend the majority of their time in sleep mode, consuming practically no energy, and only waking up

periodically to send data. After the data is sent (known as an *uplink*), they open 2 listening windows in order to receive data from the network server (this is what's known as a *downlink*). For this class of objects, latency is very high, owing to the fact that they can only receive data after sending. The advantage is their level of consumption and how suitable they are for use in control systems deployed in vast, remote areas.

- **class B objects** open their listening windows periodically, meaning they can be reached more often by the network server. However, the fact that they wake up more often means they consume more energy.
- **class C objects** are awake all the time, and even when they are not emitting data, they continue to listen out for any messages sent by the network server. The advantage is that they offer very low latency: messages sent by the server will be received straight away by the object. However, they consume vast amounts of energy, meaning they can't be used for applications involving battery-operated objects.

As you will no doubt have understood, for use in connected agriculture as shown here, only class A objects may be used, given the importance of energy self-sufficiency.

Some modern microcontrollers consume roughly 1 microampere in minimum operating mode. Supplied with small 2000mAh batteries, this ensures they will remain self-sufficient for decades.

1.3.3. Sensor networks in smart buildings

Content

1.3.3a Introduction

Connected objects can also be used to improve the way in which houses or buildings are controlled and monitored. The objective may be to optimise their energy consumption by precisely measuring the temperature at various points in the building in order to better control heating (or air conditioning); to anticipate wear and maintenance on certain items of equipment (what is known as predictive maintenance in the context of industry 4.0); or even to improve air quality by activating the ventilation system, should this prove necessary. And this is far from an exhaustive list!

In existing buildings, however, it is too costly to have to draw new cables in order to install new sensors. It is for this reason that preference is given to communication technology using short-range radio frequencies, with battery-operated sensors. These types of sensors can be deployed anywhere in buildings: in corridors, in offices, in machine rooms, etc.

Sensor networks feed measurements from different sensors back to buildings' technical management systems or an application in the cloud. These are used to control the functional aspects of the building: heating, ventilation, lighting, etc. An alert system can also be triggered to indicate when maintenance is required.

How do sensor networks work? How are they connected to buildings' technical management systems?

1.3.3b How it works

Sensor networks are designed to compensate :

- a lack of infrastructure (no wired network),
- short radio ranges by using each sensor in the network as a relay (also known as *nodes* or *routers*).

In this way, it is possible to transmit information over longer distances than the radio range of one single sensor through what are known as *hops*. This type of network is also known as a *mesh* network.

All that then has to be done is to position a gateway (also called a *well node*) sufficiently close to the network in order to route data from the network towards the building's internal network, or even directly to the internet.

Once on the building's internal network/the internet, this data can then be sent to application servers, users or buildings' technical management systems, and can thus be used to take decisions: e.g. to start up ventilation systems in the event of air quality exceeding a certain

threshold.

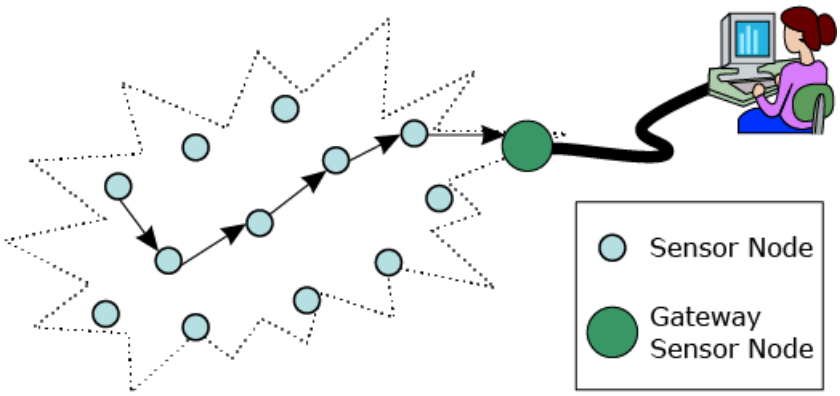


Fig. 1 : Wireless sensor network overview

Source: https://en.wikipedia.org/wiki/Wireless_sensor_network

1.3.3c Existing protocols

For this type of installation, use of the 2.4 GHz ISM band offers the best speeds and a maximum range of roughly 100 metres.

There are a number of mesh type sensor network solutions operating within this band, such as BLE mesh, Wi-Fi or 802.15.4

802.15.4. is a standard that was developed specifically for this use case, having been standardised by the IETF (*Internet Engineering Task Force*) for low-energy consumption sensor networks.

This protocol is perfectly suited to use in smart buildings, which is why we will be focusing on it here.

For more details on 802.15.4., please go tohttps://en.wikipedia.org/wiki/IEEE_802.15.4.

As we will see in greater detail in module 4 of this MOOC, 802.15.4. sensors operate on the same principle as IP networks, with a multi-layer model. Some of these layers had to be adapted in order to make them compatible with highly constrained sensors (very low memory, only a few dozen kilobytes).

Above the 802.15.4. radio layer, there is:

- an address layer for allocating IP addresses (more specifically IPv6) to sensors
- a routing layer for multi-hops
- a transport layer
- an application layer

All these layers rely on protocols designed for sensor networks using highly constrained objects, i.e. objects with extremely limited processing capacities (a few dozen MHz), very low memory (a few kilobytes) and low energy consumption. The advantage of the multi-layer model is that it makes sensor networks interoperable with conventional IT networks, enabling sensors to communicate directly with servers and vice-versa. By using the same application protocol, it is possible to relay data from sensors to the equipment used to manage the building.

1.3.3d Sensor autonomy

In 802.15.4. networks, sensors remain in standby mode for the majority of the time, meaning that latency is high for this type of network. Even if this leads to significant routing constraints - the topology of the network has to readjust depending on whether or not the sensors are awake - generally speaking, the sensors can remain self-sufficient for up to several years.

1.4. FIT IoT-Lab Usage

- 1.4.0 Introduction
- 1.4.1 Why have such a platform?
- 1.4.2 What does FIT IoT-LAB offer?
- 1.4.3 How to use FIT IoT-LAB?
- 1.4.4 The use of FIT IoT-LAB in this MOOC
- TP1. Welcome to the IoT-LAB testbed training

1.4.0. Introduction

This MOOC offers activities linked to experimenting with and programming connected objects. Practical application will involve real hardware, but no purchases will be necessary - instead, you will use hardware made available by an experimentation platform for the IoT: FIT IoT-LAB.

The purpose of this sequence is to give you a brief overview of the FIT IoT-LAB platform, explaining its origin and what it was designed for, its features, how to use it and, more specifically, how you will be using it in the context of this MOOC.

Within reach of sensors
Wireless sensor networks are becoming increasingly widespread in the Internet of Things, particularly in smart cities. This video, in French only, lets you discover the research taking place in this field and the resulting need for experimentation. We present here the FIT IoT-LAB experimentation platform, resulting from an ANR project started in 2008. It is currently deployed on six sites in France, namely Strasbourg, Lille, Saclay, Paris, Lyon and Grenoble. There are now more than 2,300 sensor nodes available to users for experiments in embedded wireless communications.

1.4.1. Why have such a platform?

The FIT IoT-LAB platform comes from the academic world. In the early 2000s, researchers working in the field of sensor networks had to validate their research through the use of network simulators. However, problems linked to wave propagation and interference also required validation through experimentation. They then had to purchase, and sometimes even design, electronic cards incorporating a microcontroller, sensors and a radio chip. Today, this might seem relatively straightforward, but 15 years ago, the makers movement and Arduino or Raspberry Pi circuit boards had yet to democratise programmable electronics for the wider public as they have since.

On top of the financial cost of the equipment, there was also the time taken up by experiments. And once they had designed the program to run on the circuit board, they then needed to deploy the experiment. This involved:

1. programming the circuit boards one by one, connecting each in turn to a computer
2. recharging the batteries used to power the circuit boards
3. deploying the different circuit boards, following a physical topology enabling radio communication
4. implementing a solution for retrieving the experiment logs (e.g. storing the output from the main network node's serial link in a text file).

Any bugs or modifications in the application to be tested would mean having to start the whole process over again. Throw in the upscaling test, featuring a sizeable number of nodes in the network, and you get an idea of just how complicated an experimental validation was.

In this context, in 2008, researchers began work on an innovative research project aimed at creating an open, large-scale experimentation platform for wireless sensor networks. The goal was to save time in the experimental phases, to test easily on a large-scale and to be able to replay experiments (e.g. for reproducible scientific purposes).

This platform is now part of theFIT infrastructure (Future Internet), a large-scale experimentation environment thanks to the federation of the following platforms:

- FIT Wireless for experimenting on a wide range of wireless network problems (Wi-Fi, 5G and cognitive radio)
- FIT Cloud, linked to cloud design, the development of OpenStack components and services
- FIT IoT-LAB, dedicated to experiments for the Internet of Things, particularly embedded technology.

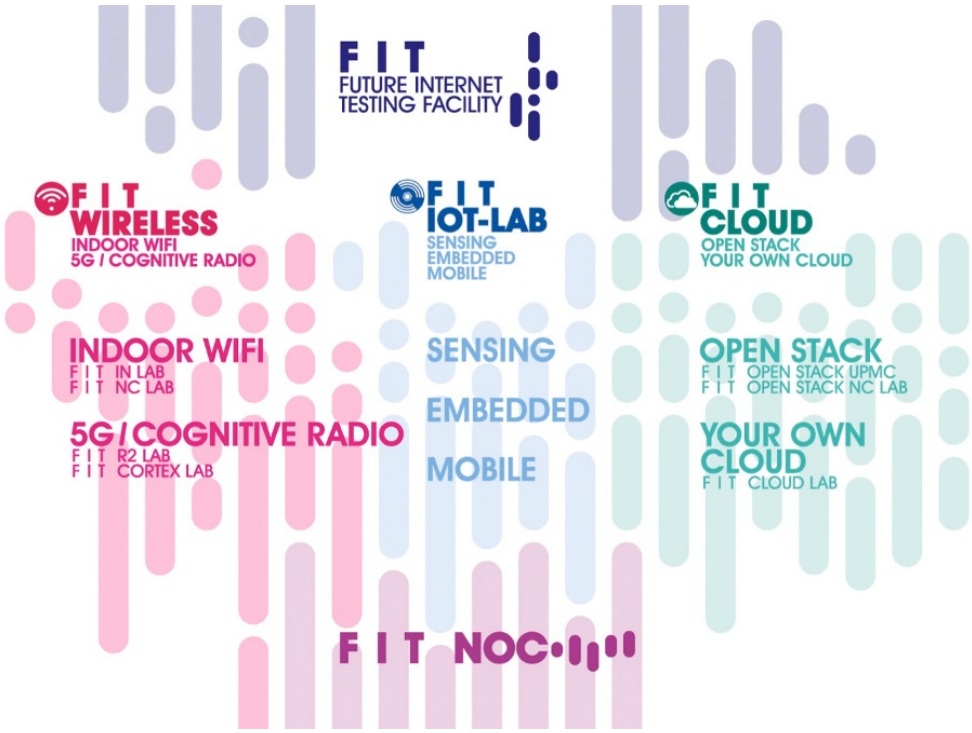


Fig. 1: The FIT federation © FIT Equipex

1.4.2. What does FIT IoT-LAB offer?

IoT-LAB now enables access to 1,500 nodes for experimenting, deployed across various different sites, the majority of which are in France. This deployment makes IoT-LAB:

- multi-platform, in that it offers various different experimentation boards
- multi-radio, in that boards don't all have the same radio chips, and some have more than one
- multi-topology, in that the physical deployments are all different (in grids, in several corridors, dense, scattered, over several levels, etc.)
- multi-OS, in that the different boards have one or more embedded OS support.

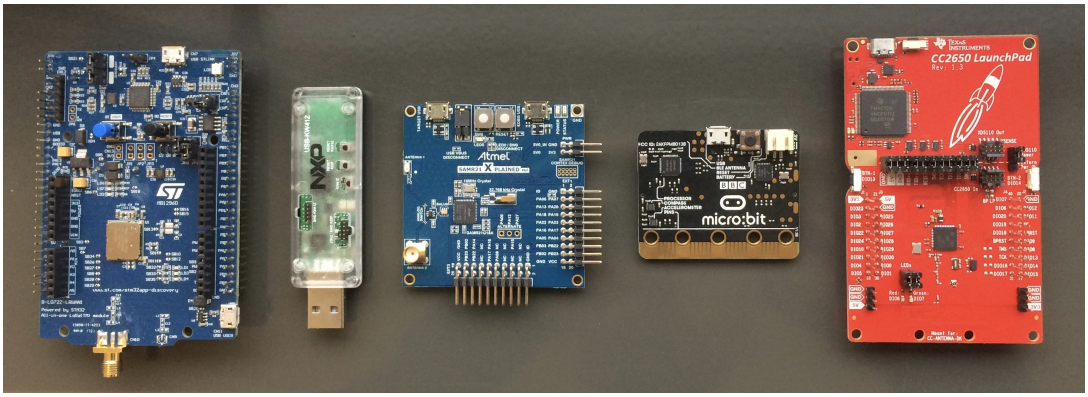


Fig. 1: Different experiment boards from retail © J. Vandaele

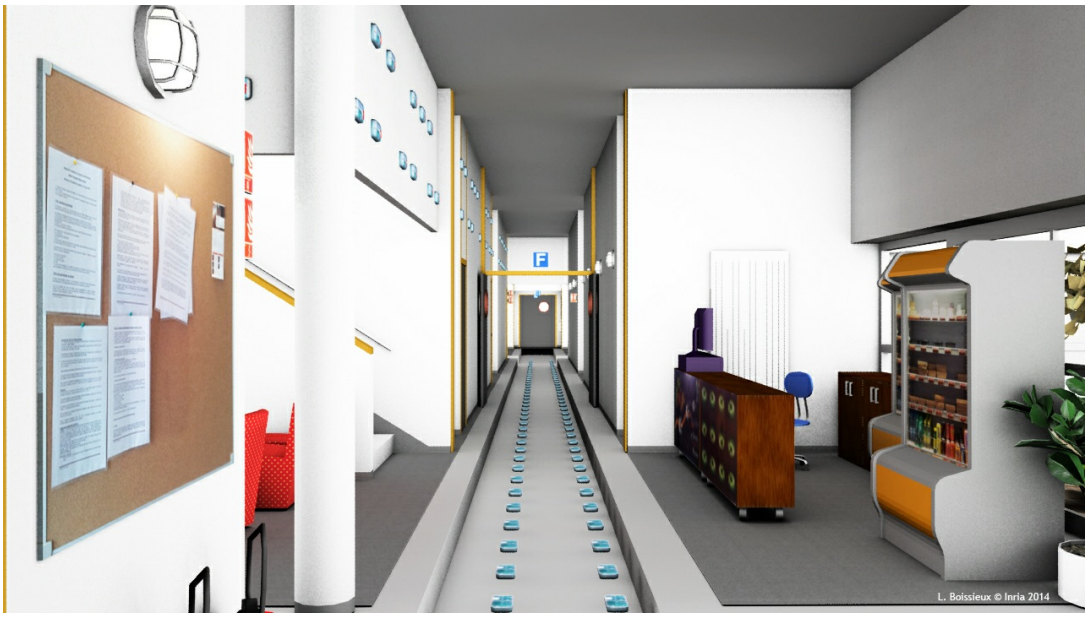


Fig. 2: Deployment of the platform in Grenoble © Inria / Photo L. Boissieux

Fig. 3: Deployment of the platform in Lille © Inria / Photo C. Morel

Fig. 4: Deployment of the platform in Saclay © Inria / Photo C. Morel

Fig. 5: Deployment of the platform in Strasbourg © ICube / Photo G. Schreiner

More than anything else, IoT-LAB is a platform that is open to all users: students, teachers, makers, researchers, manufacturers, etc. It is free to access - all that's needed is for users to create a user account and to respect the usage rules.

1.4.3. How to use FIT IoT-LAB

For a simple use of the platform, there is a web portal letting you create an experiment on several hundred boards in just a few steps. On each site, a server accessible via SSH (known as SSH frontend) is used to access a development environment, as well as to access collected data or boards serial links. The platform also has a number of command line interface tools for advanced uses or for task automation. The platform also features an open API for developers.

Around forty or so tutorials are available on the platform website, introducing all these features: https://www.iot-lab.info/tutorials/

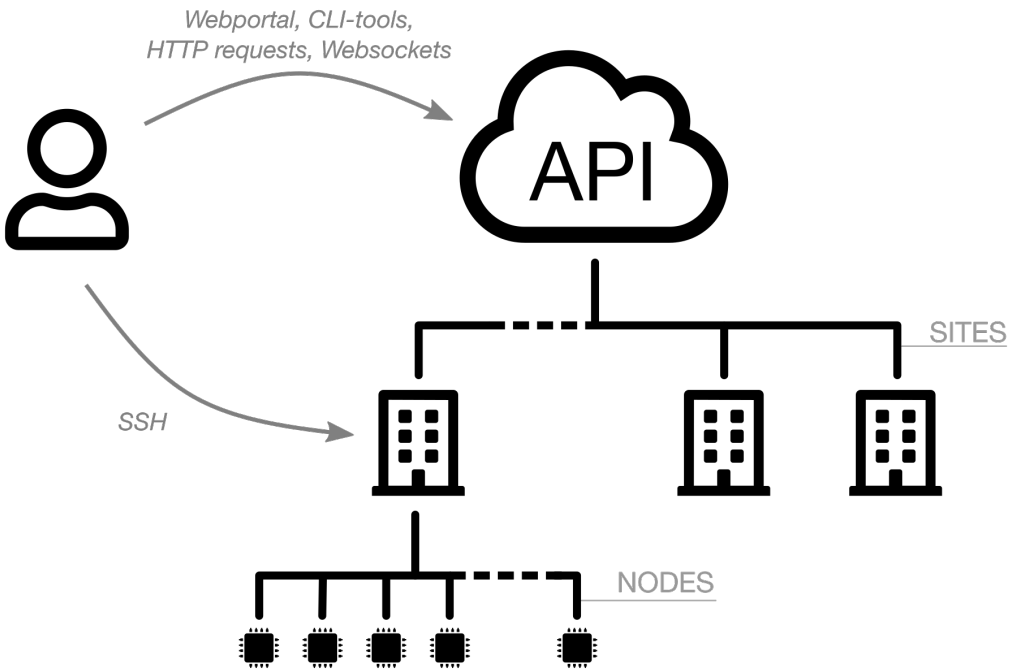


Fig. 1: Overview of interactions with the platform

1.4.4. The use of FIT IoT-LAB in this MOOC

Practical activities will be available almost exclusively through the JupyterLab environment, one single interface that will enable you to follow the instructions in the notebooks, to edit the code for applications and to interact with the platform and with experiment nodes from terminals. This environment has been customized by integrating all of the necessary IoT-LAB tools, meaning you won't have to install anything and can do everything from your web browser.

A dedicated MOOC account will be created for each participant. The authentication parameters (access to the API and SSH keys) will be installed automatically for you in your JupyterLab environment. Once again, you will not have to configure anything yourself.

The engineers from the IoT-LAB's development team (Alexandre Abadie, Frédéric Saint-Marcel, Guillaume Schreiner and Julien Vandaële) are part of the educational team for this MOOC. They are experts on this platform, and will be happy to answer any questions you may have on the forum.

For a concrete example, discover in this video how to deploy and interact with an experience from the IoT-LAB web portal and how to do the same from the MOOC environment.

Module 2. Focus on Hardware Aspects

Objective: At the end of this module you will be able to explain the hardware architecture of a connected device with the energy constraints associated. You will also be able to classify IoT devices according to their role or application.

Hands-on activities (TP) : TP2 : First IoT-Lab experiment

Contents of Module 2

2.1. IoT Device Hardware Architecture

- 2.1.0. The hardware architecture of connected objects
- 2.1.1. Embedded systems
- 2.1.2. Microcontrollers
- 2.1.3. Sensors and actuators
- 2.1.4. Power supply

2.2. Data Buses: UART, I2C, SPI

- 2.2.0. Introduction
- 2.2.1. UART data buses
- 2.2.2. The I2C data bus
- 2.2.3. The SPI data bus
- 2.2.4. Overview and elements of choice

2.3. Connected objects: IoT-LAB M3

- 2.3.0. Introduction
- 2.3.1. The connected object IoT-LAB M3
- 2.3.2. The IoT-LAB gateway
- TP2. First IoT-Lab experiment

2.1. IoT Device Hardware Architecture

- 2.1.0. The hardware architecture of connected objects
- 2.1.1. Embedded systems
- 2.1.2. Microcontrollers
- 2.1.3. Sensors and actuators
- 2.1.4. Power supply

2.1.0. The hardware architecture of connected objects

This sequence gives you the opportunity to discover the different components of a connected object: its embedded system, its means of interacting with its environment; and its power supply and energy consumption.

References

- https://en.wikipedia.org/wiki/Microcontroller
- https://en.wikipedia.org/wiki/Central_processing_unit

Video presentation introducing the hardware architecture of connected objects.

2.1.1. Embedded systems

Content

2.1.1a Connected objects: a certain type of embedded systems

Connected objects are simply mass market applications of communicating embedded systems, that have been around since the late 1960s in a range of industrial applications. The emergence of embedded systems was made possible thanks to the miniaturisation of transistors, which enabled the creation of new components, such as integrated circuits.

An integrated circuit or electronic chip is a component that reproduces multiple electronic functions of varying degrees of complexity. These can be seen as ready-to-use building blocks, capable of being integrated into applications.

Embedded systems are both electronic and computer systems, which are designed for a specific task and which operate autonomously and, where necessary, in real-time. Embedded systems are optimised for precise applications and must meet the following constraints:

- components manufacturing cost
- processing power, memory
- energy consumption
- form factor

Specifically with regard to connected objects, we can also add the following communication constraints to embedded systems:

- bandwidth for sending messages
- the radio range or wired technology

2.1.1b The architecture of embedded systems

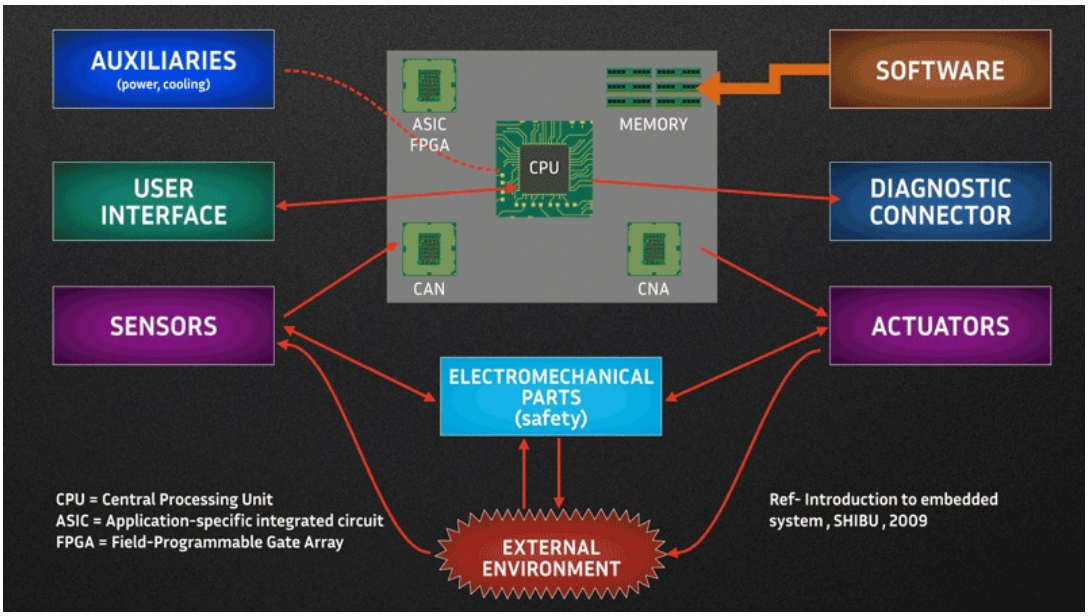


Fig. 1: Overview of an embedded system

Embedded systems can be broken down into individual hardware and software components (see figure above):

- The microcontroller (**MCU**), which coordinates inputs/outputs for the different components
- **ROM** and **RAM**, which store **software** (firmware) on a permanent basis and variables on a volatile basis during execution
- **Sensors** and **actuators** specific to the application
- The **power supply**, which supplies current and controls voltage levels for the MCU and various other components
  - The battery
  - Current input
  - Charger module
- The various different local**communication buses** used to interface the MCU with the sensors, actuators, radios, screens, automatic control units, etc.
  - ADC: analogue and digital inputs/outputs
  - I2C
  - SPI
  - UART
  - MODBUS, CAN (particularly in industrial environment, the automobile industry, etc.)
- Communicating chips for external communications
  - **Radio** chips
  - **Wired** chips
- **JTAG/JLINK**: chips used to reprogram objects' firmware. Often found on development boards, objects produced in large volumes can do without this type of component in order to save money. In such cases, an external programmer must be employed for the embedded system, or the firmware must be updated via the radio chip using OTA (Over The Air).

2.1.2. Microcontrollers

Content

2.1.2a Microprocessor architecture

The **microprocessor** is the most important component of a microcontroller. These microprocessors are comprised of:

- a (CPU) **processor**, which contains:
  - a control unit used to manage instructions in the process of being executed
  - a clock used to set the tempo of the processor
  - registers, which are small, very quick internal memories used to store instructions in the process of being executed
  - specialist processing units (arithmetical and logical or UAL, floating point, etc.)

The CPU interacts with:

- **memories**:
  - read-only memory (ROM) for program storage (firmware)
  - random-access memory (RAM) for storing the program variables in the process of being executed
- **inputs/outputs** (I/O) connected to devices

CPU, memories and I/O are linked by three buses:

- the **address bus**, which is unidirectional and which is controlled by the CPU. This is used to select the memory box in order to read or write either a piece of data or an instruction. Its size determines the maximum amount of memory that can be allocated. For example:
  - 16 bits: 2^16 = 65,536 bytes = 65.5 KB
  - 32 bits: 2^32 = 4,294,967,296 bytes = 4.29 GB
- the bidirectional **data bus**. This is used to exchange data or instructions between different elements. Its size will determine the CPU category (8-bit, 16-bit, 32-bit).
- the **control bus**, which is used to control the type of operation

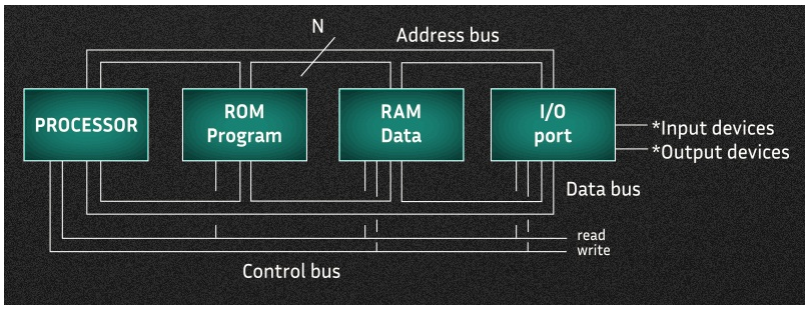


Fig. 1: Architecture of a microprocessor

2.1.2b From microprocessors to microcontrollers

One single processor is not enough to build an embedded system. Additional components, such as RAM or ROM, for example, are essential when it comes to putting together a functional object. In order to simplify the design of embedded systems, manufacturers of components supply complete pieces of hardware equipped with all core components needed to operate at a minimum level. These are known as **microcontrollers** or MCUs. Practically speaking, all components (CPU, RAM, ROM, etc.) are located on the same chip, which is miniaturised by the manufacturer.

When it comes to developing embedded systems, there are a number of advantages to using MCUs instead of multiple separate components:

- saves space on the printed circuit board: no routing between components, which require a minimum space between them for welding purposes; MCUs are single chips with input/output pins to be welded onto the embedded system
- more energy efficient - the components are closer together
- saves time on product certification: MCUs and their internal components have already passed this stage
- saves on manufacturing costs (components welded one by one)

In order to provide designers of embedded systems with as much flexibility as possible, MCU manufacturers offer a wide range of products, catering to all needs (CPU architecture, internal memory size, etc.).

In addition to the CPU and internal memory, MCUs are also generally equipped with a set of peripheral functions for use in embedded programming, such as timers, interrupts, watchdogs, analog-to-digital converters, etc.

2.1.2c Timers

Timers are directly linked to the cycles or**ticks** of the clock of the MCU with which they are synchronized. Depending on the type of MCU, timers are coded either on 8 bits (256 values) or 16 bits (65535 values). A standard MCU will easily exceed 1000000 ticks per second (CPU > 1 MHz). In order to use timers over longer periods, MCUs offer prescalers, which use a fraction of the clock pulse of an MCU.

2.1.2d Interrupts

Interrupts managed by MCUs are linked to both internal and external events. There are many different use cases for interrupts, which are essential when it comes to smoothly programming

embedded systems. Examples include controlling the on/off state of a button, or the arrival of messages on radio chips.

For internal interrupts, internal MCU timers can be programmed to generate interrupts once they have expired.

For external interrupts, external interrupt registers are linked to an external pin on the MCU to which a piece of external hardware has been connected (a sensor, a Real Time Clock (RTC), etc.). This piece of hardware will generate either a high or a low signal, triggering the interrupt, which is then fed back to the MCU.

With regard to the embedded code, interrupts are linked to processing functions to be executed. When the MCU detects an interrupt, the main code in the process of being executed is paused in order to immediately switch to the function linked to the interrupt. Once the interrupt function is complete, the main code will return to its execution where it left off.

2.1.2e Watchdogs

The purpose of **watchdogs** is to ensure embedded systems remain in a consistent state. More specifically, watchdogs prevent systems from becoming stuck in programming loops or stuck waiting for hardware that is slow to respond. In concrete terms, this is a countdown that the MCU must recharge on a regular basis; otherwise, if the watchdog expires, either an interrupt will be triggered on the MCU, or it will reset. The MCU will then start again from the beginning of the program in a consistent state.

2.1.2f Analog-to-digital converters (ADC)

MCUs are also generally equipped with analog-to-digital converters, or **ADCs**. ADCs receive analog signals, from potentiometers or photoresistors, for example, which they then convert into a digital value, which is typically precise to within 8 bits (256) or 12 bits (4096).

2.1.2g Digital inputs/outputs

MCUs are capable of assigning external pins to digital input/output ports. It is then necessary to specify the operating mode for the digital port (input or output), in addition to whether or not an internal *pull-up* resistor will be used to force the pin to its upper state. For example, an LED is connected to a digital port operating as an output in order to switch the component on or off.

2.1.3. Sensors and actuators

Content

Connected objects have to be able to interact with their physical environment, and so are fitted with multiple sensors or actuators. With regard to embedded systems, sensors/actuators are seen as items of hardware which are external to the MCU and which are linked via a data transmission bus: ADC, digital, I2C, SPI, UART, CAN.

2.1.3a Sensors

A sensor is an element capable of detecting a physical phenomenon (displacement, presence of an object, amount of light, humidity, temperature).

- Temperature, humidity
- Air, gas
- Accelerometer, GPS
- Microphone
- Current
- Pressure, buttons, potentiometers, joysticks
- Presence

2.1.3b Actuators

An actuator is an element capable of creating a physical phenomenon (moving an object, creating light, creation of heat, emission of sounds). The actuator will create the physical phenomenon thanks to an energy source.

- LEDs
- Speakers, buzzers
- Electrical relays
- Motors

2.1.4. Power supply

Content

Different groups of connected objects have different levels of energy consumption.

- **Constrained objects**: mass deployed objects, with energy-optimised embedded systems. In active state, consumption is somewhere in the region of **40/50 milliamperes**, and can drop as low as **nanoamperes** in standby mode.
- **Gateways**: objects that must always be switched on in order to relay messages from constrained objects. Gateways are normally coupled to powerful microcontrollers capable of running a classic OS (Linux) authorising processing ahead of the cloud platform. These gateways are also connected to the Internet (Wi-Fi, Ethernet) and have IP addresses. Energy consumption in active state is somewhere in the region of a few **hundred milliamperes** on a permanent basis.

2.1.4a Standby mode

Some connected objects are capable of operating autonomously for several months, or even several years. Connected objects that are capable of energy-autonomy over several months are in standby mode 99% of the time.

In order to extend battery life as much as possible, microcontrollers offer advanced energy management, which involves electrically deactivating multiple sets of components. This is referred to as standby or “sleep” mode. Different manufacturers or types of microcontroller offer different levels of standby, affecting how much energy is saved. Energy consumption ranges from milliamperes in normal mode, to nanoamperes in the most energy-efficient modes. However, with these advanced modes, objects will take longer to wake up. In some cases, RAM data can be lost when the microcontroller is reset while it is waking up (e.g. Espressif ESP32 Deep Sleep mode).

2.1.4b Radio channel access optimization

Excluding connected objects with screens, the components that consume the most energy on connected objects are **radio chips**. Radio transmission (TX) ordinarily consumes significantly more energy than radio reception (RX). The designer of the connected object may limit transmission messages on their application. Alternatively, it is necessary to optimize the radio reception by periodically putting this component to sleep. These optimizations are generally treated at the link layer level of the OSI model (MAC layer).

2.1.4c Towards autonomous systems

In order to attain full energy autonomy, it is necessary to add a power supply which produces more energy than is consumed by the connected object.

The most widely-used solutions involve recharge modules coupled to batteries (LiPo) storing energy and one or more sources of renewable energy producing electricity (e.g. solar panels, wind

turbines). The solar panel must have a surface big enough to attain the requisite level of production for making the system fully autonomous.

2.2. Data Buses: UART, I2C, SPI

2.2.0 UART, I2C and SPI data buses

Current microcontrollers offer multiple types of data bus. The role of these buses is to link the microcontroller's inputs/outputs to the different pieces of hardware of which an embedded system is comprised. Generally speaking, the type of hardware will determine the type of data bus that is required.

This sequence will introduce you to the different types of data bus, providing concrete examples of how they can be used with electronic components. You will learn how to use existing data buses for different components.

2.2.1. UART data buses

Content

2.2.1a How it works

Universal Asynchronous Receiver Transmitters or **UARTS** are point-to-point communication buses used to transfer data between 2 MCUs or 1 MCU and an integrated circuit (see example below).

UARTs use two distinct wires -**TX** and **RX** - for uplink and downlink message flows (**bi-directional**) plus a grounded reference wire **GND**.

Data is transmitted byte-by-byte. The emitter's embedded code copies the byte into the emitting UART, which then transfers it bit-by-bit to the receiver UART. Once the bits have been captured, and a full byte has been formed, this is then fed back to the receiver's embedded code.

The implementation of the protocol uses two levels of TTL tension in order to code the status of a bit:

- the high logic level or "1", which corresponds to to 5 V or 3.3 V (VCC) depending on the voltage used to supply the MCU
- the low logic level or "0", which corresponds to 0 V (GND)

The default logic level is 1 (no traffic). UART frames are broken down into bits as follows:

- **Start**: a bit at 0 used for synchronizing with the receiver
- **Data**: between 5 and 9 bits. Please note, bits are sent in order, beginning with the Least Significant Bit (LSB) and finishing with the Most Significant Bit (MSB). Generally speaking, data bits constitute a symbol. For the value 8, the data corresponds to one byte.
- **Parity** (optional): Even or Odd.
- **End**: number of bits at 1 (1, 1.5 or 2)

The speed of the UART is defined in bps, which is the number of bits transmitted in one second. This speed can be configured to different possible speeds, from 4,800 bps up to 921,600 bps.

Please note, two UARTs in relation with each other must have the same configuration if they are to be able to decode frames.

Below is an example of a frame exchanged between two UARTs

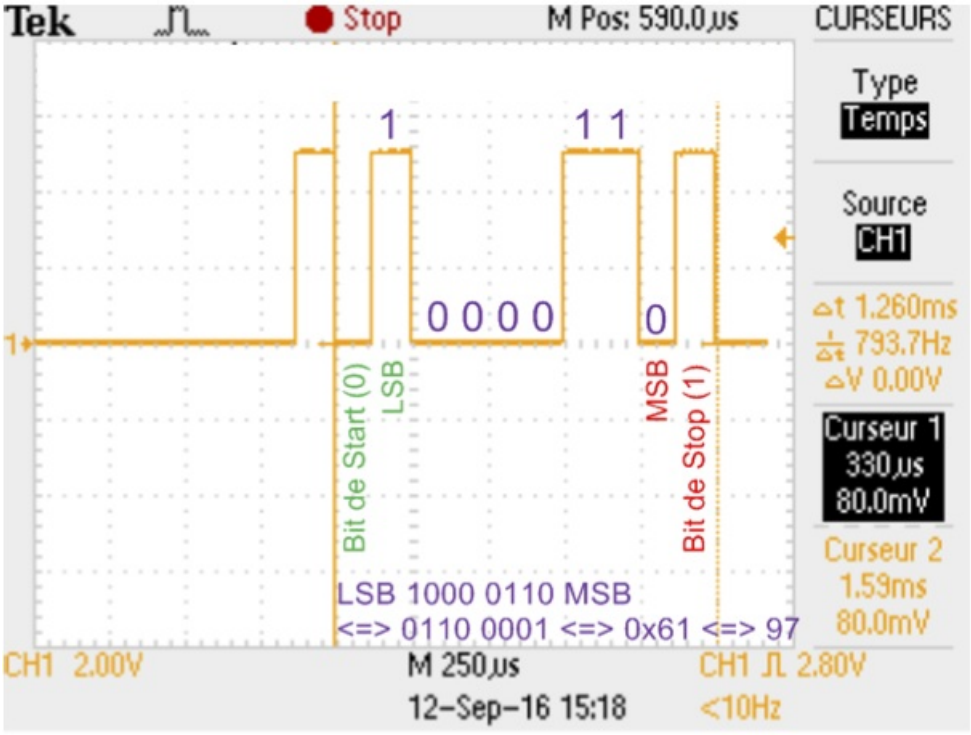


Fig. 1: An example of a frame between two UARTs

By studying the status of the tension levels, the frame can be broken down as follows:

- **0**: the frame start bit
- **10000110**: inverted data (LSB first)
- no parity bit (OFF)
- **1** Stop

Data bits are converted in order to obtain the final symbol:

- **01100001** in bits
- **0x61** in hexadecimals or **97** in decimals
- the character **a** in the ASCII table

2.2.1b Use cases

- **Interaction with the MCU**: MCUs can be connected via UARTs to computers in order to display messages (printf) and to wait for messages to be entered by the user (scanf)

```
MCU UART <-> computer UART <-> terminal
```

- **NMEA frame GPS module**: GPS type components are typically connected behind a UART in order to relay GPS frames to the MCU. These GPS frames are used to geotrack connected objects or to synchronize objects' internal clocks.

```
GPS UART <-> MCU UART
```

2.2.2. The I2C data bus

Content

2.2.2a How it works

The **I2C** (Inter-Integrated Circuit) bus was developed by Philips in the 1980s in order to link an MCU to the different circuits built-in to modern televisions. Historically, it is therefore found in a lot of audio/video circuits.

It's a serial bus that allows electronic components to communicate with each other via three wires or lines. The protocol implementation uses two signal lines plus one reference line connected to ground.

- the **SDA** (Serial Data) line: used to transfer data
- the **SCL** (Serial Clock) line: used to time the sending of messages on the SDA line
- the **GND** line (ground): used as a reference for high and low logical levels on the SDA and SCL lines

Data exchanges are **bi-directional**, but with a Master-Slave model used. Messages are always sent by the Master. The I2C bus can also have more than one master. Each subscriber (Master or Slave) has a 7-bit address allowing for no more than 128 addresses on the same bus. Each byte transferred is acknowledged.

The speeds of I2C buses can reach as high as 100 kbits/s (standard) with the first version of the protocol, and up to 400 kbits/s (fast) and 3.4 Mbits/s (HS) with the latest versions of the protocol.

SDA and SCL lines use two logical levels: low or “0” and high or “1”: The rest state mode of the I2C bus corresponds to the high level state for the SDA and SCL lines.

SDA and SCL synchronization

Signals between SDA and SCL are **synchronized**. A piece of data from the SDA line is considered as being valid if the SCL line is high. The SDA line will maintain the data (0 or 1) for as long as the SCL is high. Once the SCL has gone back to low, SDA can then change value for the next bit of the byte to be transferred.

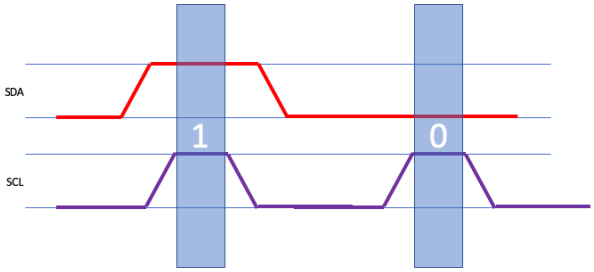


Fig. 1: Transmissions of bits 1 and then 0 on the I2C bus

The I2C bus communication process

1. The Master sends a message on the bus
2. Start condition by the Master: switches the SDA line from 1 to 0 while keeping the SCL line at 1. The rate of the SCL line is controlled by the Master.
3. The Master sends the address of the recipient (7 bits) + read/write access mode (1 bit)
4. Acknowledgement by the Slave

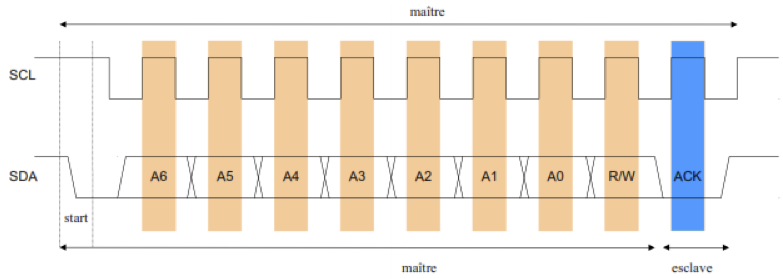


Fig. 2: I2C addressing phase

5a. Data transferred, read by the Slave, then acknowledged by the Master

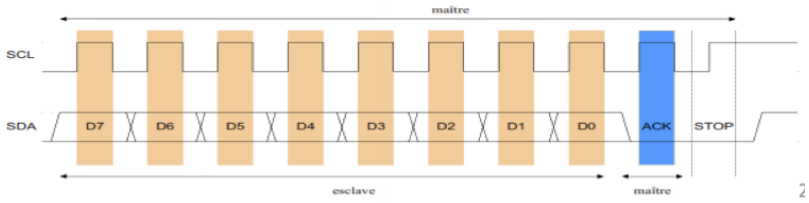


Fig. 3: I2C reading of data sent by the Slave to the Master

5b. Data transferred, written by the Master, then acknowledged by the Slave

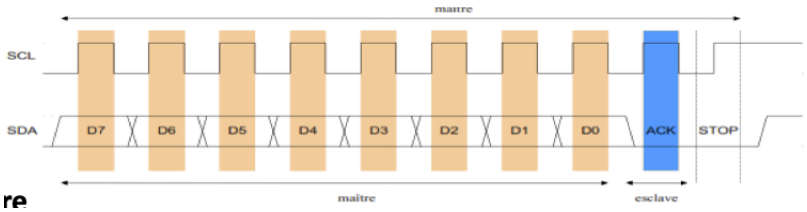


Fig. 4: I2C writing of data sent by the Master to the Slave

6. Stop condition by the Master: switches the SDA line from 0 to 1 while keeping the SCL line at 1.

2.2.2b Use cases

- Enabling an MCU to interrogate an I2C sensor
- Creating a wired link between two MCUs in order to exchange data. Please note, only the Master may initiate communication. For example:
  - > Two identical MCUs: Arduino UNO <-> Arduino UNO
  - > Two different MCUs: Arduino UNO <-> Raspberry Pi
- Reading and writing data in an EEPROM flash memory, linked to the MCU using an I2C bus

2.2.3. The SPI data bus

Content

2.2.3a How it works

The *Serial Peripheral Interface* or **SPI** bus was designed in the 1980s by Motorola. The SPI link is a serial bus used for synchronous data transmission between a master and one or more slaves (multipoint). The transmission takes place in full duplex.

The SPI bus consists of two data lines and two signal lines, all unidirectional:

- **MOSI** (Master Out Slave In): This line allows the master to transmit data to the slave.
- **MISO** (Master In Slave Out): This line allows the slave to transmit data to the master.
- **SCK** (SPI Serial ClocK): Clock signal, generated by the master, which synchronizes the transmission. The frequency of this signal is set by the master and is programmable.
- **SS** (Slave Select): This signal is used to individually select (address) a slave. There are as many SS lines as there are slaves on the bus. The possible number of SS connections of the master will therefore limit the number of slaves.

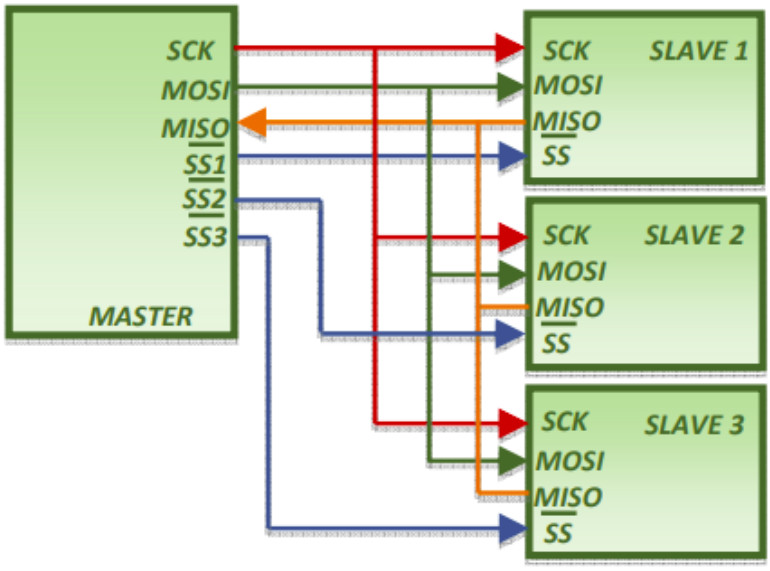


Fig. 1: Serial bus for SCLK, MISO and MOSI lines. Direct link for SS lines between Master and each Slave

At the level of the message exchange process:

1. The Master activates the clocking of the clock
2. The Master selects a Slave via the SS line (converter, shift register, memory, ...). Each Slave is active only when selected.
3. Data can then be exchanged in both between Master and Slave using the MISO and MOSI lines.

Like the I2C bus, SPI operates in Master-Slave mode. On the other hand, it does not offer a standardized acknowledgement mechanism between Master and Slave. SPI is also more flexible

than I2C on the number of bits to be transmitted per message and offers a higher data rate up to 20 Mbits/s.

2.2.3b Use cases

- Exchanging data between MCUs and radio chips
- Reading and writing data in an EEPROM flash memory, linked to the MCU using an SPI bus

2.2.4 Overview and elements of choice

UARTs offer the flexibility of point-to-point, bi-directional communication, which can come from either of the two extremities. However, the format of the frames is relatively rigid, and speeds can be insufficient in certain cases.

Unlike UARTs, I2C and SPI data buses enable data exchange at much faster speeds. However, these both require a Master/Slave operating model, which can be an obstacle when it comes to initiating communication. The I2C protocol specifies the data format and ensures secure exchanges through acknowledgement. SPI, on the other hand, does not specify any message format in particular, leaving this entirely up to the designer of the application. This makes SPI much more flexible when it comes to transporting data, in addition to being much quicker.

In terms of the number of components supported, I2C supports 128 addresses, divided up into series of three lines. The Master of the SPI bus must have the right number of pins for the SS lines linked to each of their Slaves.

2.3. Connected objects: IoT-LAB M3

- 2.3.0. Introduction
- 2.3.1. The connected object IoT-LAB M3
- 2.3.2. The IoT-LAB gateway
- TP2. First IoT-Lab experiment

2.3.0. Introduction

In this sequence, we will take a detailed look at the components and characteristics of the IoT-LAB M3 and A8 connected objects used on the FIT IoT-LAB platform, as well as how you can access them. You will learn how to analyse the different components of the FIT IoT-LAB's main connected objects and how to access them remotely.

When designing the FIT IoT-LAB platform, we also designed our own hardware. We will detail here the hardware architecture of the IoT-LAB M3 board, which has been deployed in large numbers to allow large-scale experiments. We will also briefly explain how we achieved the hardware integration of the experiment boards into the platform for remote use.

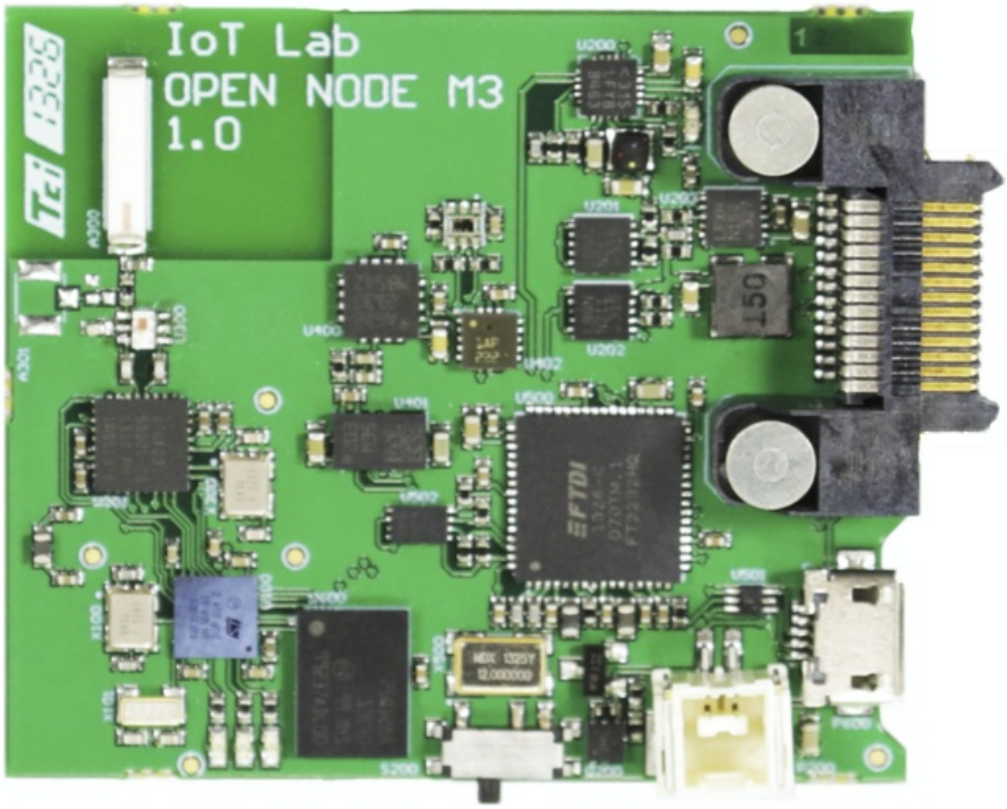


Fig. 1: Carte IoT-LAB M3

Video presentation of the connected objects of the FIT IoT-LAB platform: What are the embedded components? How can we interact remotely with these objects?

2.3.1. The connected object IoT-LAB M3

Content

2.3.1a Synoptic

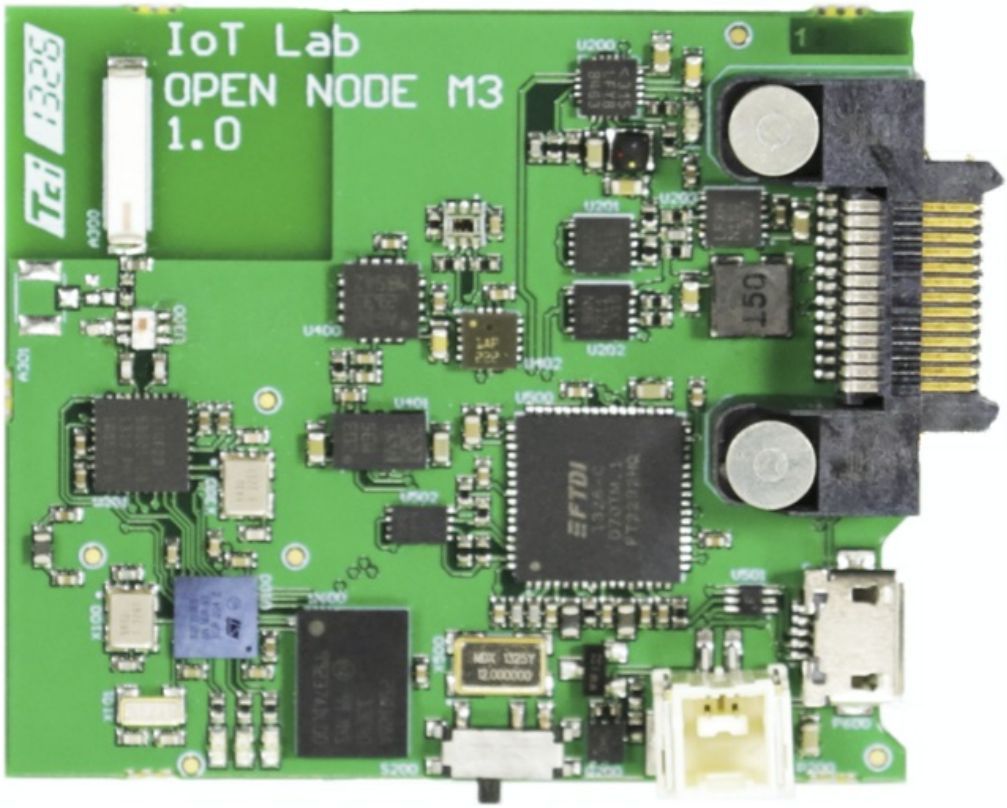


Fig. 1: IoT-LAB M3 board

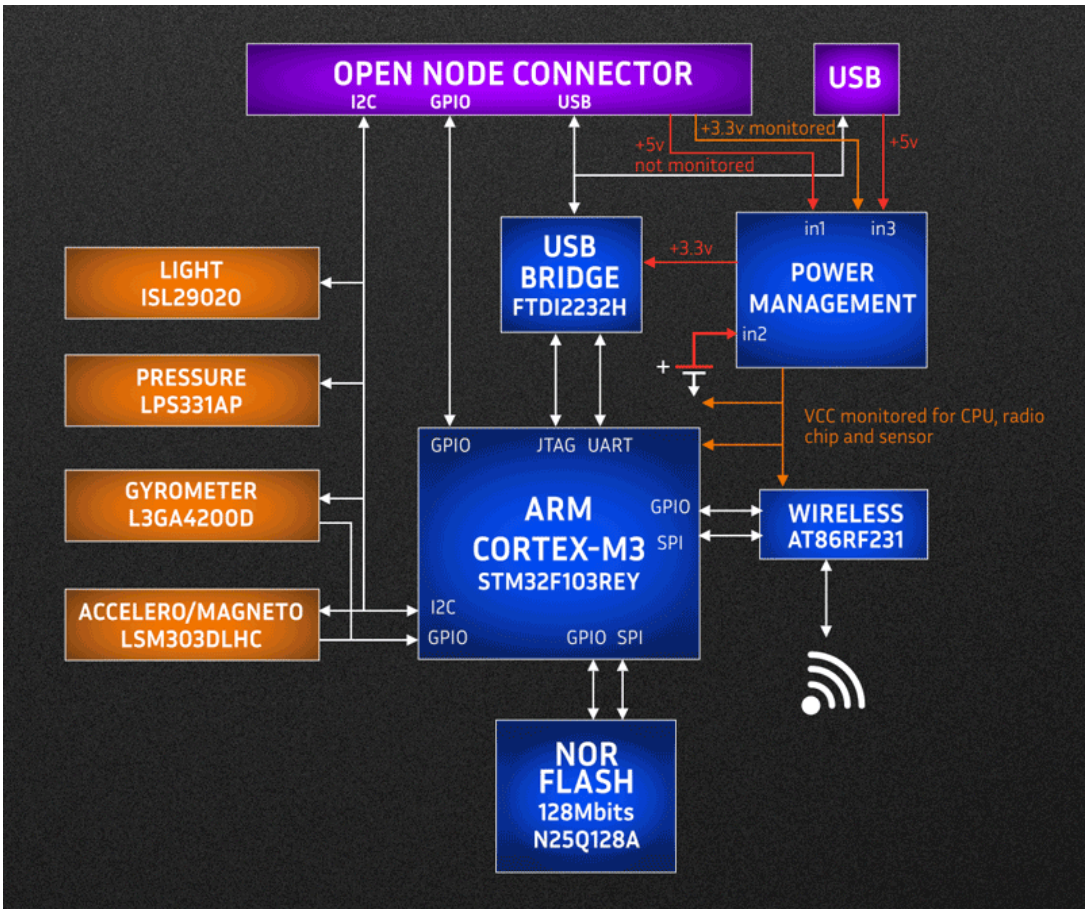


Fig. 2: IoT-LAB M3 architecture

2.3.1b MCU

The connected object **IoT-LAB M3** takes its name from the **CPU** it is equipped with: the Cortex M3 manufactured by ARM. The Cortex M3 is a 32-bit CPU set to a maximum speed of 72 MHz. The Cortex M3 is integrated into the STM32 **MCU** (reference STM32F103REY) manufactured by STMicroelectronics, more commonly known as ST. The STM32 MCU is equipped with 64 KB of **RAM** and 256 KB of **ROM**. These technical specifications put the connected object IoT-LAB M3 in the category of **constrained objects** employing the use of embedded operating systems. An example of a connected object employing the use of this same MCU is the Apple TV 4 remote control.

In terms of its dimensions, the M3 has a *form factor* (i.e. the CAD design of the embedded system) 4 cm wide and 5 cm long. It should be borne in mind that the object IoT-LAB M3 was designed as a generic development board. Depending on the use case, a manufacturer could further optimise the CAD in order to reduce components to the size of a thumbnail.

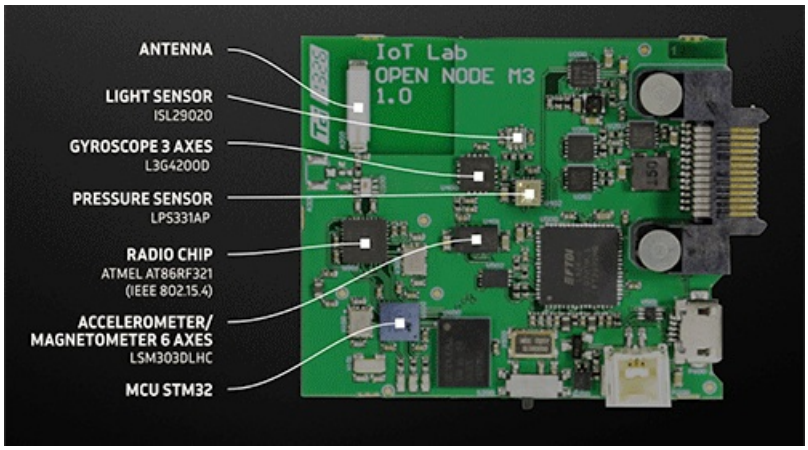


Fig. 3: External IoT-LAB M3 components

2.3.1c External hardware

In order to communicate with the different external components, the object IoT-LAB M3 uses the data buses **I2C** and **SPI** of its SMT32 MCU.

Radio chips

The radio chip AT86RF231 manufactured by Atmel (purchased in 2016 by Microchip) communicates over the 2.4 GHz ISM frequency band. This radio chip was designed to implement the MAC layer of the IEEE 802.15.4 standard, used by the Zigbee communication protocol, among others. The maximum bandwidth is 256 kbits/s. A small, 9 mm long ceramic antenna is welded onto the circuit. This type of radio technology falls into the "short-range" category, with maximum distances of 40/50 meters indoors and up to around 100 meters outdoors. Depending on these technical specifications, the radio chip consumes up to 14 mA when transmitting at maximum power. It is interconnected with the MCU on the SPI bus, enabling rapid data exchange. It is also connected to a GPIO port on the MCU in order to control the waking up of the radio chip in the event of it being set to standby mode.

External NOR flash memory

A 128-Mbits external NOR flash memory (N25Q128A13E1240F) is connected to the MCU via the SPI bus in order to enable rapid data transfer. This memory is useful when it comes to storing data acquired by sensors while a program is running. Another use case is for OTA (Over The Air) updates to the MCU's firmware. Some embedded operating systems partition this ROM memory in order to download and store different versions of the firmware.

Sensors

4 sensors connected to the MCU via the I2C bus are embedded into the IoT-LAB M3:

- the light sensor ISL29020 : this measures ambient light intensity in lux.
- the pressure sensor LPS331AP: this measures atmospheric pressure in hPa.
- the accelerometer/magnetometer LSM303DLHC: this provides feedback on an object's acceleration, and can be used to detect movement. By determining a threshold, it generates a change of state on one of the MCU's digital inputs/outputs in order to create an **interrupt**, which can be used to bring the MCU out of standby mode.

- the gyroscope L3G4200D: this measures the orientation of an object in space and can be used, for example, to determine the orientation of the screen of a tablet or a smartphone.

Actuators

3 LEDs (red, green and orange) are connected to the MCU via the **digital inputs/outputs** on the IoT-LAB M3. Locally, these are used to notify MCU actions or states. Remotely, on the IoT-LAB experiment platform, these LEDs enable us to illustrate the monitoring of energy consumption by changing their state periodically (on/off).

2.3.1d Power supply

The M3 node can be supplied in 3.3 V in two different ways:

- via the USB port
- via the external battery

In terms of the energy consumption of the different hardware devices, it should be noted that, at full power, the MCU consumes 14 mA, to which we must add the energy consumption of the other hardware devices. The radio chip, for example, consumes 14 mA when transmitting and 12 mA when receiving.

Typical use cases for constrained connected objects often require the use of a battery. In order to maximise the lifespan of objects, the energy-saving strategy employed involves switching the MCU and its various different components to standby mode once the object has finished processing. In the case of the STM32 MCU, consumption is reduced to 1 mA when in sleep mode, and remains below 1 µA in standby mode. The AT86RF231 radio chip supports the same type of features, with comparable energy savings.

2.3.1e Programming

The JTAG interface can be used to reprogram and debug the MCU from a computer. It can be accessed via the FTDI component, which is located behind the USB and IoT-LAB connectors. The latter is used for the purposes of integrating the IoT-LAB M3 object into the IoT-LAB platform. This FTDI component also features an UART port, which displays messages on the user's terminal.

2.3.2. The IoT-LAB gateway

Content

Hardware integration

The connected object shown previously is typically identical to IoT development boards commonly connected to a computer via a USB port for programming and debugging. In a context of experimenting with applications or network protocols on a large scale, it becomes tedious to reprogram a dozen or even a hundred objects in this way.

The FIT IoT-LAB platform provides a solution to this problem by integrating each experimental board called **Open Node** (ON) behind a mini-computer called**Gateway** (GW). The term Gateway is used here, in a hardware sense, as it links the experimental board with the other hardware components of the platform. This gateway will be in charge of programming the experimental board. In addition to this intermediate gateway, an autonomous on-board system called **Control Node** (CN) is able to monitor the*Open Node* to measure its energy consumption, radio noise and traffic. The advantage of using a dedicated MCU is to guarantee real-time execution of these measurements. Finally, the complete IoT-LAB node is connected via a private network to a *IoT-LAB site server* which will be able to manage it remotely.

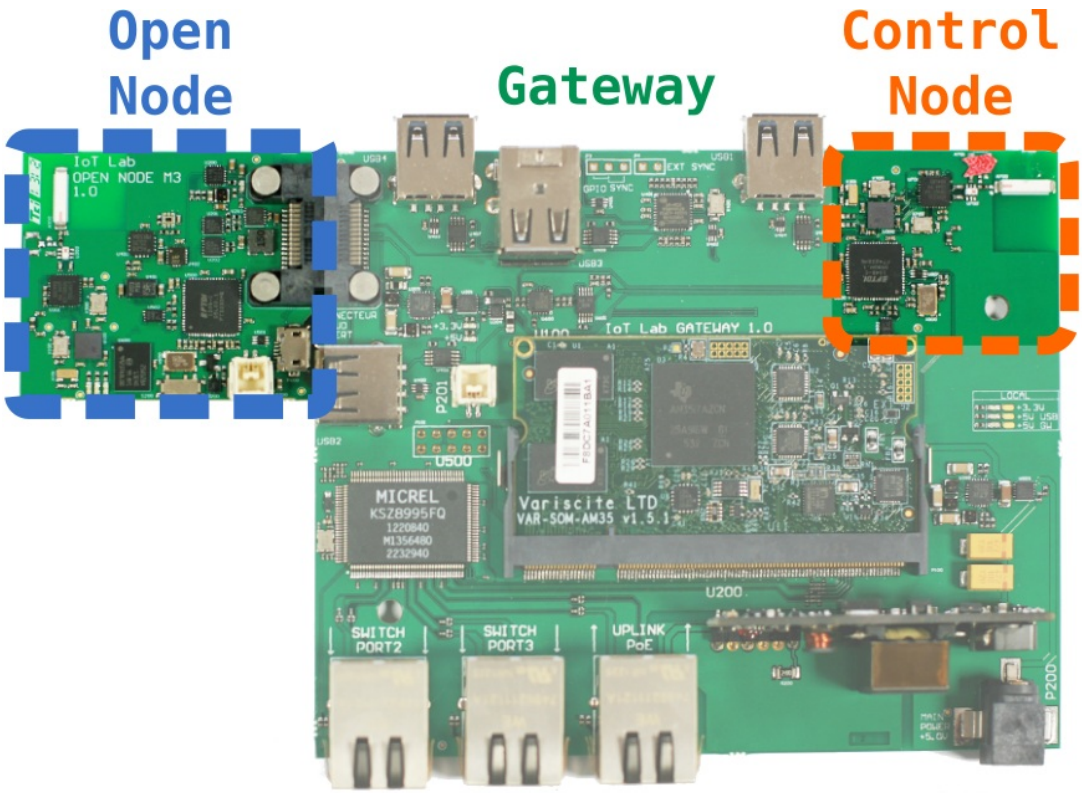


Fig. 1: Complete IoT-LAB node = ON + GW + CN

Remote interaction

To facilitate deployment, the Gateway includes a PoE (Power Over Ethernet) module which, as its name suggests, allows power to be supplied via the Ethernet port in addition to the network link. Being the only link to the hardware, it is also through this link that the usual interaction links with the experimental board pass:

- the serial link of the Open Node is redirected to a TCP socket served by the Gateway on port 20000;
- the Open Node debug link is redirected to a second TCP socket served by the Gateway on port 3333;

as well as the data sent back by the Control Node:

- the sniffer data is written to a TCP socket served by the Gateway on port 30000;
- the automatic monitoring data is written directly to files in the user's workspace.

Dynamic network filtering rules make these links accessible to the user who has reserved these experimentation nodes, from the SSH frontend of the site concerned, for the duration of the experiment.

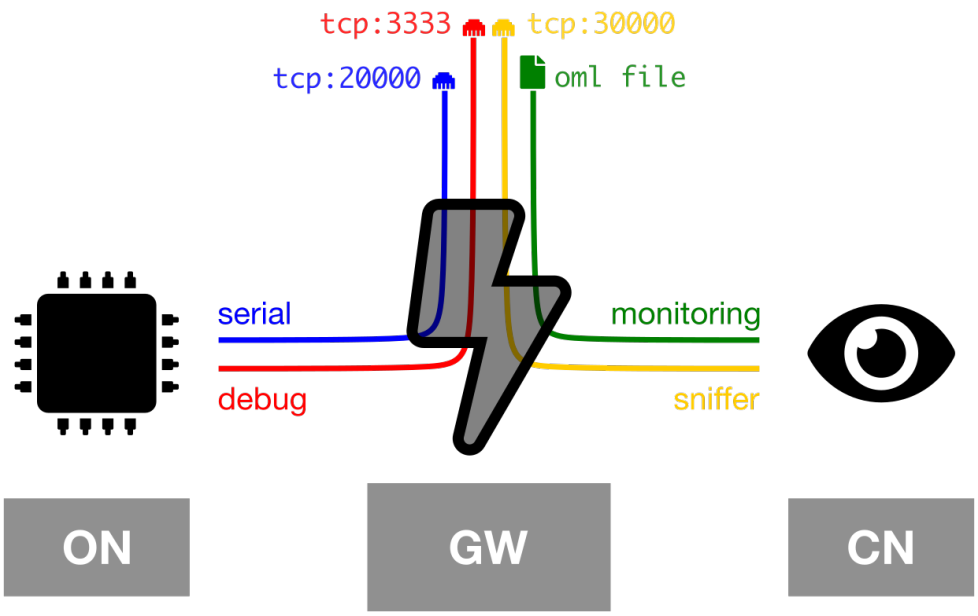


Fig. 2: Links redirected by the Gateway board

Visit the IoT-LAB website for more information about theIoT-LAB Gateway.

Module 3. Focus on Embedded Softwares

Objective: At the end of this module you will be able to apply the specific programming principles for a connected object. You will also be able to describe the characteristics of the RIOT operating system.

Hands-on activities (TP) : TP3: First RIOT application TP4: Extending a RIOT application TP5: Managing threads with RIOT TP6: Use timers TP7: Use sensors on the IoT-LAB M3 board

Contents of Module 3

3.1. Software Solutions For Writing an Application

- 3.1.0. Software solutions for programming connected objects
- 3.1.1. Programming a connected object
- 3.1.2. What is an operating system (OS) and how to choose the right one?
- 3.1.3. The main existing operating systems
- 3.1.4. Higher level solutions

3.2. Discovering RIOT

- 3.2.0. Introduction
- 3.2.1. RIOT characteristics
- 3.2.2. Source code structure
- 3.2.3. The build system
- TP3. First RIOT application
- TP4. Extending a RIOT application

3.3. RIOT Architecture

- 3.3.0. Introduction
- 3.3.1. The kernel and how it is used
- 3.3.2. Scheduling and Threads
- TP5. Managing threads with RIOT
- 3.3.3. Advanced use of threads
- 3.3.4. Power management

3.4. APIs hardware

- 3.4.0. Introduction
- 3.4.1. The hardware abstraction layer
- 3.4.2. Timers
- TP6. Use timers
- 3.4.3. Interacting with GPIOs
- 3.4.4. High-level drivers

3.5. Evaluation par les pairs de votre application IoT

- 3.5.0 Introduction
- 3.5.1. Presentation of "Peer Evaluation" *(not included)*

- 3.5.2. Scoring scale *(not included)*
- TP7. Use sensors on the IoT-LAB M3 board
- 3.5.3. RIOT application on FIT IoT-LAB
- 3.5.4. Submit your activity and evaluate your peers *(not included)*

3.1. Software Solutions For Writing an Application

3.1.0. Software solutions for programming connected objects

During this sequence, you will discover what is involved in programming a connected object and the software solutions used to write IoT applications.

References:

- https://hal.inria.fr/hal-01245551/document
- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8308310&tag=1
- https://www.osrtos.com/

3.1.1. Programming a connected object

A microprogram (*firmware*) is a program for a microcontroller which contains the application code, the *bootloader* code, the code needed to interface with the hardware, the code for the *kernel*, and the code for the libraries used. If we were to make an analogy with computers, installing a microprogram is like wiping a hard-drive, flashing the BIOS and reinstalling both the operating system and the application.

Generally speaking, microcontrollers are no longer programmed in assembly language but in a higher level language (C, C++, etc.). This source code is compiled in machine language, and must then be sent to the microcontroller (*flashed*) by a programmer (e.g. OpenOCD for the ARM and MIPS architectures and AVRDUDE for AVR Microchip (formerly Atmel) microcontrollers).

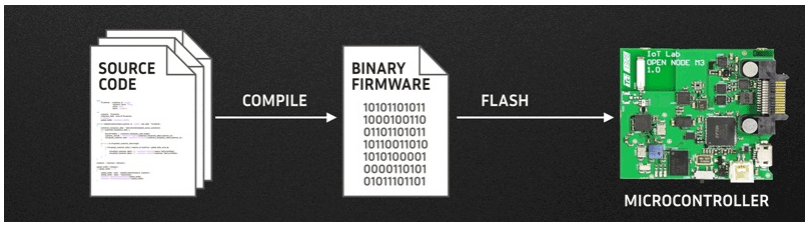


Fig. 1: The source code is compiled and then flashed

In order to write a microprogram, you will need the code for the application, the code for managing communications and the code for the input/output devices used (displays, motors, sensors, etc.). You will also need to adopt a strategy for managing memory, optimise energy consumption, guarantee its security and support a wide range of hardware targets.

As you will understand, the wide variety of connected objects, material constraints, the need for interoperability and security are all reasons for using an operating system as opposed to starting from scratch.

3.1.2. What is an operating system (OS) and how to choose the right one?

Content

This section outlines the characteristics you would expect to find in an operating system and the key design points that will help you choose the right one.

3.1.2a The main characteristics of operating systems

Memory management

The amount of space taken up by the OS in terms of random-access memory (RAM) and storage (ROM) is an important factor, given the limitations of the connected objects that we'll be dealing with.

Hardware support

Connected objects come in all shapes and sizes. Finding a way of supporting the different types of microcontrollers, the different hardware devices and other sensors can quickly become overwhelming, making the OS's capacity to provide hardware abstraction an important consideration.

Network connectivity

Given the constraints of wireless communication (e.g. radio interference) and managing power consumption, operating systems must be capable of offering different network protocol stacks for different uses, in addition to being able to easily integrate new ones.

Efficient power use

Some applications require an object to be able to run off a battery for multiple years without being recharged. The OS must therefore provide developers with ways of limiting the power consumption of their applications, while also using as little power as possible.

Real-time capabilities

An OS is said to be 'real-time' (a Real-Time Operating System or RTOS) if it is capable of processing inputs in a fixed time. This characteristic is needed for embedded systems with constraints in terms of response time.

Security

Protecting data and guarding against external attacks is a major challenge facing connected objects. The OS must therefore provide the mechanisms needed to develop secure applications (cryptography libraries and security protocols) and to ensure that they remain secure in the long run (e.g. using remote updates).

3.1.2b Key design points

Kernel

The kernel of the operating system is responsible for loading and executing processes, in addition to providing the hardware abstraction mechanisms. There are a number of different approaches:

- the microkernel approach (µ-kernel), which is more robust and more flexible
- the monolithic approach, which is less complex and more efficient
- hybrid approaches, which sit somewhere between the two previous approaches

Schedulers

Schedulers have an impact on power consumption, real-time capabilities and programming methods. Operating systems can offer several types of schedulers, but only one is used for execution:

- preemptive schedulers assign CPU time to each task
- cooperative schedulers give each task the responsibility of "returning control"

Memory allocation

Given the limited memory of connected objects, how this memory is managed is extremely important:

- static allocation requires knowing the quantity of memory needed in advance
- dynamic allocation fragments memory, resulting in scenarios where there is no more memory available

Managing network buffers

The memory for managing packets in the network protocol stack may be:

- allocated per layer (consuming more power)
- changed to reference (more complex)

Programming models

Operating systems can either be:

- *multithreaded*, where each task has its own context
- event-driven: each task is triggered by an external event, such as an interrupt

Programming languages

Proprietary languages may be better suited to developing connected objects, making them more secure and efficient; while standard languages (such as C or C++) facilitate portability, enabling standard debugging tools to be used.

Functions

OS functions are provided by both the kernel (the scheduler, synchronization mechanisms, etc.) and by different libraries (network, security, etc.).

Tests

As is the case with all software, tests play a crucial role when it comes to ensuring the stability of the OS.

Certification

For some critical applications, certification may be required. This is costly to implement, however, given that it must be performed by a third-party body; as a result, not all projects are able to certify their versions.

Documentation

Full, up-to-date and easily understandable documentation is essential for an OS in that this is the basic method for writing an application.

Code maturity

Code maturity is difficult to estimate. It depends on the age of the project, the number of contributors and the number of users, among other factors.

Licensing

Licenses can be divided into three categories:

- Proprietary licenses, which limit access to the source code and, therefore, to the number of contributors.
- *Copyleft* licenses (such as GPL), which force developers to share modified versions with the rest of the community. These licenses are generally less popular among manufacturers.
- Permissive licenses (BSD, MIT, Apache, etc.) grant access to the source code, giving users more freedom, but this can result in the community becoming fragmented.

Support

The *open source* OS community provides support with the best possible quality of service (best effort) via forums, mailing lists and bug tracking tools. Some operating systems, such as ARM Mbed, offer commercial support, ensuring maximum response delay to questions (eg. two working days) or for bug correction.

3.1.3. The main existing operating systems

Content

3.1.3a A history of operating systems used in the IoT

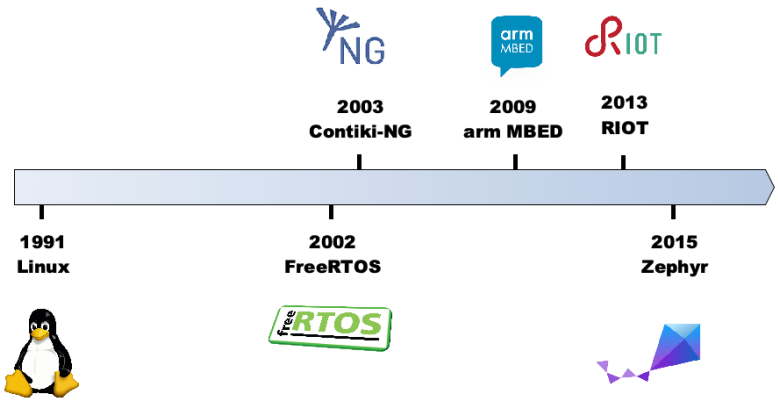


Fig. 1: Start dates of the main IoT OS.

3.1.3b RIOT

- License: LGPL
- Supported architectures: AVR, ARM, MIPS32, MSP430, PIC32, RISC-V, x86, ...
- Main contributors: Freie Universität Berlin, Inria, Hamburg University of Applied Sciences
- Main characteristics:
  - RTOS
  - Networks: Bluetooth LE, WPAN, LPWAN, Wi-Fi/Ethernet
  - Minimum memory footprint: ~ 1.5kB RAM, ~ 5kB ROM (1)
- First version: 2013
- Number of contributors over the last 12 months: 108(2)

3.1.3c Zephyr

- License: Apache 2.0
- Supported architectures: ARM, RISC-V, x86
- Main contributors: Linux Foundation Project (Intel, NXP, Oticon, Nordic Semiconductor, ST, Linaro)
- Main characteristics:
  - RTOS
  - Networks: Bluetooth LE, WPAN, Wi-Fi/Ethernet
  - Minimum memory footprint: "fitting in devices with at least 16k RAM" (3)
- First version: 2015
- Number of contributors over the last 12 months: 66(2)

3.1.3d Contiki-NG

- License: BSD
- Supported architectures: ARM, AVR, x86, MSP430, PIC32
- Main contributors: Atmel, Cisco, ETH, Redwire LLC, SAP, Thingsquare
- Main characteristics:
  - Protothreads
  - Networks: Bluetooth LE, WPAN, WiFi/Ethernet
  - Minimum memory footprint: 10 KB of RAM, 100 KB of ROM (4)
- First version: 2003
- Number of contributors over the last 12 months: 17 (2)

3.1.3e ARM Mbed

- License: Apache 2.0
- Supported architectures: ARM Cortex-M
- Main contributors: ARM
- Main characteristics:
  - RTOS
  - Networks: Bluetooth LE, WPAN, LPWAN, Wi-Fi/Ethernet
  - Minimum memory: 8 KB of RAM, 14 KB of ROM (5)
- First version: 2009

3.1.3f FreeRTOS

- License: MIT
- Supported architectures: AVR, MSP430, ARM, x86, 8052, MIPS
- Main contributors: Amazon
- Main characteristics:
  - RTOS
  - Networks: Bluetooth LE, WPAN, Wi-Fi/Ethernet
  - Minimum memory: 4 KB of RAM, 9 KB of ROM (6)
- First version: 2002
- Number of contributors over the last 12 months: 3 (2)

1: http://riot-os.org/ 2: https://www.openhub.ne, http://github.com 3: https://docs.zephyrproject.org/latest/introduction/index.html 4: https://github.com/contiki-ng/contiki-ng/wiki 5: https://os.mbed.com/blog/entry/Optimizing-memory-usage-in-mbed-OS-52/ 6: https://www.freertos.org/index.html

3.1.4. Higher level solutions

Content

Although it is necessary to develop in C with many operating systems, there are other solutions, which we will take a look at here.

3.1.4a C++

C++ has endured a bad reputation of late, but the majority of functions have no impact on either size or speed. As is the case with C, it is necessary to understand what happens at machine code level in order to use C++ effectively within an embedded system.

3.1.4b Rust

Rust, a language developed by Mozilla, is also used to develop microprograms. It was designed to be concurrent and safe, and is interoperable with code written in C. The Rust compiler supports the following architectures: ARM, MIPS and RISC-V.

3.1.4c Java

Some Java virtual machines can be used to execute Java bytecode on microcontrollers.

3.1.4d Interpreted languages

C is more difficult to use than interpreted languages such as Python or Javascript. It is for this reason that there are interpreters for IoT-adapted versions of these languages (Micropython and Jerryscript). As is often the case, there is no ideal solution, and gains in terms of productivity and accessibility may be offset by losses in terms of performance and memory.

There are many operating systems and languages used for the development of connected objects. We will take a closer look at the RIOT operating system in the next sequence, both because it is supported by the FIT/IoT-LAB platform and because authors of this MOOC are contributors of this project.

3.2. Discovering RIOT

3.2.0. Discovering RIOT

A video presentation of the RIOT project and the characteristics of this operating system for microcontrollers.

This sequence will introduce you to the RIOT operating system, which is widely-used for programming microcontroller-based connected objects. We will present a general overview of this project before taking a look at its main characteristics.

3.2.1. RIOT characteristics

Content

3.2.1a System characteristics

RIOT has all the characteristics you would expect from an operating system designed to operate on a microcontroller: it functions in real-time and takes up very little memory space. Generally speaking, microcontrollers are highly constrained in terms of memory (kilobytes of RAM and ROM) and are much slower than modern microprocessors: they operate at frequencies in MHz, while microprocessors operate at frequencies in GHz. Another key characteristic of microcontrollers is their ability to “go to sleep”, enabling them to minimize their power consumption: this is reduced to just a few microamperes in the most extreme cases. An object operating on a microcontroller is thus able to run off a battery for several months, or even several years. As we will see later on, RIOT provides an original mechanism for managing the sleep state of microcontrollers and thus minimizing their power consumption.

In order to get the most out of microcontrollers, the RIOT operating system is based around a so-called **micro-kernel** architecture. This micro-kernel contains only the building blocks needed in order for the system to function:

- **A multi-task system**, which makes it possible to have more than one task, also called execution context, on the same microcontroller. Execution contexts are totally independent of each other: each manages its own memory space. These tasks/execution contexts are commonly known as **threads**.
- **A real-time scheduler**, which is responsible for switching between execution contexts depending on their status. This scheduler is referred to as *tickless*, owing to the fact that the rescheduling takes place following a hardware interrupt and not via busy waiting (e.g. busy loops).
- **A mechanism for communication between tasks** used to exchange information in the form of messages between different execution contexts.
- **Synchronization mechanisms** for dealing with concurrency in cases where multiple threads are attempting to access shared resources for writing/reading, such as memory variables or the microcontroller's hardware. Two such synchronization mechanisms are mutex and semaphore.

In its most basic configuration, RIOT only requires 2.8 KB of RAM and 3.2 KB of ROM.

The structure of a RIOT application is closely related to the*thread* concept. In its most simple version, of at least two *threads*:

- The **main thread** in which the main function of the application's program is executed.
- The **idle thread**, an execution context with the lowest priority that the system will switch to in situations where all the other threads are either blocked or exited. This thread is also

responsible for managing power consumption modes. If all the other threads have finished their tasks, this means that the system no longer has anything to do and can therefore switch to sleep mode.

The mechanism for communication between *threads* is also useful when it comes to effectively dealing with external interrupts, without any risk of losing an intermediary state. Given that interrupts can happen at any time, it is vital to be able to guarantee the status of the system once this interrupt has been handled. The efficient way in which interrupts are handled in RIOT involves a message being sent to the thread responsible for processing the interrupt from the latter's context. This enables the interrupt to be handled within a secure context (that of the *thread* receiving the message), without any loss of status in the system.

This is also known as *soft* real-time: the system will ensure that all external events (interrupts) are processed, but cannot guarantee that they will be treated immediately.

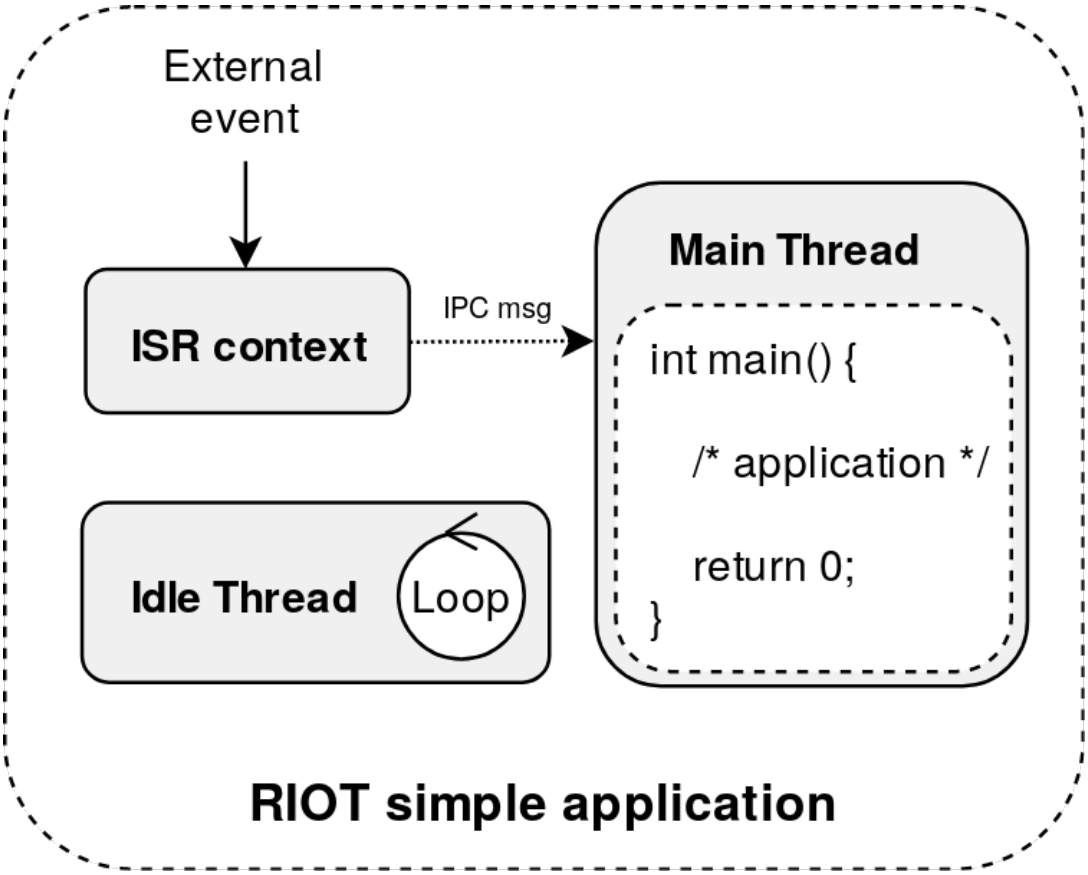


Fig. 1: Structure of a RIOT application, © A. Abadie

3.2.1b The hardware abstraction layer

A hardware abstraction layer is added to this real-time micro-kernel, enabling RIOT to execute on a wide range of hardware targets.

This hardware abstraction layer is divided into 4 levels:

1. **The microcontroller level**, the lowest level of block as it touches the core of the object. This functional block is where certain elements common to all platforms are defined, enabling applications to know which functions are supplied by a microcontroller (random number

generators, writing on flash, EEPROM, etc.).

2. **The board level**, which is where certain variables (or macros) common to all platforms are defined. Some boards, for example, feature access to certain serial buses (SPI, I2C) with their configuration, while others don't. A specific microcontroller is defined for each board.

3. **The hardware level**: RIOT defines generic APIs for the main types of hardware available to microcontrollers, i.e. UART, SPI, I2C, PWM, etc. These are generic APIs which can be used to write code that is compatible with different hardware platforms without the need for modification.

4. **The driver level**, which is where all radio drivers, sensor drivers and actuator drivers can be found. This is the layer which depends on higher level hardware since it in itself is heavily reliant on the *hardware* layer when it comes to accessing different platforms generically.

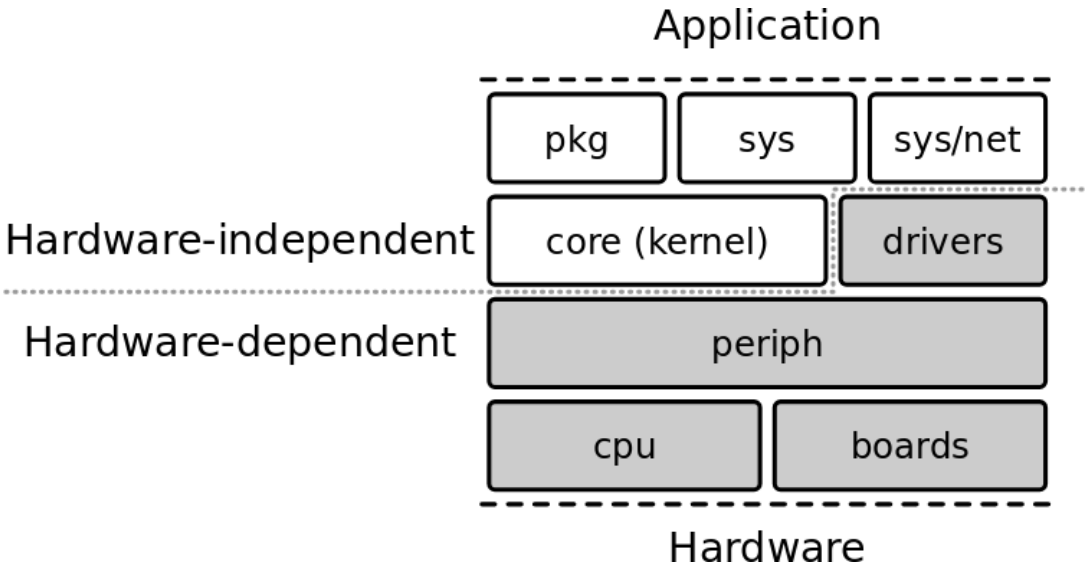


Fig. 2: General structure of RIOT

Source: E. Baccelli et al. (2018) "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT", IEEE Internet of Things Journal. PP. 1-1. 10.1109/JIOT.2018.2815038

When compiling a RIOT application, you must always specify the target for which the firmware is to be produced. This gives the following mapping: one application ⇒ one board ⇒ one microcontroller model.

There is now an extensive range of hardware support for RIOT thanks to this hardware abstraction layer: the system is capable of operating on a number of hardware architectures, from 8-bit to 32-bit, and on ARM, AVR, MIPS, RISC-V and Xtensa. What this means is that the system can be used on boards produced by the main manufacturers: Microchip, NXP, STMicroelectronics, Nordic, TI, Espressif, etc.

3.2.1c A modular system

As we saw previously, RIOT is based on a micro-kernel architecture, to which developers add the modules needed for their application. This modular mechanism means you **only need to include what is actually necessary** for an application, thus limiting the microcontroller's memory requirement (in kB).

RIOT's system provides a whole ecosystem of modules that can be used to develop applications quickly and easily, without having to reinvent the wheel.

These modules are split into multiple categories:

- **System libraries**: these modules are fully independent of the hardware, supplying features that can be used to develop an application. The **shell** module, for example, provides an interface that makes it easy to implement a command interpreter via the standard input-output (stdio), generally using the board's serial link (UART). There are also cryptography modules, and a highly-efficient (in terms of memory) formatting module for string. The **xtimer** module provides advanced features for adding delays or actions (*callbacks*) at different moments during the execution of the application.
- **Actuator and sensor drivers**: these modules are generally based on the HAL (Hardware Abstraction Layer), meaning they can be used on any type of microcontroller. They are highly practical, making it easy to add sensor reading, motor control or an external display to an application, without having to write this code yourself. This category also includes drivers for external storage media, such as SD cards or communication interface drivers (radio, CAN, etc.).

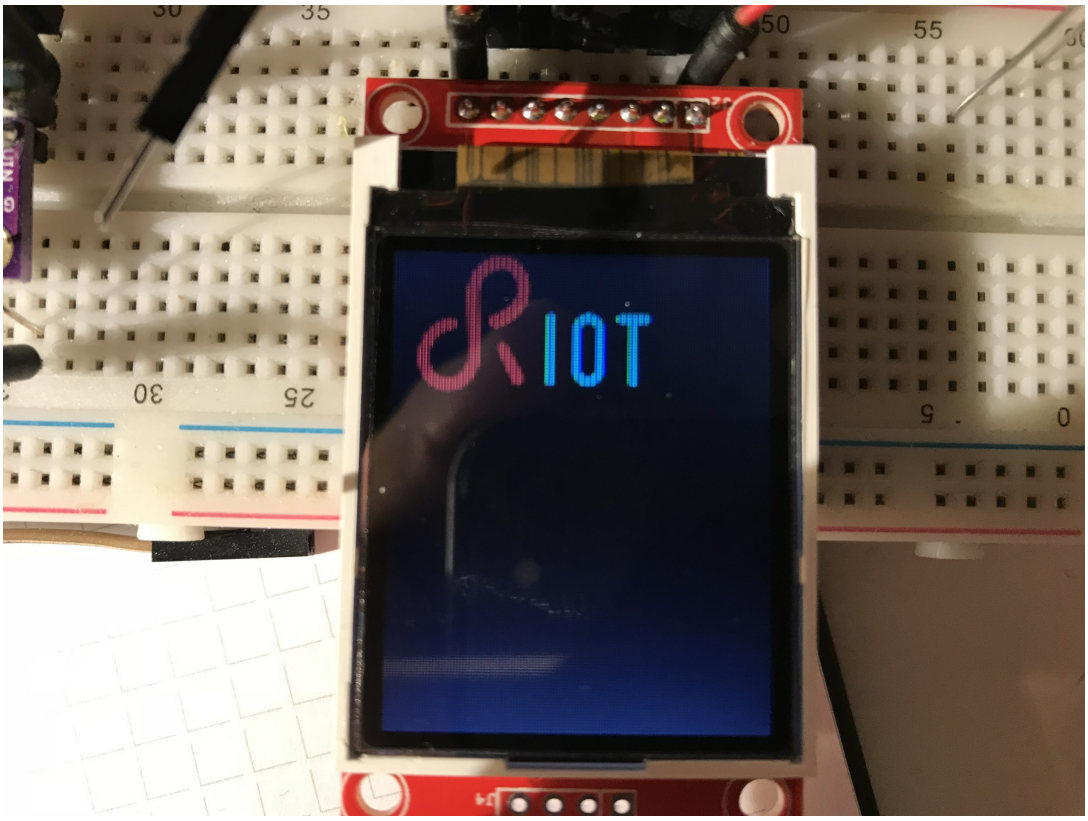


Fig. 3: Graphical user interface using the Ucglib package

- **Network protocol** modules are actually a sub-category of system libraries, supplying implementations for protocols currently used in the IoT: CoAP, MQTT-SN, etc. We will return to this in more detail in module 4, which deals exclusively with "Network communication".

- **External packages** are modules used to import source codes external to RIOT into an application. For maintenance reasons of the code base, it is not possible to copy as-is the whole code from another project. However, it is always useful to be able to integrate it easily into its application. All kinds of projects can be found in external RIOT packages: embedded interpreters for Javascript or LUA, file systems such as LittleFS or fatfs, complete networking stacks, such as Iwip or OpenThread, but also cryptography libraries, graphic libraries and even machine learning libraries (uTensor, TensorFlow-Lite).

Package	Overall Diff Size	Relative Diff Size
ccn-lite	517 lines	1.6 %
libfixmath	34 lines	0.2 %
lwip	767 lines	1.3 %
micro-ecc	14 lines	0.8 %
spiffs	284 lines	5.5 %
tweetnacl	33 lines	3.3 %
u8g2	421 lines	0.3 %

Fig. 4: Comparison of required lines of code changes when porting an external package to RIOT

Source: E. Baccelli et al. (2018) "RIOT:an Open Source Operating System for Low-end Embedded Devices in the IoT", IEEE Internet of Things Journal. PP. 1-1. 10.1109/JIOT.2018.2815038

The RIOT build system is also responsible for resolving dependencies between modules in order to ensure the consistency of the generated application: for example, to load a driver that requires the I2C bus and the `xtimer` system, all you have to do is to load the module corresponding to this driver and the build system will automatically load the modules required for `I2C` and `xtimer`.

3.2.1d Network protocol stacks

RIOT supports the main network protocols currently used for the Internet of Things through what are known as **network protocol stacks**. They're called stacks because they implement the different layers of the OSI (**O**pen **S**ystems **I**nterconnection) model defined in order to make items of IT equipment communicate.

Some of these network stacks are grouped together in a set designed for IP-oriented (Internet Protocol) communication. This set implements protocols that are compatible with the current internet, making it possible for objects to communicate with other items of equipment over the internet (such as computers, smartphones, etc.). These IP-oriented network stacks can be used with wired communication technology, like Ethernet, or radio, such as Wi-Fi, 802.15.4 or BLE. We will take a more detailed look at these protocols in module 4 “Network communication”.

The network stacks supplied by RIOT are either supplied natively, like **GNRC** (`GeNeRiC`), or as external packages, like **lwIP**, or **OpenThread** (an open-source implementation of the **Thread** specifications).

Lastly, **other network stacks** are also available:

- For the **CAN** (Controller Area Network) protocol, which is widely used in the automotive industry. This stack is implemented natively in RIOT.
- For support for ***Bluetooth Low Energy*** (BLE). This stack is provided as a package from the NimBLE project.

- For support for **LoRaWAN** networks. This stack is provided as a package from the Loramac-node project.

3.2.2. Source code structure

As we saw before in this sequence and as is shown by the figure below, RIOT's architecture is based around a number of essential components:

- its **hardware abstraction layer**, which is where you will find all the code that depends on the hardware and provides the shared programming interfaces
- its **kernel**, which manages the different execution contexts, communication between tasks, priorities, etc.
- **system libraries** (`xtimer` , etc.), **packages**, etc.

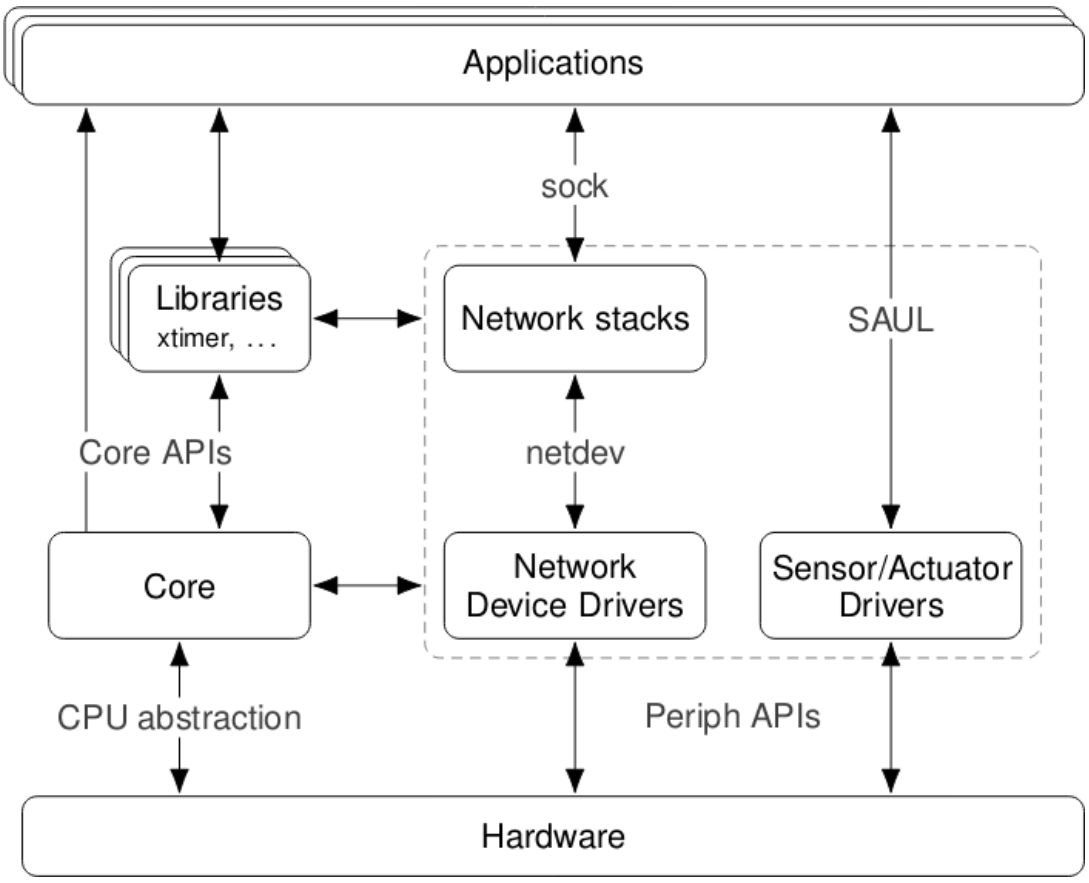


Fig. 1: Interaction between APIs in RIOT

Source: E. Baccelli et al. (2018) "RIOT:an Open Source Operating System for Low-end Embedded Devices in the IoT", IEEE Internet of Things Journal. PP. 1-1. 10.1109/JIOT.2018.2815038

Other components provide higher abstraction levels, making it easier to integrate more complex functions:

- the `netdev` API acts as a bridge between heterogeneous communications drivers (radio, Ethernet, etc.) and protocol drivers (such as GNRC, LwIP or OpenThread)
- the `SAUL` (Sensor Actuator Uber Layer) API provides a common interface enabling data from similar sensors to be read (temperature, pressure, etc.).

The **source code** is organized as follows:

- **boards:** contains the code specific to the support for the boards. The code for each supported

board can be found in its own directory. For each board, the CPU model used, the way in which internal clocks are configured and the default configurations for devices (UARTs, SPIs, etc.) are all defined. There is also the default configuration for the serial port (e.g. if `/dev/ttyACM0` is created when the board is plugged into a computer via one of its USB ports) and the configuration relating to the programming tools (OpenOCD, AVRdude, etc.)

- **core:** contains the code relating to the kernel, to*thread* management and to inter-process communication.
- **cpu:** contains the code specific to the support for the microcontrollers (vendor header file, definitions, internal device drivers, etc.). This is also where the CPU initialization sequence (or entry point) is implemented for each architecture. This function, for example, is called *reset_handler_default* for the ARM CortexM architecture.
- **dist:** contains the code for RIOT helper tools. These tools are normally scripts and are used, for example, by RIOT's continuous integration system in order to verify the quality of the code or, more directly, to make it easier for developers to flash boards.
- **doc:** contains Doxygen static documentation files, enabling RIOT documentation to be generated automatically from the source code.
- **drivers:** contains the higher-level drivers for external modules (sensors, actuators, radios). This directory is also where the API definitions for the internal CPU devices can be found (concrete implementations are located in the CPU directory).
- **examples:** contains several example applications.
- **makefiles:** contains part of the code for RIOT's build system.
- **pkg:** contains all external packages adapted to RIOT. Each external package is located in its own directory.
- **sys:** contains the system libraries code and the in-house GNRC network stack (ir `sys/net`).
- **tests:** contains the code for the unit tests and test applications. Generally speaking, each function/driver/package is associated with a test application, which means that these applications can be used as good examples (it is also recommended to look there for some usage examples).

3.2.3. The build system

Content

3.2.3a. Introduction

RIOT's build system corresponds to all files that are used to produce, from an application's source code written in C (or C++), the binary code (the firmware) that will be written on the microcontroller's flash memory. This build system also contains the rules for launching the write procedure on the flash memory, for opening a serial terminal to read the application's standard output, for launching a debugger and a whole host of other useful things.

RIOT's build system relies on the `make` tool for performing all these tasks. `make` is a widely-used tool for compiling source code in a binary that can run directly on a machine, but it can also be used for other types of tasks. All these tasks are commonly known as *targets*. In order to carry out all the operations needed for the different tasks performed by RIOT's build system (compiling, flashing, opening a terminal, etc.), `make` needs *recipes*. `make` recipes are written in `Makefile` files, which can be found pretty much everywhere in RIOT's source code. Note that you won't need to know the `Makefiles` syntax to continue in this MOOC. RIOT's build system handles all the complex tasks for you. As we will see later, only certain variables have to be set.

3.2.3b. How it works

To generate a firmware from the source code of an application, you must launch `make`, giving it as a parameter the `Makefile` located in the directory for this application. By default, `make` will search through the current directory for a `Makefile`, so the compilation can be made in one of two ways:

- by going to the application directory and selecting `make` directly:

```
$ cd <application_dir>
$ make
```

- by using the `-C` option of `make` to specify the path to the application directory:

```
$ make -C <application_dir>
```

  This method is useful if you are compiling applications located in different directorys, as it is not necessary to change the current directory each time.

It is always the application's Makefile that must be seen by `make`.

3.2.3c. Variables and targets for development

An application's `Makefile` is normally divided into 3 parts:

1. The definition of general variables
2. The addition of the modules, features, and packages necessary for the application
3. The inclusion of the rules defined by the build system

In the first section, one variable is always defined: `RIOTBASE`. This variable corresponds to the path where RIOT's source code base directory is located. This path will be used to include the generic rules (the famous `Makefile.include` file) at the end of the `Makefile`, in step 3.

Other variables, such as `BOARD` or `APPLICATION` are also defined in this first part:

- `BOARD` contains the name of the hardware target for which our application will be built. Use of the operator `?=` instead of `=` makes it possible to overwrite this variable from the command line
- `APPLICATION` contains the name of the application and is used, for example, as the name for the generated firmware files (these files have the elf, bin or hex extension)

The modules required for the application are added in the second part of the `Makefile`. The name of a module will generally correspond to the name of the file containing RIOT's code. Depending on the type of module to be loaded, 3 different variables are used:

- `USEMODULE` contains a list of the system or driver modules loaded in the application.
- `FEATURES_REQUIRED` contains a list of the CPU features required for the application. For example, for UART and SPI buses, the modules `periph_uart` and `periph_spi` are loaded.
- `USE_PKG` contains a list of packages to load in the application. The names of the possible packages correspond to the names of the sub-directorys in the `pkg` directory.

These three variables contain lists, and so the operator `+=` must be used to modify them.

Lastly, an application's `Makefile` is punctuated by the inclusion of the main `Makefile.include` file, which is located at the root of RIOT's code base directory. This file contains all logic used to manage the tools required for the selected target (in `BOARD`), the loaded modules and the packages. This is also the file used to define the targets corresponding to the different tasks to be carried out:

- `all` will compile the application code and call the appropriate compiler according to the architecture of the CPU for the selected hardware target. Calling `make` without a `make` target will result in the target `all` being called.
- `flash` will call the flashing tool corresponding to the board in order to write the produced firmware on the flash memory. Some of the more widely-used flashing tools available in RIOT include OpenOCD, JLink, Avrdude and Edbg. These depend on the board's programming interface, or a potential external programmer connected to the board (JLink, ST-Link).
- `term` calls the tool for connecting to the board's serial port. This target can be used to display on a computer the messages written on the board`s serial port.

With `make`, it is also possible to invoke several tasks into one command: `make` will ensure that they are called in the right order (provided the `Makefile` has been written properly). This can be used to compile, flash and connect to the board's serial port using just one command:

```
$ make -C <application_dir> flash term
```

3.2.3d. Other useful variables and targets

In addition to the basic targets seen before, the build system provides other targets, which can prove very useful.

- **info-build** returns useful information about the build of an application: a list of supported boards, CPU, a list of the modules loaded, etc.)
- **flash-only** is an alternative to `flash`, sparing you from having to recompile before flashing (because `flash` depends on `all` but not `flash-only`).
- **list-ttys** returns a list of all available serial ports on the host computer, plus information on the plugged-in boards (serial number). This target is useful when you're working with more than one board plugged-in at the same time on the local host).
- **debug** lets you debug an application directly on the hardware using GDB (GNU Debugger). For example, with OpenOCD, this target:

  1. Will launch a GDB server locally on the computer. This server will execute the GDB commands directly on the board via a debugging protocol.
  2. Will launch the GDB client and connect it to the GDB server. It is then possible to use the GDB debugging commands in order to control the execution of its application (load it, add stop point, check variable value, display the current position in the C code or assembler, etc.)

To obtain a full list of all available `make` targets, simply enter `make` before pressing the `<tab>` button. Autocompletion in the shell will then display this list.

Other useful variables can also be used in an application's `Makefile` or turned into command lines. A full list of these variables can be found in the file `makefiles/vars.inc.mk`.

One common variable is:

- **CFLAGS**, which is used to produce a finer compilation configuration of its application. This variable makes it possible to bypass the values of some preprocessor's macros in order to activate or deactivate the compilation of certain parts of the code. For example:

  > the option `-DLOG_LEVEL=4` changes the *logging* level in the application to *verbose*, meaning it will display more messages. Please note, however, that the size of the firmware will also increase when debug log level is enabled.

  > the option `-DDEBUG_ASSERT_VERBOSE` lets you display more debug messages in the event of a `FAILED ASSERTION` (i.e. a call to the *assert* function with a false assertion): the message will contain the name of the file and the line where the error occurred, making the problem easier to fix. Used in the Makefile, the variable *CFLAGS* must be modified using the operator `+=`:

  ```
  CFLAGS += -DDEBUG_ASSERT_VERBOSE -DLOG_LEVEL=4
  ```

- **DEVELHELP** to activate various verifications, such as enabling debug mode, assertions, displaying thread names, etc.
- **PORT** to specify a different serial port from the default serial port when multiple boards are plugged at the same time. This variable is used by the `make` target *term* in order to open the serial terminal. If the default serial port is `/dev/ttyACM0`, for example, but *List-ttys* gives

`/dev/ttyACM1` for the serial port of the visible board we are interested in, we can set `PORT=/dev/ttyACM1` on the command line or in the application's `Makefile` in order to connect the terminal to the right port.

- **IOTLAB_NODE** to transparently use the boards hosted on the IoT-LAB testbed. This variable may contain either the value `auto-ssh` for automatically choosing a board corresponding to the `BOARD` value, or the fully qualified domain name of a board on the platform, e.g `m3-42.grenoble.iot-lab.info`:

```
$ make BOARD=iotlab-m3 IOTLAB_NODE=auto-ssh -C examples/hello-world flash
term
```

The previous command will compile the application `examples/hello-world` for the board `iotlab-m3`, before flashing the firmware generated on a random iotlab-m3 board (one from the experiment in progress) and finally opening a terminal on the board's serial port. This is working the same as if the board was plugged directly into the computer.

3.3. RIOT Architecture

- 3.3.0. Introduction
- 3.3.1. The kernel and how it is used
- 3.3.2. Scheduling and Threads
- TP5. Managing threads with RIOT
- 3.3.3. Advanced use of threads
- 3.3.4. Power management

3.3. The RIOT kernel

3.3.0. Introduction

This sequence will expand your knowledge of RIOT - by the end, you will understand how RIOT's internal kernel works and how to create a multi-task application.

3.3.1. The kernel and how it is used

RIOT's kernel contains all core features of an operating system:

- the scheduler
- multi-task (or *multithreaded*) management
- synchronization between *threads*
- interrupt management

All the source code implementing the kernel's features can be found in the `core` directory. This code is independent of the hardware: the concept is to have a kernel that operates in the same way on all types of supported architectures.

What this means is that the kernel is initialized after the other hardware essential initializations have been carried out. Figure 1 outlines the full initialization sequence for a RIOT application.

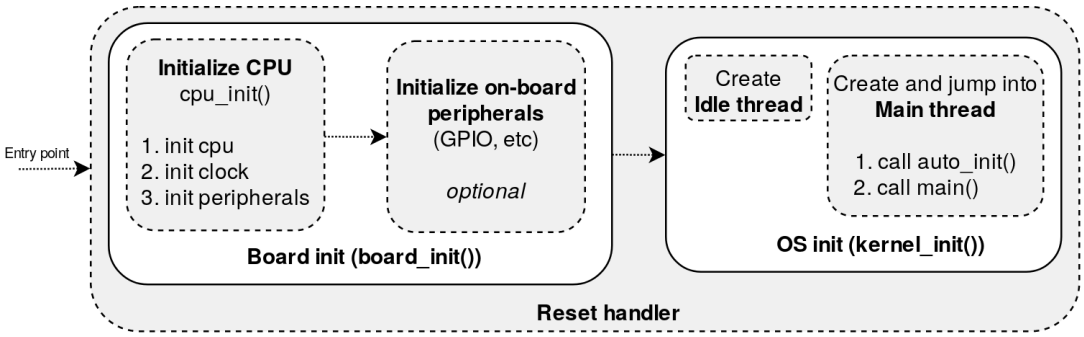


Figure 1: The initialization sequence for a RIOT, © A. Abadie

This initialization sequence takes place during the execution of the entry point for the CPU, and is therefore divided into 2 phases:

1. The hardware initialization in the `board_init()` function. This function is generally implemented, for each board, in the file `boards/<board name>/board.c`. The `board_init()` function first initializes the CPU by making a call to `cpu_init()`: this will initialize the CPU, its internal clocks and priorities between interrupts. Next, if they are required for the application, hardware peripherals (UARTs, RTCs, etc.) are also initialized.
2. Once the hardware is ready, the kernel itself can be initialized. The implementation of the kernel initialization can be found in the file `core/kernel_init.c`, and involves simply creating 2 *threads*, *idle* and *main*, and "jumping" to the *main thread*, which will be the starting point for the application, launching the application's `main()` function.

The application itself, and even the modules loaded in the application, can also start up *threads*. While the application is being executed, the kernel scheduler will switch between *threads* depending on their priority levels.

3.3.2. Scheduling and Threads

Threads are scheduled in the RIOT kernel preemptively, using a priority system. This is what is called `static` scheduling, where:

- the unblocked and non terminated *thread* with the highest priority is active, i.e. it will be executed on the CPU.
- any *thread* in the process of being executed can be interrupted by another *thread* with higher priority
- in case of inactivity, i.e. when all *threads* are either blocked or terminated, the system will automatically switch to a specific *thread*, the so-called *idle thread*
- an interrupt can preempt any *thread* at any time to execute the interrupt subroutine (*ISR*).

Scheduling uses a *tick-less* policy, i.e. active *threads* are not selected through periodic checks on the status of *threads* but through incidents (typically internal interrupts). The time taken to switch between *threads* will remain constant, irrespective of the number of *threads*, meaning the algorithmic complexity of the RIOTs scheduler is *O(1)*.

The RIOT kernel supports 16 different priority levels, with values ranging from 0 to 15. The lower the priority value, the higher the priority. The default *threads* for an application, *idle* and *main*, have the priorities 15 and 7 respectively. The *idle thread* is allocated the lowest priority possible, meaning it will only be granted access to the CPU once all other *threads* are either blocked or terminated.

Another important aspect of *threads* in RIOT is that they have their own memory stack. This is a very important property in that it ensures a degree of isolation between the different execution contexts for the *threads*: a *thread* may not directly access (and, therefore, modify) a local variable of another *thread*.

Threads in RIOT are used through an extremely simple API:

- To begin with, in the code, a *thread* is presented in the form of a function with the following signature:

```
static void *thread_handler(void *arg);
```

  The parameter `arg` may be used to share the context between the creator *thread* and the created *thread*.

- Next, a *thread* is created using the `thread_create()` function, defined in the header `thread.h`:

```
kernel_pid_t pid;
pid = thread_create(stack,  /* stack array pointer */
            sizeof(stack), /* stack size */
            THREAD_PRIORITY_MAIN - 1, /* thread priority*/
            flag, /* thread configuration flag, usually*/
            thread_handler, /* thread handler function */
            NULL, /* argument of thread_handler function*/
            "thread name");
```

In the above example, the thread is initialized with the memory space available in the `stack` array, with a priority of 6 (making it higher priority than *main*), the pointer towards the function to be executed, no context and a name. The memory stack of the *thread* is declared globally in the form of a static array:

```
static char stack[THREAD_STACKSIZE_MAIN];
```

The *thread* management API also provides the `thread_getpid()` function, which returns the process identifier of the current *thread*.

3.3.3. Advanced use of threads

Content

3.3.3a. Managing threads concurrency

As we saw in the previous section,*threads* in RIOT have their own memory stack. It can happen, however, that *threads* share variables, via the context variable `arg` of the `thread_create()` function, for example, or that they access global variables from the application or even from the system itself.

To avoid simultaneous access issues related to these use cases, the RIOT kernel provides two mechanisms to synchronize *threads*:

1. **Mutex** can be used to handle mutual exclusion problems such as access to variables being locked by the active *thread*. No other *thread* will be able to access this variable until it is released. The *mutex* API is defined in the file `mutex.h` and it is very easy to use:

```
mutex_t lock;
mutex_lock(&lock);
/* portion of code protected by the mutex */
mutex_unlock(&lock);
```

2. Semaphores can be used to manage more complex synchronization problems than is possible with *mutex*, such as "meeting" problems, for example, or "producer-consumer" problems. (*The Little Book of Semaphores*, Allen B. Downey) The API for using semaphores in RIOT is defined in the file `semaphores.h` .

3.3.3b. Communication between threads

Threads in RIOT are also able to communicate with each other within the same application. This is generally referred to as inter-process communication or IPC.

IPC messages can either be exchanged between 2*threads*, or between an interrupt context (via the interrupt subroutine, ISR) and a *thread*.

IPC can have 2 modes:

- **synchronous**: in this mode, the*thread* that sent the message will be locked until the recipient *thread* has received the message.
- **asynchronous**: in this mode, there is no blockage when messages are sent. This is possible through the use of a message queue in which the message is posted. The message will be processed once the recipient is ready. **Messages are always sent asynchronously from interrupt subroutines.** In such cases, you must make sure you initialize a message queue in the receiving *thread*.

The API for the IPC is defined in the header `msg.h` (which is found in `core`). The type for a variable containing a message is `msg_t` and has 2 attributes:

- `type`, which can be used to identify the type of message and to process it differently according to the type of message
- `content` - which is a *union* - may contain either a full 32 bits unsigned integer `unit32_t`) or the address of a variable (i.e. a pointer).

The code below shows how to define and fill in a message:

```
msg_t msg;
msg.type = MSG_TYPE;
msg.content.value = 42; /* content can be a value */
msg.content.ptr = address; /* or content can be a pointer */
```

In this case, the macro `MSG_TYPE` would be defined globally in a header file:

```
#define MSG_TYPE  (1234)
```

Messages can be sent using several functions, each with a different behaviour:

- A blocking delivery (synchronous) for a *thread* with a given *pid* number can be made using the following call:

  ```
  msg_send(&msg, pid);
  ```

  Comment: if this call is made from an interrupt subroutine, then delivery will be non-blocking (asynchronous).

- A non-blocking delivery can be made using the call:

  ```
  msg_try_send(&msg, pid);
  ```

- It is also possible to send a message and to block the sending *thread* until a response is received. In such cases, you must use the call:

  ```
  msg_send_receive(&msg, &msg_reply, pid);
  ```

  The parameter `msg_reply` will contain the reply from the *thread* to which the initial delivery was sent.

  A response can be sent from the recipient using the call:

  ```
  msg_reply(&msg, &msg_reply);
  ```

The RIOT kernel has 2 functions for receiving messages from *threads*:

- a call to `msg_receive` will block the current *thread* until a message is received:

```
msg_receive(&msg);
```

- a call to `msg_try_receive` will verify whether or not a message has been received, and, if so will return `1`.

```
int ret = msg_try_receive(&msg);
if (ret > 0) {
  /* message received */
}
else {
  /* no message received */
}
```

Receiving functions are typically used in an endless loop to wait for messages from another *thread* or from an interrupt subroutine:

```
void *thread_handler(void *arg)
{
  /* endless loop */
  while (1) {
    msg_t msg;
    msg_receive(&msg);
    printf("Message received: %s\n", (char *)msg.content.ptr);
  }
  return NULL;
}
```

When messages are being sent from an interrupt context, these messages will be sent asynchronously, meaning you must initialize a message queue at the start of the *thread*.

```
void *thread_handler(void *arg)
{
  msg_t msg_queue[8];
  msg_init_queue(msg_queue, 8);

  while (1) {
    /* Wait for messages and process them */
  }

  return NULL;
}
```

3.3.4. Power management

The power management mechanisms in RIOT rely on the capacity most microcontrollers have to switch to extremely low-power operating modes. This involves switching the microcontroller over to one of these modes automatically once all tasks are either terminated or blocked, awaiting an external event.

However, as we saw earlier, RIOT applications begin by default with 2*threads*: *main* and *idle*. Furthermore, a RIOT application will switch over to the *idle thread* once all tasks are either terminated or blocked.

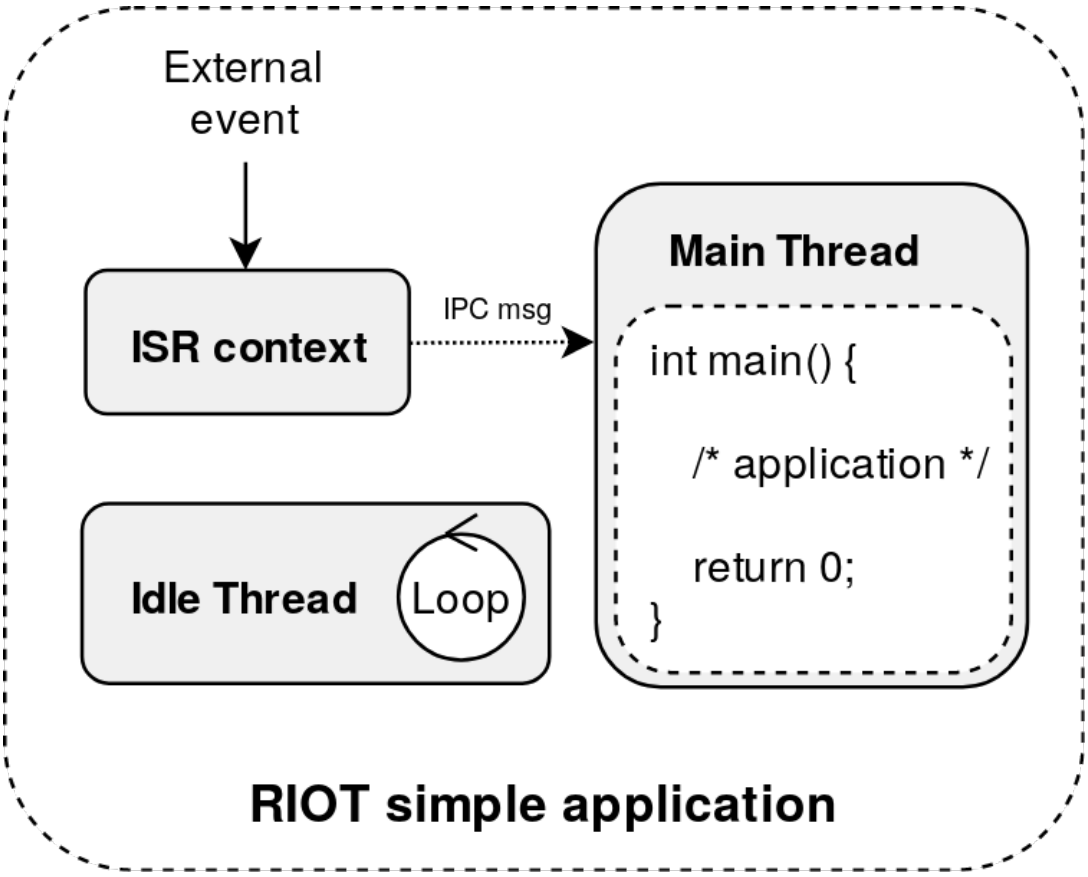


Fig. 1: Structure of a RIOT application, © A. Abadie

In RIOT, it is therefore the*idle thread* which is responsible for switching the microcontroller over to the operating mode with the lowest possible power consumption. This is referred to as putting the microcontroller to sleep.

RIOT defines a maximum of 4 low-power modes, from 3 to 0, but, depending on the architecture or group of microcontrollers, in reality, there are fewer levels.

For ST Microelectronics microcontrollers, for example, RIOT defines 2 low-power modes, while for NXP Kinetis, there is only one.

In all cases, the operating mode with the lowest power consumption will be `0`.

Switching over to the mode with the lowest possible power consumption is done using a*cascade* algorithm:

1. Beginning with the number of modes defined for a given architecture (e.g. 2 for STM32), RIOT will check to see if the highest mode is unblocked.
2. If this mode is blocked, it will remain in the current mode (idle).
3. If the mode is unblocked, it will check to see if a lower mode is unblocked, and so on.

This method ensures that the system automatically selects the mode with the lowest power consumption.

By default, none of the low-power modes will be unblocked.

Unblocking the low-power modes to use is left to the needs of the developer of the application - depending on the mode, some features will not be available (such as RAM retention, for example), or certain devices will be deactivated. The selection is made by customizing the macro `PM_BLOCKER_INITIAL` during the compilation process, using the variable `CFLAGS`. To unblock the mode 0 by default, for example, all you need to do is to add the following to an application's Makefile:

```
CFLAGS += '-DPM_BLOCKER_INITIAL={ .val_u32 = 0x01010100 }'
```

You will note that, in this example, with mode `1` always blocked, no low-power mode will be activated for an STM32. But low-power mode will work on a Kinetis, however.

It is also possible to block or unblock a mode while an application is being executed using the `pm_block` and `pm_unblock` functions. These functions are defined in the header `pm_layered.h`.

3.4. APIs hardware

3.4.0. Introduction

In this sequence you will be given an overview of the important APIs for interacting with hardware. By the end of this sequence, you will be able to write RIOT-based IoT applications applicable to a wide range of situations.

3.4.1. The hardware abstraction layer

Content

3.4.1a. A reminder of the concept

As we saw previously, the concept of hardware abstraction in RIOT involves 4 blocks (see figure 1):

- **CPUs**
- **boards**
- the APIs for hardware peripherals
- high-level **drivers**

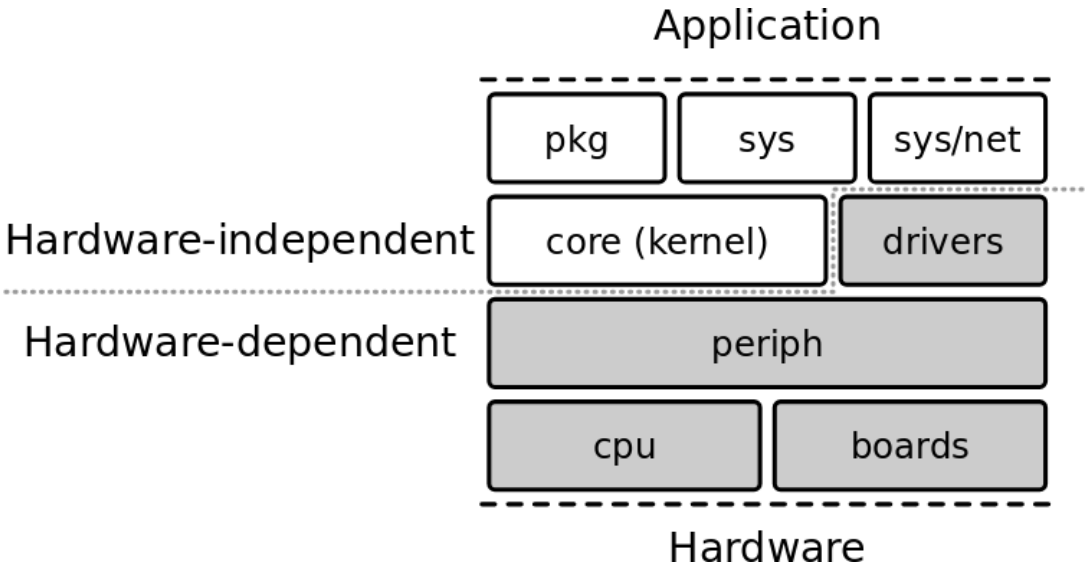


Fig. 1: General structure of RIOT

Source: E. Baccelli et al. (2018) "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT", IEEE Internet of Things Journal. PP. 1-1. 10.1109/JIOT.2018.2815038

A target device for a RIOT application is seen as a microcontroller mounted on a board, which will expose some of the pins of the microcontroller, potentially connecting up external hardware devices such as sensors, actuators or radios.

As a result, you must define what the hardware target will be during the compilation process (e.g. iotlab-m3, samr21-xpro, etc.). The build system will then select which parts of the code to compile (in the directories *boards*, *cpu*, *drivers*), as well as the toolchain to use.

Hardware abstraction makes it possible to compile a RIOT application for different hardware targets without changing the source code. The result of the build for each target can be found in a different sub-directory of the application directory:

```
<application-dir>/bin/<board-name>
```

3.4.1b. Board level abstraction

Board level abstraction can be found in the *boards* directory of the RIOT's source code.

In this directory, apart from *common*, each sub-directory corresponds to a board supported by the operating system. The directory *common* contains parts shared by a number of boards - hardware configurations, tools, etc. - thus avoiding having too much code duplication for certain very similar boards.

The directory names specific to each board correspond to the name used to inform the build system of the hardware target to use when generating a piece of firmware. Put simply, these names correspond to the value assigned to the variable `BOARD` during the call made to *make*. In terms of the build system, each directory in *board* determines its own *module* - specifying `BOARD=<board name>` will load the module corresponding to the board in its RIOT application. To find out which toolchain was used to produce the firmware (i.e. which compiler, linker, which CPU architecture), each board support must define the CPU model, group and architecture. This is done in the file `Makefile.features`. This `Makefile` file also lists the features (including the hardware drivers) available for this board. The configurations for the internal clocks and hardware are defined in the header file `periph_conf.h`. This choice is due to the fact that the majority of hardware devices, such as UARTs, I2Cs or SPIs, are dependent on how they are connected to the board (which pin the RX is connected to for UARTs, etc.) The configuration structures are specific to the architecture of the CPU used on the board, but they can be used uniformly in shared APIs: it is not the type of the structure which is used, but rather its index number in the list of configured hardware devices. We will return to this concept later in this section.

To summarize, the support for a board will define the macros for using LEDs and buttons, and potentially for specifying the different pin configurations for sensors or radio technologies. The name of these macros will be uniform across all boards, ensuring they remain portable. All these macros are defined in the header file `board.h`.

3.4.1c. CPU level abstraction

CPU level abstraction can be found in the *cpu* directory of RIOT's source code.

For each type of CPU, a hierarchical approach is used for hardware abstraction:

- at the highest level, a distinction is made between different **architectures**, e.g. ARM, AVR, MIPS, RISC-V, etc. This level is where the initialization sequences for each architecture can be found: entry point executed after a system reboot, interrupt management, etc. Generally speaking, these functions are specific to individual architectures.
- A distinction is then made between the specializations for each **group** in a given architecture. The ARM architecture, for example, is where the STM32 (STMicroelectronics), SAM (Microchip) and Kinetis (NXP) groups can be found. The way in which integrated circuits are structured varies considerably between manufacturers, meaning specific implementation/configuration is required.
- Each manufacturer also defines **types**, such as *stm32l0* and *stm32f7* for STM32 or *sam0* and

sam3 for SAM.

- Lastly, at the lowest and most specific hierarchical level, there is the microcontroller **model**, such as *stm32l072cz* or *samd21g18a*. It is this model that will be entered in the board configuration. The associated hierarchical dependencies (*type -> group -> architecture*) are then resolved by the build system.

This hierarchical approach cuts back on code duplication, maximizing reuse and simplifying long-term maintenance.

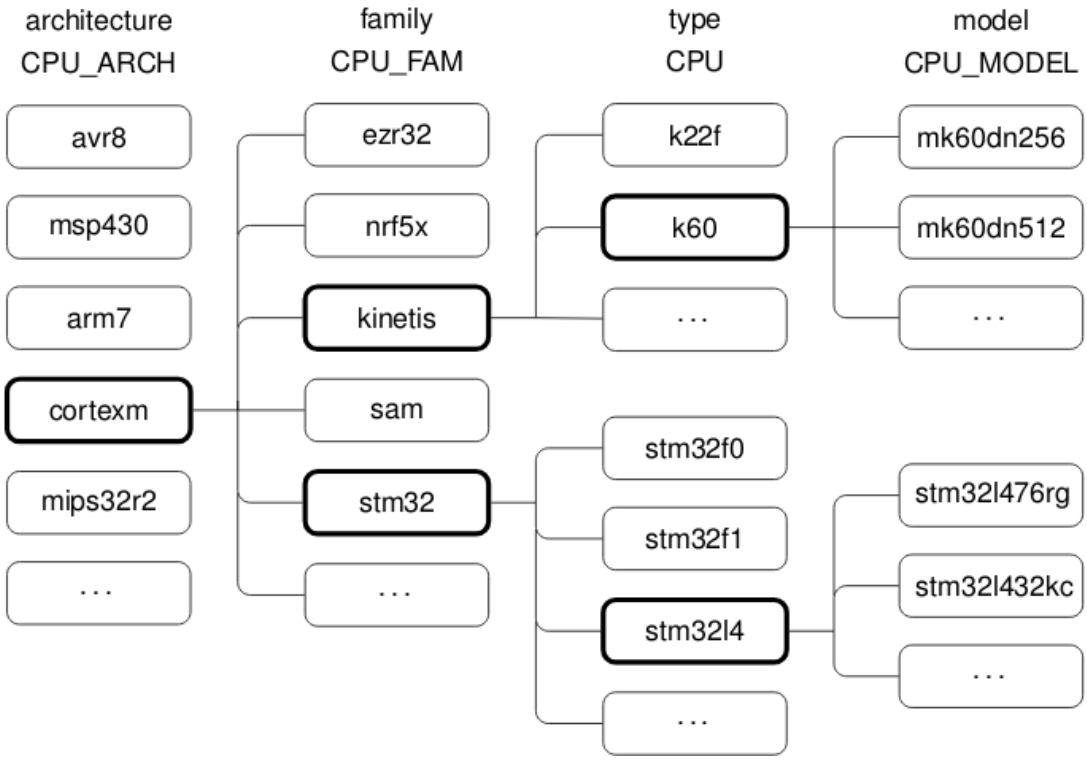


Fig. 2: CPU models organization

Source: E. Baccelli et al. (2018) "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT", IEEE Internet of Things Journal. PP. 1-1. 10.1109/JIOT.2018.2815038

3.4.1d. The APIs for CPU peripherals

The purpose of common APIs for each CPU's internal peripherals/features is to provide a shared interface above as many different architectures/groups/types as possible. What this means is that the same code can be executed regardless of the underlying hardware, thus ensuring that applications remain portable.

In order to achieve this portability, RIOT defines shared APIs in header files located in `drivers/include/periph`. Concrete implementations, dependent on each CPU, can be found in a sub-directory `periph` of each CPU. For a given hardware target, the corresponding implementation will be selected by the build system, but the functions signature will not change.

A key point to note in RIOT is that concrete implementations are written *from scratch*, using manufacturer libraries as little as possible. This requires a lot more work from the community, but it also tends to enable more efficient implementations, with less code duplication, as little

memory as possible used up and more readable code. This also ensures vendor independence: the only file needed is the header file where MCU register addresses, bit fields, interrupts and the macros specific to the different CPU models are described.

To add support for an internal CPU peripheral/feature, i.e. to compile the corresponding module in its application, developers must modify the content of the **FEATURES_REQUIRED** variable in the application's Makefile.

Here are a few modules (or *features*) provided by these APIs:

- The **periph_timer** module can be used to operate the CPU's internal counters. In order to do so, you must

  1. go to the application's Makefile and load this module in the list of required functions.

  ```
  FEATURES_REQUIRED += periph_timer
  ```

  2. Include the header `periph/timer.h` in its `.c` file:

  ```
  #include "periph/timer.h"
  ```

  You must also ensure that the board provides the support for this hardware peripheral (which is normally the case, otherwise the board would be practically unusable). This can be verified in the file `boards/<target>/Makefile.features` which must contain the line:

  ```
  FEATURES_PROVIDED += periph_timer
  ```

- The module **periph_i2c** provides the support for I2C hardware devices. In a way that is similar to `periph_timer`, the module `periph_i2c` has to be added to the list of required features (ensuring that the board provides the support), before then including the header `periph/i2c.h` in its `.c` file. The I2C support in RIOT currently only supports `master` mode, meaning it is not yet possible to implement a module behaving like an I2C `slave` (such as a sensor, for example).
- On the same design, you will have the modules **periph_uart**, **periph_spi**, **periph_pwm**, **periph_adc**, **perhip_rtc** (Real-Time Clock), **periph_rtt** (Real-Time Ticker).

The full list of all these functions is described in the RIOT online documentation at http://doc.riot-os.org/group__drivers__periph.html.

The next two sections take a closer look at how APIs are used to control timers and GPIO, but if you would like to find out more about the use of these hardware APIs or to find out how to use the other features (RTC, UART, SPI, I2C, ADC, DAC, EEPROM), the RIOT source code contains examples of applications in the *tests* directory. The names of the applications testing these APIs will start with `periph_<hardware device name>`. Even if they are only tests, these are very good use examples.

3.4.2. Timers

Uniformly managing timers on heterogeneous platforms is generally a complex problem. Indeed, the majority of CPUs provide several internal timers capable of timing in parallel, at different speeds and in different conditions (some are able to run while the CPU is asleep, while others can't).

That said, timers are among the most important features, in that they provide a time base and allow events (such as interrupts) to be generated at a specific time or at different time intervals.

In a microcontroller, a timer can't count up to infinity because it has a maximum number of cycles pre-determined in accordance with the manufacturer's specifications. This number of cycles will depend on the *length* of the timer, which may be 8, 16 or 32 bits. Once the timer reaches its maximum value (referred to as an *overflow*), it will start again from zero.

RIOT's API for managing timer devices makes it possible to transparently use timers on a wide variety of architectures. The following tasks can be performed:

- starting and stopping timers using the functions*timer_start()*/*timer_stop()*,
- reading the current value using*timer_read()*,
- configuring a *callback* after a specific delay using the functions*timer_set()* or *timer_set_absolute()*.

This API is quite useful, but it has its limitations, and is not capable of really getting the most out of the multi-task functions of an operating system.

The system module*xtimer* was developed to address these limitations. The concept behind this module is to provide a multiplexing mechanism on top of a hardware timer: one single API that can be used to manage multiple delays from one hardware timer. The *xtimer* module enables delays that are precise to the microsecond. Given that the *xtimer* module is implemented above the *periph_timer* API, it is fully portable.

The module's API is extremely easy to use. Here are a few examples of the most important uses of this module's functions:

- retrieving the current system time in microseconds:

```
xtimer_ticks32_t now = xtimer_now();
```

- blocking the execution of a*thread* for a delay of*sec* seconds:

```
xtimer_sleep(sec);
```

  During this time, the system can switch to another*thread* in order to carry out other tasks.

- blocking the execution of a*thread* for a delay of*microsec* microseconds:

```
xtimer_usleep(microsec);
```

- calling a *callback* function after a delay of *offset* microseconds.

  > Using an `xtimer_t` type variable to execute a *callback* function at a certain time:

```
void cb(void)
{
  /* code executed in callback */
}
[...]
xtimer_t timer;
timer.callback = cb;

xtimer_set(&timer, offset);
```

3.4.3. Interacting with GPIOs

The API for the **periph_gpio** module provides a unified interface for controlling a microcontroller's input/output pins. The API's functions can be used after having included the **periph/gpio.h** header.

GPIOs are generally grouped together by *port* on a microcontroller, with the API using the macro `GPIO_PIN(<port>, <pin>)` in order to obtain the memory address for the device in the microcontroller. As a result, the value returned by this macro will depend on the architecture of the microcontroller.

In order to start using a GPIO pin, you must first initialize it with the right mode by calling the function `gpio_init()`. Usual modes (*INPUT*, *INPUT* with pull-down, *INPUT* with pull-up, *OUTPUT*, etc.) for using GPIOS are available, provided they are supported by the underlying hardware target.

```
gpio_init(GPIO_PIN(0, 5), GPIO_OUT);
```

The functions `gpio_set` and `gpio_clear` can be used to switch the GPIO from high state to low state respectively:

```
gpio_clear(GPIO_PIN(0, 5));
gpio_set(GPIO_PIN(0, 5));
```

The management of GPIO interrupts is deactivated by default, but can be added using the module **periph_gpio_irq**. The purpose of this feature is to optimize the size of the code in cases where the use of GPIO interrupts is not necessary. GPIOs with interrupts are initialized using the function `gpio_init_int()`.

Here is a simple example:

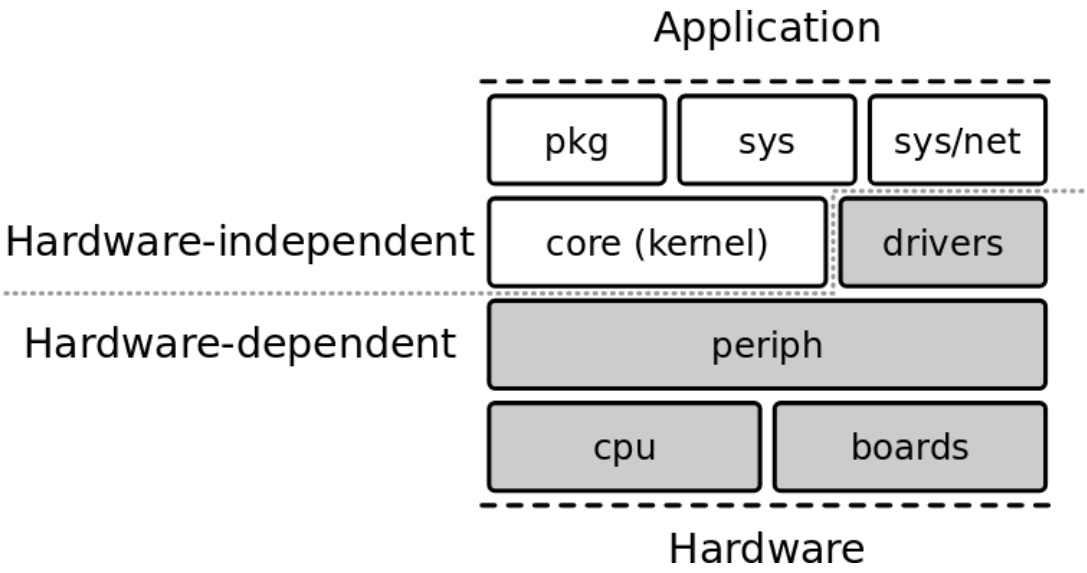
```
static void gpio_cb(void *arg)
{
   (void) arg;
   /* manage interrupt */
}

int main()
{
   gpio_init_int(GPIO_PIN(PA, 0), GPIO_IN, GPIO_RISING, gpio_cb, NULL);
}
```

3.4.4. High-level drivers

Content

3.4.4a Concept



General structure of RIOT

Source: E. Baccelli et al. (2018) "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT", IEEE Internet of Things Journal. PP. 1-1. 10.1109/JIOT.2018.2815038

Device drivers are used to operate a wide variety of sensors, actuators and radios connected to microcontrollers on GPIOs or on data buses such as UARTs, I2Cs or SPIs.

In order to ensure they are compatible with all architectures supported by RIOT, these high-level drivers rely on the standardized APIs we came across earlier for interacting with microcontrollers.

The fact that all these drivers are implemented directly in RIOT's source code makes developing the application easier and ensures they will be maintained long-term by the community.

The way in which these device drivers are implemented by RIOT makes it possible to use more than one device of the same type at the same time in the same application: it is possible, for example, to have multiple sensors on the same I2C bus, owing to the fact that they use different addresses. The driver will know how to handle this, because in its memory it uses a descriptor containing the status of each device. This descriptor is kept in memory during the whole application life-time which is kept for the duration of the application's life time. This characteristic also provides a better way of dealing with concurrency issues (multiple *threads* trying to access the same device), while making it easy to use these devices in separate *threads*/contexts (remember that each *thread* has its own memory *stack*).

All these high-level drivers are implemented in the `drivers` directory in RIOT, with each driver directory name corresponding to a module that can be imported into an application's `Makefile`

in order to compile the corresponding driver.

All that has to be done is to include the header file defining the interface `#include <driver name>.h`).

As is the case for internal CPU peripherals, there is a test application in the *tests* directory for each driver. These applications are all called `driver_<driver name>` and, once again, even if they are for test purposes, provide very useful examples of how these drivers are used.

3.4.4b Initialization

In order to use high-level drivers correctly, an important aspect to consider is their initialization procedure. Each driver in RIOT defines an initialization function (e.g. `<driver name>_init()`) taking two parameters:

- the first parameter is the pointer towards the driver descriptor, which will contain the status of the device while the application is being executed. The type is usually called `<driver name>_t`.
- the second parameter is the pointer towards the structure containing the driver's initialization parameters for this device. Each driver implementation will supply an initialization configuration with default parameters (normally adapted for the most widely-used boards). These parameters are defined in the header file `drivers/<driver name>/include/<driver name>_params.h`. Given that the default parameters use macros, these can easily be overwritten, either from the application code, or from a support board. In this second case, macros will be defined in the file `board.h` of the support board. This mechanism, which mixes together header files and macros, is used to specialize the driver's initial configuration to suit the needs of its application and/or the configuration of the board.

A typical initialization sequence for a driver will be performed in the following example:

```
#include "driver_name.h"
#include "driver_name_params.h"

static driver_name_t dev;

int main()
{
   [...]
   driver_name_init(&dev, &driver_name_params[0]);
   [...]
}
```

3.5. Peer evaluation of your IoT application

- **3.5.0 Introduction**
- **3.5.1. Presentation of "Peer Evaluation"** *(not included)*
- **3.5.2. Scoring scale** *(not included)*
- **TP7. Use sensors on the IoT-LAB M3 board**
- **3.5.3. RIOT application on FIT IoT-LAB**
- **3.5.4. Submit your activity and evaluate your peers** *(not included)*

3.5.0. Introduction

In this sequence you will write a RIOT-based IoT application that can be applied in a wide range of situations.
This practice will take the form of a **peer-reviewed activity**. This activity counts for 5% towards the grade for successful completion of this Mooc.
Key dates:
(1) from April 9th to April 26th: Submission of your work
(2) from April 27th to May 10th: Correction and grading of 3 peer-reviewed papers

3.5.3. RIOT application on FIT IoT-LAB

This video shows the RIOT application developed during the programming exercise in TP7. The sensor values are read from an M3 board in the FIT IoT-LAB testbed.

NB: this video gives an overview of what you will get out of the IoT application. It was shot long before the Jupyterlab environment was available and integrated into the course, which is why the manipulations are done on a computer and not via Jupyterlab. The Makefile and main.c files are already included in the riot/basics/sensors folder of the course. You can modify them by following the notebook sensors.ipynb.

Module 4. Focus on Low-Power Communication Networks

Objectif : À la fin de ce module, vous serez capable de décrire les protocoles de communication IoT avec les différentes couches réseaux. Vous serez également en mesure d'écrire votre première application IoT avec l'utilisation du protocole Internet CoAP pour récupérer les valeurs d'un capteur de température.

Activités pratiques (TP) : TP8: Discover 802.15.4 TP9: Discover IPv6 and 6LoWPAN TP10: Discover UDP socket TP11: Discover CoAP protocol TP12: Discover LwM2M protocol TP13: RPL

Contents of Module 4

4.1. Low-Power Wireless Networks

- 4.1.0. Introduction
- 4.1.1. Topologies, constraints and objectives
- 4.1.2. The OSI / IETF model

4.2. The 802.15.4 Communication Protocol

- 4.2.0. Introduction
- 4.2.1. Air as a communication medium
- 4.2.2. A shared medium
- 4.2.3. Saving energy through synchronization
- 4.2.4. Deterministic networks and guarantees
- TP8. Discover 802.15.4

4.3. 6LoWPAN : IPv6 for the IoT

- 4.3.0. Introduction
- 4.3.1. Discovering IPv6 and 6LoWPAN
- TP9. Discover IPv6 and 6LoWPAN
- TP10. Discover UDP socket

4.4. CoAP: le protocole d'application Internet

- 4.4.0. Introduction
- 4.4.1. The architecture of the CoAP protocol
- 4.4.2. The format of CoAP messages
- 4.4.3. Examples of interactions
- 4.4.4. Resource discovery
- 4.4.5. Resource observation
- 4.4.6. CoAP installation on FIT IoT-LAB
- TP11. Discover CoAP protocol
- TP12. Discover LwM2M protocol

4.5. Routing in Low-Power Wireless Networks

- 4.5.0. Introduction

- 4.5.1. The RPL routing graph
- 4.5.2. Creating and maintaining topologies
- 4.5.3. Managing the impact on memory
- TP13. RPL

4.1. Low-Power Wireless Networks

- 4.1.0. Introduction
- 4.1.1. Topologies, constraints and objectives
- 4.1.2. The OSI / IETF model

4.1.0. Introduction

In this sequence, we will introduce Wireless Personal Area Networks (WPAN), with a particular focus on low-rate WPANs or LRWPANs.

For these LRWPANs in the context of the Internet of Things, we will be studying:

- their topologies, i.e. the possible interconnections between communicating objects
- their communication constraints (energy limitations, loss of connection) and expected characteristics (range, rate/speed, latency, reliability, etc.)
- the OSI (Open System Interconnection) model, which is the reference model for interconnecting systems open to communicating with other systems, and which can be used to write an entire communication system

By the end of this sequence, you will know how to specify a wireless network architecture and its communication protocols: which radio communication channel to use, how to share this communication medium between different wireless objects and which routing and data transport protocols to use.

A video presentation on low energy-consumption WPANs: their architecture, existing communication protocols and associated constraints.

4.1.1. Topologies, constraints and objectives

Content

4.1.1a Topologies

Wireless networks, whether those used initially for mobile communications or those used more recently with technology such as Bluetooth or Wi-Fi, all share a common point: they rely on a centralised infrastructure. Objects communicate with each other via access points, which are responsible for organising pairing, resource allocation and a range of other essential functions (e.g. energy efficiency, security).

These are known as star topologies, which are widely used in standards dealing with wireless local area networks (WLANs). It's the default case for Wi-Fi, (the IEEE 802.11 standard), with an access point and objects linked to it:

Fig. : Star topology

Other types of network topologies exist, one of which is mesh topology. In this case, there are links between each pair of nodes:

Fig. : Full mesh topology

or partially, i.e. only certain links actually exist and can be used:

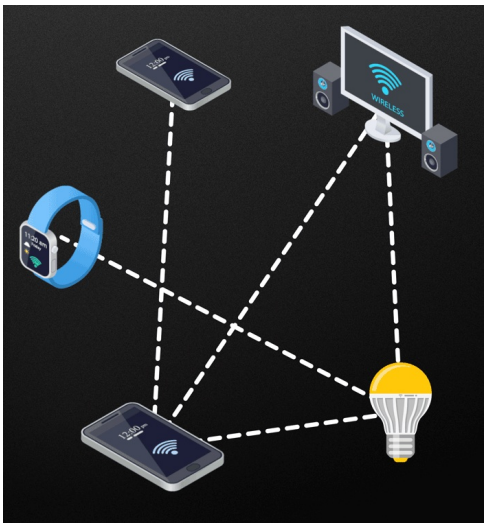


Fig. : Partial mesh topology

However, there is practically no transposition for token ring or bus topologies, which are used most notably in wired networks, as is the case with Ethernet, for example.

As we will see further on, mesh topologies are the best suited for LRWPAN networks, but they are also the most complex to set up, if only because of the routing paths that have to be calculated in order for data to be transmitted between objects.

4.1.1b Constraints

In the context of the Internet of Things, communicating devices are constrained. They are battery-operated and have limited resources (memory, processing). What this means is that the devices in question are roughly 150 times less powerful in terms of their processing power than next-gen smartphones, in addition to having 1000 times less memory (e.g. a 16 MHz clock with 2 K of RAM and 32 K of flash memory on an Arduino Uno microcontroller).

However, they must be capable of carrying out a range of potentially costly and complex operations in order to communicate (e.g. accessing shared resources, identifying routing paths for data, ensuring exchanges are reliable and secure, etc.). For transmissions, for example, it is necessary to have the listening time of the radio channel in order to detect incoming messages or to identify the resource as being available in order for it to transmit itself. This radio listening activity consumes a significant amount of an object's energy budget.

Generally speaking, wireless transmission and reception are very costly operations. The further an object is looking to transmit, the more it must increase its transmitting power and, as a result, its energy consumption. This is just one reason (antenna quality is another) why IoT objects are considered as using low ranges. This makes it difficult or even impossible to use star topologies, as is the case with other wireless networks, and so mesh topologies are used, where each object is only able to communicate with a sub-section of the network. This is known as multi-hop routing, which is used for gradually transmitting data from a source to a destination. Lastly, it should be noted that the radio links considered in these networks have a high loss rate, meaning they are not stable long-term and are not necessarily symmetrical. These constraints must be taken into account when defining the communication protocols used between objects.

What this means is that this mesh (and unstable) topology must be built and maintained, requiring data to be exchanged between objects. As we will see, a hierarchy can be established based on objects' capacities (e.g. objects are referred to as having total or reduced functions in the IEEE 802.15.4 standard). This multi-hop topology also requires data to be relayed from its source to its destination. In such cases, the routing paths must be calculated, evaluated and updated on a regular basis. This leads to costly processing, in much the same way as the cryptographic operations which have to be carried out during secure exchanges. These occupy part of the limited resources of each communicating object (e.g. time, processing power, energy).

4.1.1c Objectives

Protocol stacks used in LRWPANs have a number of objectives.

As mentioned earlier, objects must have the necessary communication resources for exchanging with other parts of the network. Most importantly, a decision has to be made as to the type of radio technology used and how it is to be configured (e.g. frequency, modulation, coding), in addition to defining the expected characteristics of the radio links (e.g. loss rate, latency, speed, etc.).

Among other factors, these define the communication ranges and, as a consequence, the topology formed by the different objects. We will see how to establish this topology and to maintain it (e.g. linking together objects, inlets/outlets for network elements).

Once the topology has been established, objects can then use it to exchange data. We will see how to identify each of these using IP addresses (e.g. the IPv6 protocol), which are also essential to integrating these networks into the internet.

Before that, addressing makes it possible to send data to destinations and to identify the source of received data. In order for exchanges to take place on multi-hop topologies, the paths the data will follow must be established. This is the purpose of routing solutions, which must guarantee the delivery of data and resilience in dynamic environments (e.g. link breakdowns, node errors, mobility).

Lastly, we will see how to transport data from applications and to ensure exchanges are reliable,

before providing some basics regarding communication security.

We will then see how all these preoccupations can be separated using layered architectures. The other sequences within this module will outline each aspect of the communication stack cited above.

4.1.2. The OSI / IETF model

Content

We are going to take a look at a network architecture based on layer models and how the specificities of the IoT are taken into account.

4.1.2a Network architecture structure

In order to implement an IoT network, there are certain requirements in terms of connectivity, data delivery, information transport and application security and reliability that must be met. The variety of hardware, the diversity of radio environments, the complexity of multi-hop topologies and the needs of the vast array of possible applications on these networks have to be taken into account. Such preoccupations are nothing new - indeed, they were addressed while the internet as we know it today was being created and as it evolved.

Let's take a look back at the traditional approaches used to solve the range of problems posed. We will outline how these models are employed in the context of IoT networks.

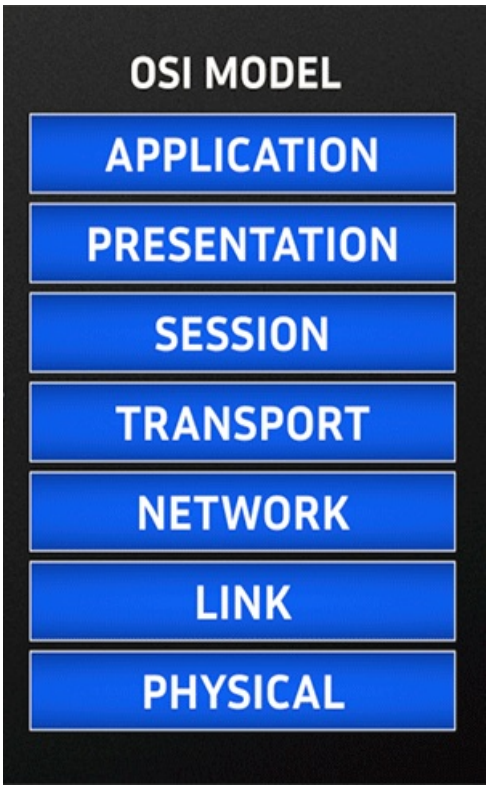
A WPAN is comprised of a set of appliances communicating wirelessly around an individual person's workspace. We will focus here on the structure of the communication protocols found in connected objects forming WPANs.

Reliable wireless communication and appropriate data collection are needed in order to supply end applications. To that end, the following characteristics must be carefully examined: target appliances produce relatively low volumes of data periodically. They use low-capacity batteries and limited ranges for wireless communication, given the main constraints regarding energy consumption and design.

There are a range of different communication protocols, which enable objects to exchange, structure and validate data. Generally speaking, communication between such systems requires various different mechanisms being set up in order to enable information (e.g. web pages) to be converted into the electrical signals used for transmission (over cables or via radio links). These include information coding; shared environment management; synchronizing, detecting and correcting errors; identifying objects (addressing); establishing links and communication routes; transporting information; security, etc.

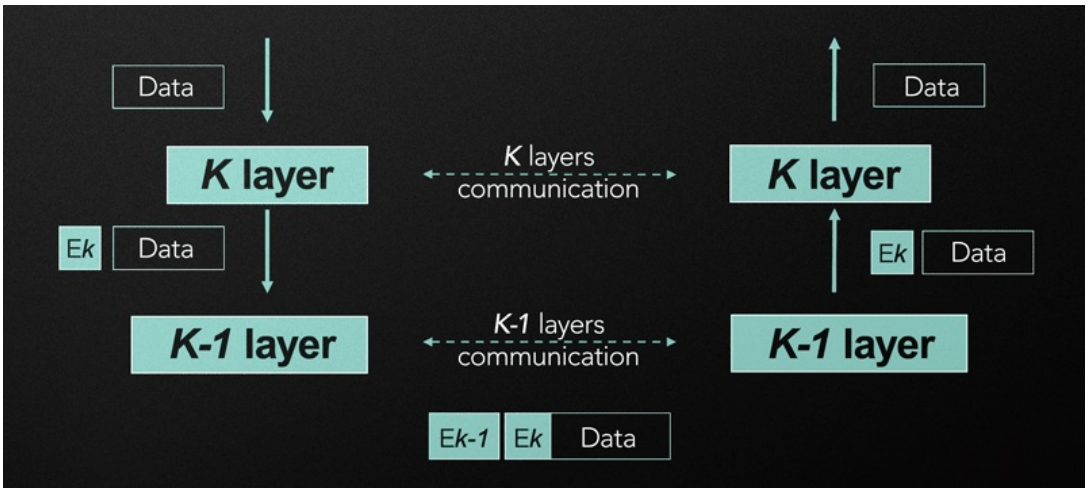
4.1.2b The OSI model

The issue of network organisation prompted the standardisation of a communication model based on so-called layered architectures. The OSI (Open System Interconnection) model was the first to have been proposed, by the ISO (the International Standardisation Organisation). It constitutes the basic reference model for interconnecting systems open to communication with other systems. It describes the architecture of network communications, the goal being to identify the main functions linked to communication and to sort these into layers:



OSI model.

The OSI model also provides abstraction principles through concepts such as layers, interfaces, services and protocols. A layer thus presents a set of system calls and functions in a program (interface). It defines the functions it is responsible for (services) using primitives (commands or events). It can also be used to define the communication protocol relating to this layer (the format, the signification for packets/messages exchanged, etc.). This protocol is a set of rules that can be used to establish and maintain communication at this level (or layer). A data layer K will use the service provided by the layer K-1, as described by the latter's interface. In return, the data used by a layer K will only have any meaning (i.e. becoming information) for the upper layer K+1. This data encapsulation is accompanied by the addition of a header specific to each layer. This mechanism is illustrated below:



Encapsulation according to the OSI model.

In this way, in a full architecture for communication between open systems, the mechanisms to be implemented range from physical-level protocols (synchronization, coding data into signals and vice versa, handling data bits) to application protocols (e.g. HTTP), in addition to routing protocols (where the packet is the unit of information used) or session protocols (e.g. transmission security).

In the standard systems we use on a daily basis, a distinction is made between, for example:

- communication devices and their drivers (roughly corresponding to the physical and link layers)
- the operating system (in which you will find the mechanisms specific to the transport and network layers)
- the applications where the application, presentation and sessions level functions can be found

The OSI model is a layered structure designed to connect open systems together. This is an ideal, theoretical stack, where each layer is responsible for a set of preoccupations. A data layer enables the one above it to use its service and to implement its own service. The role of the data link layer, for example, is to establish the radio vicinity the routing layer will use to construct the routing topology. The OSI model can be used to write an entire communication system. However, existing systems (and the protocols they use) won't necessarily contain an equivalent for each layer of the OSI model. It should be noted that this does not define any protocol.

4.1.2c The IETF IoT protocol stack

The majority of communication protocols have been standardised by the IEEE (the Institute of Electrical and Electronics Engineers) and the IEFT (the Internet Engineering Task Force). While the first covers the physical layer and a sub-section of the link layer, the second is responsible for everything found "above the wire (of the communication medium, strictly speaking - Ed.) and below the application".

In the context of the IoT, communication takes place via wireless links. These employ various types of technology (e.g. Wi-Fi, Bluetooth, ZigBee, LoRa, Sigfox). The radio frequencies used are either separate or shared, meaning the radio environment has to be well managed as a shared resource. Different transmission schemes are used, enabling wireless communication across a range of propagation environments. What this means is that a large number of radio chips are available for building connected objects. We will focus here on the physical layer defined by study group 4, group 15, committee 802 (i.e. the IEEE 802.15.4 standard), which is used by the majority of available IoT objects.

The protocols embedded into a system will depend on its constraints on it and the use to which it is put. For example, in order to ensure that IoT deployments are able to operate long-term, appliances must use as little energy as possible. A significant amount of research and engineering in the context of the IoT has focused on energy efficiency. Given that radio chips use up the most energy, their use must be limited. Appliances alternate between inactive and active modes. They only communicate where necessary, while limiting the amount of time they listen to the medium for ongoing transmissions from other nodes. As a result, thorough medium access control is needed (also known as MAC). Excessive activity results in huge amounts of radio noise, interference, etc., while periods of inappropriate activity can lead to the topology becoming disconnected and the nodes becoming isolated. Standards such as IEEE 802.15.4 provide solutions for this type of coordination, enabling each appliance to discover which objects in its environment it will be able to communicate with. Together, these components form a whole that is equivalent to the physical and link layers found in the OSI model.

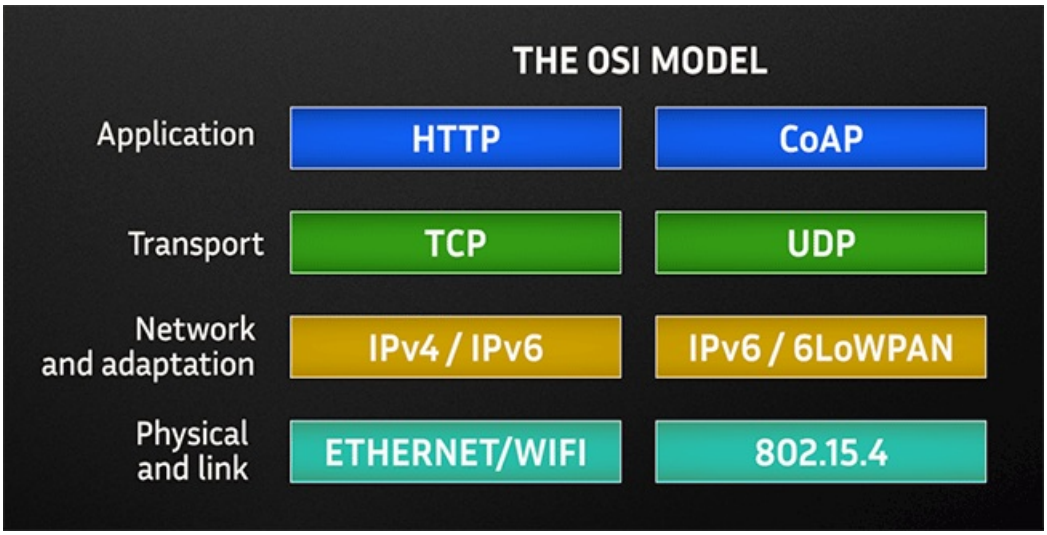
Based on this logical topology, hardware devices must find a way of sharing their data with receiver stations. Given the limited radio range, this is done through paths made up of wireless links. The right decisions have to be taken in order to select the neighbouring node that will be the next hop on this path. This object will be tasked with relaying the data packet to its own next hop towards its destination. Once again, a whole host of research and engineering projects have focused on energy-efficient routing in IoT networks. Recent standards such as RPL can be used to build routing topologies in difficult radio environments. These establish the functions of the routing layer.

Once it is possible to transport the data from one point to another in the network, the issue of the reliability of end-to-end communication must be tackled. The goal is to prevent the loss of packets or the desynchronization of data packets in order to ensure relevant and efficient data collection. In the internet that we use everyday, transport protocols such as TCP are used to apply these properties. In IoT networks, the constraints of embedded equipment require adapted solutions, a common example of which is the use of the UDP transport protocol. In reality, these don't provide the upper layer protocol with any guarantees that a message will be delivered.

On the whole, the desired reliability will depend on the needs and the requirements of the application, which can vary considerably. For remote healthcare or home monitoring applications, for example, the data that is collected has to be highly reliable. With devices such as smart lights or watches, meanwhile, less attention is required.

We have listed the main preoccupations to factor in when assessing the architecture and protocols for low energy-consumption WPANs: the radio medium and the method used to share it between the wireless appliances; routing; transport; and the final application. These problems are separate, and so are dealt with separately.

The IETF (the Internet Engineering Task Force) is made up of working groups tasked with identifying specific solutions for each layer. In the context of the IoT, for example, the proposed structure supposes the use of IEEE 802.15.4 for the physical layers and MAC. The IEFT also provides higher level protocols for routing, transport and application. The global stack is typical of what you would find in an open connected object.



IETF protocol stack

4.2. The 802.15.4 Communication Protocol

4.2.0. IEEE 802.15.4: a communication standard for the IoT

This sequence will introduce you to the IEEE 802.15.4 standard, one of the communication standards for objects with constraints in terms of processing power or energy. We will see how air is used as a communication medium and how it is shared, in addition to how energy consumption can be optimised and how guarantees in terms of access and robustness can be delivered.

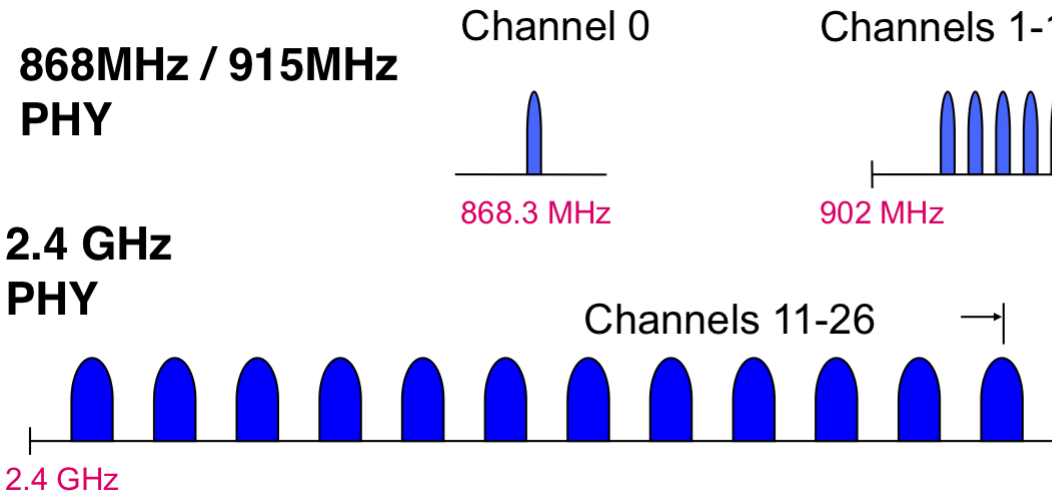
A video presentation on the main concepts behind the 802.15.4 communication standard: modulation of the radio wave, managing collisions and coordinating communication periods in order to ensure the robustness and the reliability of radio transmissions while limiting energy consumption.

4.2.1. Air as a communication medium

There are now many different types of wireless communication technologies, some of which rely on radio communications. A radio communication system modifies a radio carrier wave (e.g. frequency, amplitude, phase), whose job it is to transport data from the emitting device. Parallels can be drawn here with human communication, which is made possible by the vocal cords vibrating air in order for sound to propagate. For radio transmissions, we refer to modulation of the carrier wave, which propagates by means of an antenna generating an electromagnetic field. The frequencies and the characteristics of the bandwidth used, meanwhile, determine which speeds can be reached.

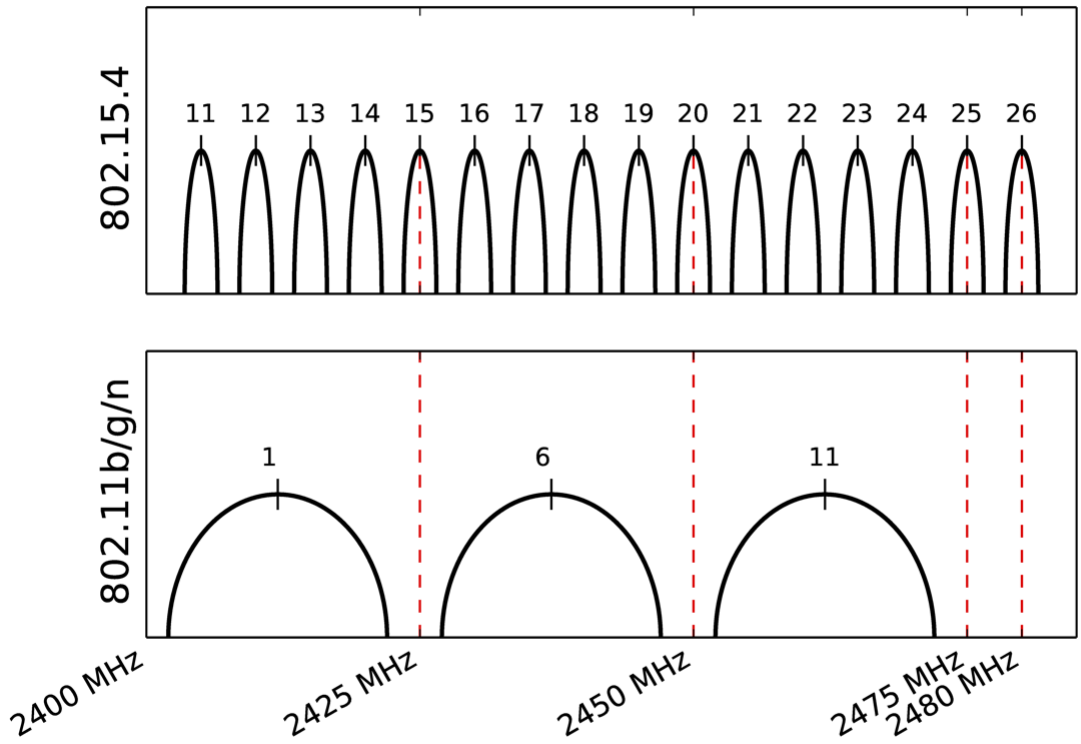
The 802.15.4 standard defines the use of:

- 16 channels in the 2.4 GHz ISM frequency band
- 10 channels in the 915 MHz ISM frequency band in the USA and in Australia
- 1 channel in the 868 MHz frequency band in Europe



Communication operations are costly, and use up a significant chunk of the energy budget for connected objects.

In the context of the IoT, the standards proposed must define not only the communication medium (frequency, modulation, etc.) but also the way in which it will be shared as a resource, either by multiple objects or with any other types of radio technology that may be present. For 802.15.4, the 2.4 GHz frequency band is in competition with the channels allocated to Wi-Fi, as shown by the figure below.



These preoccupations are central to the physical and link layers found in the OSI model. More specifically, the way in which objects share the communication resource is defined by the medium access control (MAC) sublayer.

4.2.2. A shared medium

Air is a common medium, and so we must share it.

When friends meet up for dinner, in order for the group to have a conversation, a relatively simple principle has to be followed: only one person may speak at any one time. In much the same way, in order to enable a set of objects to exchange messages in a wireless environment, the 802.15.4 standard proposes following this same principle, using the CSMA/CA protocol (CA stands for *Collision Avoidance*).

In this type of situation, we have a coordinator waiting for data from multiple devices with data to transmit. The device wishing to transmit must first check to see if the medium is free. If it is not, the device will come up with a random delay, after which it will try once again to transmit. If the medium is free, the device will transmit its data.

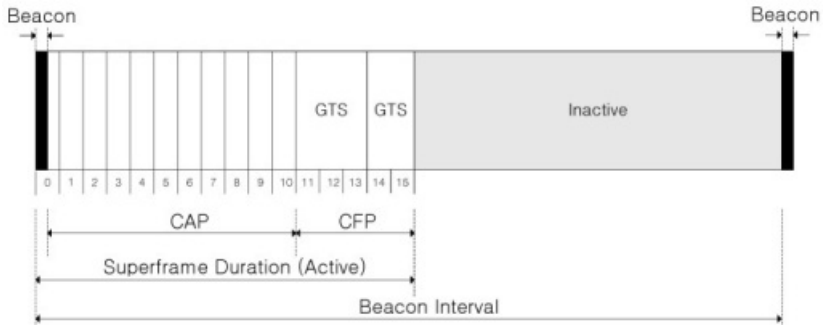
In this type of scenario, a network arranged in a star-shape around a coordinator will issue a request to be given a coordinator plugged into the mains in order to enable it to continuously listen out for messages. The devices themselves will be capable of running off batteries, operating in standby mode for a significant chunk of the time and waking up to share their message.

There is, however, one major disadvantage to this method: access to the medium is not guaranteed within a given time period. This will depend primarily on the density of the network and the number of devices looking to emit.

4.2.3. Saving energy through synchronization

Having to listen and analyse before being able to speak uses up a lot of energy. The constrained nature of objects meant that a solution had to be found in order to save energy.

With this in mind, a mode including the sending of *beacons* was added to the CSMA/CA protocol. The purpose of this mode is to synchronise devices with the coordinator, enabling all devices to become independent, including the coordinator itself. The coordinator organises access to the channel and data transfer using a structure known as a *Superframe*, which repeats itself while remaining the same size and which is comprised of an active period and an inactive period (optional).



The Active period is subdivided into 16 *Slots*. The first is dedicated to the transmission of the *Beacon*. The remaining slots form 2 periods, which differ according to their medium access method:

1. A first with contention, the *Contention Access Period* (CAP)
2. A second without any contention, the *Contention Free Period* (CFP).

The *Beacon* will contain specifications regarding the current *Superframe*, including a description for the different periods cited above and their duration in terms of the number of *Slots*.

During the CAP, the network will operate in standard CSMA/CA mode, where each device waits until the medium is free before transmitting and where each transmission is aligned to a *Slot* start. Transmission is limited by the remaining size of the CAP. The emitting device must wait for the next *Superframe* if it is unable to emit during the CAP. The same shall apply if the number of *Slots* needed for the transmission is higher than the remaining number of *Slots*.

During the CFP, the network will operate in TDMA mode (*Time Division Multiple Access*). This medium access control technique divides up the time on the bandwidth and distributes the available time among the devices wishing to emit. The medium will be allocated in turns to the different emitting devices. In the *Beacon*, the coordinator will specify how the CFP is to be broken down into guaranteed time slots for an emitting device.

Lastly, a **period of radio inactivity** may be defined. As a minimum, this will make it possible to switch radio components to standby mode, giving the upper layers the time to process data. In cases involving applications collecting low frequency data, the device can be put into deep sleep mode, thus saving energy between readings.

What happens with a *Superframe* can be broken down as follows:

1. The coordinator sends a *Beacon*

2. Devices looking to emit compete with each other during the CAP, adopting the CSMA/CA mode. They may emit data, in addition to requesting to be allocated a GTS in an upcoming *Superframe*.
3. The devices allocated a GTS by the coordinator, which will be communicated in the *Beacon*, will have a free channel for emitting or receiving data.
4. The devices observe a period of inactivity.

In terms of energy economy, the coordinator will be active during the *Superframe's* Active period. The other devices will only be active during the CAP if they have something to transmit, and/or during a GTS if they have been assigned an interval.

4.2.4. Deterministic networks and guarantees

Content

4.2.4a Guarantee constraints

The medium access techniques discussed so far enable sharing. A range of scenarios in the modern world (e.g. the factory of the future, smart buildings, assisted driving, etc.) involve decisions being made based on data from connected objects. In these new, critical applications, the underlying networks must provide guarantees in terms of access to the communication resource and data delivery in environments that are often difficult for radio communication (e.g. physical obstacles, external interference linked to the presence of other wireless technologies in the environment).

Medium access control centred around CSMA/CA does not enable these constraints to be met. It defers emissions when the channel is not free, but cannot guarantee that it will be free at a later stage.

4.2.4b Time division and frequency hops

As we saw earlier, the IEEE 802.15.4 standard enables objects to ask for and obtain guaranteed time slots (GTS), during which only they will be able to communicate. This *time-division multiple access* (or TDMA) was covered in an amendment presented in 2012 (IEEE 802.15.4e), aimed at providing better support for "industrial markets", by dedicating these *time slots* (TS) to certain communications. In this proposal, the entire superframe was thus controlled in accordance with TDMA. The amendment IEEE 802.15.4e also proposed performing *channel hopping* (CH) in order to obtain a greater degree of robustness when faced with external interference or obstacles.

In 2015, the TSCH mode was integrated into the standard and works as follows.

4.2.4c The TSCH mode in the IEEE 802.15.4 standard

The TSCH mode in the IEEE 802.15.4 standard is derived from the beacon mode presented earlier. Between two coordinator beacons, we now use the term *slotframe*, which represents the time division (*absolute slot number*, or ASN) and the channel hop (*channel offset*), as represented below.

```
Chan.   +----------+----------+          +----------+
Off.0   | TxRxS/EB |   OFF    |          |   OFF    |
Chan.   +----------+----------+          +----------+
Off.1   |   OFF    |   OFF    |   ...    |   OFF    |
        +----------+----------+          +----------+
             .
             .
             .
Chan.   +----------+----------+          +----------+
Off.15  |   OFF    |   OFF    |          |   OFF    |
        +----------+----------+          +----------+

slotOffset     0          1                   100

EB:  Enhanced Beacon
Tx:  Transmit
Rx:  Receive
S:   Shared
OFF: Unscheduled by this specification
```

Fig. : A slotframe containing 101 slots in TSCH mode in the IEEE 802.15.4 standard (RFC 8180, IETF).

A frequency/time block must then be allocated to each communication link. There are different strategies for doing this, one of which is to use routing topology. This is the case for the *minimal scheduling function* (MSF), which was adopted by the IETF and which enables objects to request blocks with their next hop in the routing topology.

The figure below illustrates a simple topology comprised of an object (object A) to which two nodes (B and C) must send data. The *slotframe* can be built by taking this transmission scheme into account; B and C have thus obtained cells, meaning they now have a specific time and a specific frequency for communicating with A.

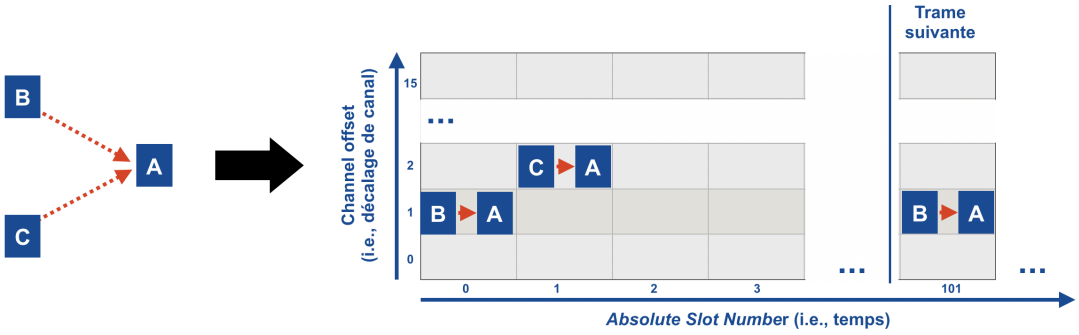


Fig. : B and C's communications with A are programmed in the slotframe.

The initial communications enabling the object to be inserted into the routing topology take place in the shared cells. A shared cell will occur at a frequency chosen at random by the coordinating object. The latter will then use this to transmit control packets enabling it to build and maintain the routing topology. It will also use this to send its *enhanced beacons* (EB), containing information relating to the *slotframe*.

Any objects looking to join the network must listen to each of the 16 available frequencies until a beacon has been received. The latter will supply them with the information needed for synchronisation with the coordinator and for insertion into the routing topology.

4.3. 6LoWPAN : IPv6 for the IoT

4.3.0. Discovering 6LoWPAN and UDP: IPv6 for the IoT

The 6LoWPAN adaptation layer is defined within the IETF. It enables constrained connected objects to use IPv6 addresses and to exchange over the existing internet.

The goal of this sequence is to understand the challenges linked to the use of IPv6 in the context of the IoT and the issues raised by its integration within constrained objects (compression, fragmentation). This sequence also introduces the concept of IPv6 stateless address auto-configuration on each item of equipment.

4.3.1. Discover IPv6 and 6LoWPAN

Content

4.3.1a Why 6lowPAN

A so-called LowPAN network is comprised of objects communicating wirelessly using limited resources (e.g. speed, energy). The radio links employ the use of standards such as IEEE 802.15.4, for example. The purpose of this sequence is to explain how these objects can be integrated into the internet as we know it. To put it another way, by taking space and processing constraints into account, we will see how IP (Internet Protocol) can be established within these networks in order to be able to access these objects in a way that is as transparent and as upscalable as possible.

Deploying IP-compatible objects means facilitated integration of objects in existing infrastructures, which have demonstrated their robustness and their capacity for upscaling. With IP-compatible objects, there is no need for intermediary gateway machines, and so the end-to-end principle, an essential aspect of internet architecture, is maintained.

IP connectivity requires a significant amount of effort given the objects in question. This is as much about being able to address objects (i.e. making them exist on the internet), as it is about being able to reach them (i.e. routing).

First and foremost, the sheer quantity of connected objects, both those around currently and those expected to arrive in the future, means a vast addressing space is required. Then, there is the fact that the density of LowPAN networks prevents users or administrators from attempting manual operations on each object. This means that auto-configuration mechanisms are required, through which each object can determine the IP address it will use.

These two constraints can be met by using IPv6, which is why this was chosen as the protocol to use for interconnectivity with the internet.

We will now take a look at what this protocol consists of, how it handles lower-level information and addresses (e.g. IEEE 802.15.4) and how it can be adapted for use on the constrained objects in question.

4.3.1b IPv6 recap

An IPv6 address is coded on 128 bits, meaning a huge addressing space is required (i.e. roughly 100 IP addresses for every atom on Earth).

These addresses are written hexadecimally in order to make them easier to read. This can also be compressed. In this way, an address such as 2001:0db8:0000:85a3:0000:0000:ac1f:8001 can be abridged to 2001:db8:0:85a3::ac1f:8001. "::" designate the longest sequence of 0s, while "0"

summarises a sequence of consecutive 0s over 2 bytes (between two consecutive ":").

As is the case for IPv4 addresses, a part of the IPv6 address is specific to the communication interface being addressed, with the other corresponding to that of the network in which it is located.

Aside from these addresses, IPv6 provides simplified packet headers, making it easier for them to be processed by routers responsible for conveying them. Lastly, the IPv6 protocol comes natively with security (IPsec), service-quality and auto-configuration mechanisms.

It was these characteristics that prompted the research into IPV6 being integrated into LowPANs formed of constrained objects. These contributions were made in the context of the IETF's 6LoWPAN working group, and we will now take a look at their most important contributions:

- header compression
- packet fragmentation
- address auto-configuration

4.3.1c The challenges of encapsulating IPv6 packets in IEEE 802.15.4 frames

As mentioned in RFC 2460, IPv6 requires that every link in the internet have a maximum transmission unit (MTU) of 1280 octets or greater. On any link that cannot convey a 1280-byte packet in one piece, link-specific fragmentation and reassembly must be provided at a layer below IPv6.

The total size of an IPv6 packet is thus minimum 1280 bytes. These IPv6 packets must then be encapsulated in IEEE 802.15.4 frames, the maximum size of which is only 127 bytes.

What's more, the header itself takes up 25 bytes. In addition, it is highly recommended that 21 out of the remaining 102 bytes be used to encrypt the data being transmitted. This leaves 81 bytes for encoding the IPv6 packet, but the IPv6 header alone takes up 40 of these. If we are to suppose that a protocol such as UDP is being used, which uses 8-byte headers, that only leaves 33 bytes for the data from the application. If TCP is being used, its 20-byte header will leave only 21 bytes for data.

In order to be able to use or even to consider applications generating more data, it is essential to free up some room in the IPv6 packet, by compressing the header, for example. If the packets are still too large, transmissions can be fragmented, and put back together again once they reach their destination.

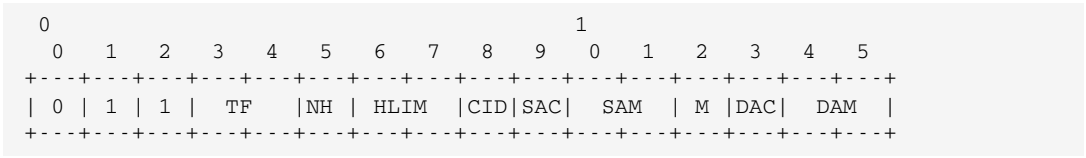
These header compression and fragmentation mechanisms were standardised within the IETF's 6LoWPAN working group (RFC 6282 and 4944 respectively).

4.3.1d IPv6 header compression

The IPv6 header compression mechanism (LOWPAN_IPHC) uses information available in the 6LoWPAN layer. Let's suppose that the communications using LOWPAN_IPHC rely on IPv6, that the size of the payload can be subtracted from the lower layers (e.g. from the fragmentation header detailed below), or that the interface addresses are generated based on prefixes specific to the underlying IEEE 802.15.4 links.

The LOWPAN_IPHC compressed header contains information which can be used to re-establish the source and destination addresses, in addition to the headers following them.

The LOWPAN_IPHC header has the following format:



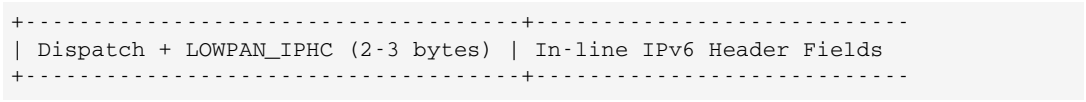
The first three bits (011) indicate that this is an IPv6 packet from a UDP traffic stream and that it has been compressed. The LOWPAN_IPHC header starts with the bits 3 and 4 (TF), which represent the traffic class and the flow label. This information takes up 28 bits in a standard IPv6 header. In 6LoWPAN, only 4 possible configurations have been retained, meaning 2 bits are enough to identify them.

The field NH (Next Header, 8 bits in an uncompressed header) is reduced to one single bit while the field HLIM (Hop Limit) only covers 4 possible values (no compression, 1, 64 or 255), moving from one byte to just 2 bits.

On the next bit, the field CID (Context Identifier Extension) indicates which context elements are to be taken into consideration for address compression. This latter is specified using the fields SAC (Source Address Compression) and DAC (Destination Address Compression), with 1 bit each indicating whether or not this was a case of context-based compression. The fields SAM (Source Address Mode) and DAM (Destination Address Mode), meanwhile, specify how the addresses were compressed (only 2 bits needed to code the 4 possibilities corresponding to each SAC or DAC value, giving 8 possibilities in total for address compression), while the field M (Multicast Compression) needs only one bit to indicate whether or not the recipient address is a multicast address.

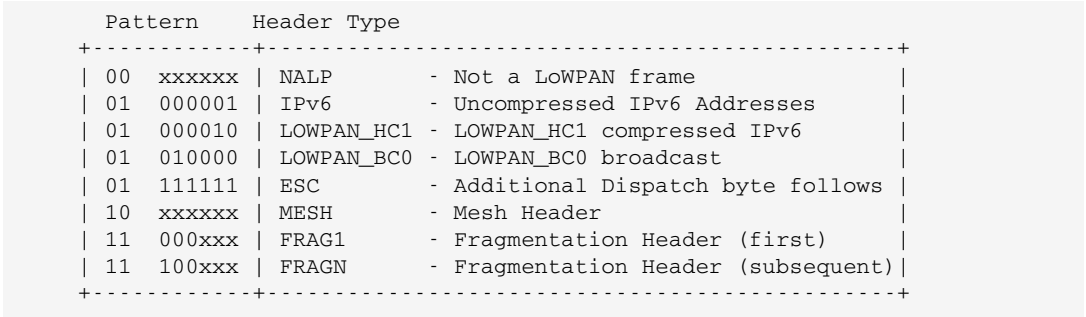
In this way, only 8 bits in total are used to compress the source and destination addresses, which would take up 128 bits each in a non-compressed header.

A LOWPAN_IPHC compressed header takes the following shape:



where the field Dispatch can be used to identify the type of header to follow, i.e. the type of packet encapsulated in the IEEE 802.15.4 frame.

Here are a few examples of values and their correspondences for the Dispatch field:



Note here that the LOWPAN_IPHC header can use 5 bits from Dispatch in addition to its byte, meaning it can be made up of 13 bits. This contains all fields of the IPv6 header that were compressed. The others then come after (In-line IPv6 Header Fields). The best-case scenario occurs with link-local communications, where IPv6 headers can be reduced to 2 bytes (instead of 40) - one for the Dispatch and the other for LOWPAN_IPHC. In cases where other IP hops have to be borrowed by the packet, the IPv6 header must include 7 bytes: 1 for the Dispatch, 1 for the LOWPAN_IPHC, 1 for the field Hop Limit and 2 each for the source and destination addresses.

This compression can increase the payload of the packet to 75 bytes.

4.3.1e Fragmentation in 6LoWPAN

In cases where an IPv6 packet (irrespective of whether the header has been compressed), can be included in an IEEE 802.15.4 frame, normal encapsulation takes place, without any fragmentation. If this is not the case, it is possible to generate a number of fragments and to add a fragmentation header that will enable reassembly once it reaches its destination. The first fragment will contain the following header:

```
                1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 0 0 0|    datagram_size    |          datagram_tag           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Here you will find the 5 bits of the Dispatch indicating that this is a first fragment. The field datagram_size specifies the size of the IP packet prior to fragmentation, while the field datagram_tag is a label, which is identical for all fragments of the same IP packet.

The following fragments must then contain the following fragmentation header:

```
                      1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 0 0|    datagram_size    |          datagram_tag           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|datagram_offset|
+-+-+-+-+-+-+-+-+
```

Here you will find the 5 bits from the Dispatch positioned at 11100, signifying that this is a fragment of an IPv6 packet, but not the first fragment. The fields datagram_size and datagram_tag will remain, while the field datagram_offset indicates the position of this fragment within the IPv6 packet.

4.3.1f Address self-configuration

We have seen how the IPv6 header can be compressed. One of the most important factors in this compression concerns the absence of the source and destination addresses (32 bytes), which are replaced by one single allocated byte in LOWPAN_IPHC. A total of 31 bytes out of a standard IPv6 header's initial 40 are saved in this way.

The goal is now to understand how these source and destination addresses can be re-established based on the information contained within this byte from the LOWPAN_IPHC and other context elements available within the 6LoWPAN adaptation layer.

Among the available information is the information on the lower layer, IEEE 802.15.4. First and foremost, all objects employing the use of this technology have an extended unique identifier. Globally, all communication interfaces (whether wired or wireless) are given a unique number by their manufacturer (an Organisationally Unique Identifier, OUI). This is comprised of three bytes representing the manufacturer, followed by three others forming the object's serial number. This unique number (OUI) can be used as a basis for extended identifiers, such as EUIs.

This means that a 64-bit EUI number is available for IEEE 802.15.4 objects. The network in which an object operates also has an identifier, called a prefix, which has 64 bits. Put together, these two pieces of information (the network prefix followed by the interface identifier) form an IPv6 address that is specific to this item of equipment:

```
   10 bits            54 bits                  64 bits
+----------+----------------------+----------------------------+
|1111111010|         (zeros)      |    Interface Identifier    |
+----------+----------------------+----------------------------+
```

This is called stateless auto-configuration. There are many other use cases, meaning objects must configure their IPv6 address depending on the messages received from border routers.

4.3.1g References

- Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks (IETF RFC 6282)
- Transmission of IPv6 Packets over IEEE 802.15.4 Networks (IETF RFC 4944)

4.4. CoAP: le protocole d'application Internet

4.4.0. Introduction

The CoAP constrained application protocol is a web protocol standardized by the IETF (RFC 7252). It is designed for IoT applications with resource-constrained devices (CPU, memory, battery) connected through rate-limited radio links, such as in LRWPAN networks. It is a protocol that uses many of the terms and concepts of the HTTP protocol with the Core model (Constrained RESTful Environments). It is based on standard Web models with a request-response interaction principle, uniform interfaces allowing interoperability and resources addressable by URIs.

In this sequence we will describe the architecture of the CoAP protocol and the format messages take, before giving some examples of interactions.

References

- The Constrained Application Protocol (CoAP) -https://tools.ietf.org/html/rfc7252
- Constrained RESTful Environments (CoRE) Link Format -https://tools.ietf.org/html/rfc6690
- Observing Resources in CoAP -https://tools.ietf.org/html/draft-ietf-core-observe-08
- Block-Wise Transfers in the Constrained Application Protocol - https://tools.ietf.org/html/rfc7959
- Datagram Transport Layer Security -https://tools.ietf.org/html/rfc6347
- Code register in CoAP messages -https://www.iana.org/assignments/core-parameters/core-parameters.xhtml#response-codes

4.4.1. The architecture of the CoAP protocol

CoAP is based on a client-server model, such as HTTP, where clients send requests on resources exposed by the server and identified by URIs. Unlike HTTP, CoAP works asynchronously based on UDP datagrams.

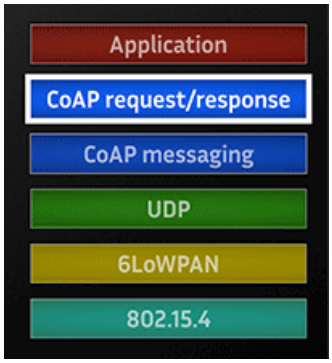


Figure 1: CoAP Architecture

It is based on two layers:

- a "Message" layer to manage the reliability and sequencing of end-to-end exchanges;
- a "Requests/Responses" layer to manage the methods and response codes in request-response type interactions.

The “Message” layer controls reliability and sequencing by processing identification in messages, but also the correspondence between requests and responses. This will be done by indicating/reading the Message IDs and tokens present in the message headers.

The “Requests/Responses” layer is based on a method of codification for requests and responses. For requests, it's based on the methods used by HTTP:

- ‘GET’ to obtain the representation of a resource
- ‘POST’ to create a new resource or to update an existing resource
- ‘PUT’ to update the resource with the attached representation
- ‘DELETE’ to delete a resource

Responses are also codified in the same way as in HTTP:

- ‘2.xx’ “Success” for a request that has been correctly received, understood and accepted
- ‘4.xx’ “Client error” to indicate an error on the client side
- ‘5.xx’ “Server error” to indicate an error on the server side

4.4.2. The format of CoAP messages

As we have just seen, in addition to the information to be transmitted, CoAP messages must contain a set of information relating to the two previously described layers. Messages are encoded in a simple binary format, with a well-defined structure: messages start with a fixed header of 4 bytes (32 bits), followed by a "Token" field of variable size between 0 and 8 bytes. There is then a field for options, and, lastly, the data occupying the rest of the datagram, preceded by a separator of one byte containing the value "1111111".

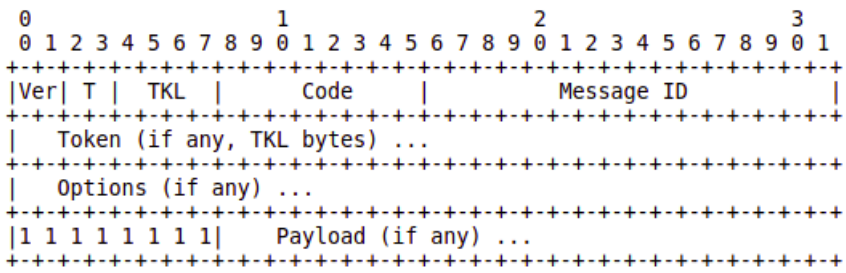


Figure 2: The format of a CoAP message

In detail, a CoAP message contains the following information :

- **Version (Ver)** (2 bits): indicates the CoAP version number used ('01' by default).
- **Type (T)** (2 bits): indicates the type of message in the context of a request or response.
  - Request:
    - 0: ***Confirmable (CON)*** : This message expects a corresponding acknowledgement message.
    - 1: ***Non-confirmable (NON)*** : This message is not waiting for an acknowledgement message.
  - Response:
    - 2: ***Acknowledgement (ACK)*** : This message is a response that acknowledges a confirmable message.
    - 3: ***Reset (RST)*** : This message indicates that a message has been received which cannot be processed.
- **Token Length (TKL)** (4 bits): indicates the length of the Token field, which can vary from 0 to 8 bytes.
- **Code** (8 bits): indicates (with the format X.xx: 3-bit class and 5-bit detail) whether the message is a request, a response or an empty message. For requests, it will also indicate the method used (GET, POST, PUT or DELETE); for responses, contains a code which identifies the nature of the response. Unlike HTTP, it is coded in binary in the message.
- **Message ID** (16 bits): serial number of the message, used to detect duplicate messages and to match an ACK/RST message with its initial CON/NON message.
- **Token** (0 - 8 bytes): field whose value is generated by the client, and whose size is indicated by the TKL field, used to match a request with the response. For example in the case of an empty acknowledgement that precedes a subsequent response.
- **Options**: contains additional information related or not to the type of the message. It is the equivalent of HTTP headers with for example the "Max-Age" parameter to define the validity period of the transmitted data.
- **Payload**: contains the data transmitted by the application (the payload start marker is 0xFF). The size of the payload must not exceed 1024 bytes but a CoAP data block transfer mechanism called "Block-Wise" allows the sending of different fragments each considered as a message.

4.4.3. Examples of interactions

Content

4.4.3a Non-confirmable messages

A message that does not require reliable transmission, without any guarantee of good reception, may be sent as a Non-Confirmable Message (NON). An example is a regular temperature sensor measurement over a long period of time, where the non-receipt of a message is not critical and it is possible to wait for the next request. In this case, messages are not acknowledged, but retain token information for duplicate detection. Figure 3 illustrates this scenario.

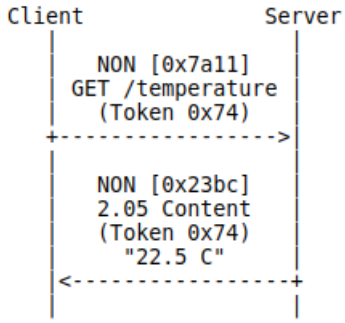


Figure 3: a non-confirmable message

4.4.3b Confirmable messages

Figure 4 contains a basic example of a confirmable message (CON). A client requests the temperature on the server with a GET request. The server responds immediately with an ACK response, the value read by its sensor. There are two important points to note:

- The request (CON) and the acknowledgement (ACK) have the same "Message ID". This allows the implementation of a reliability mechanism with the client which can retransmit an identical confirmable message until it receives the corresponding acknowledgment or until it exceeds a maximum limit of attempts (MAX_RETRANSMIT which is 4 by default). The interval between retransmissions increases exponentially with the number of failed attempts.
- The acknowledgement message also contains the response data with the temperature value. CoAP uses the piggybacking technique to optimize transfers and reduce bandwidth.

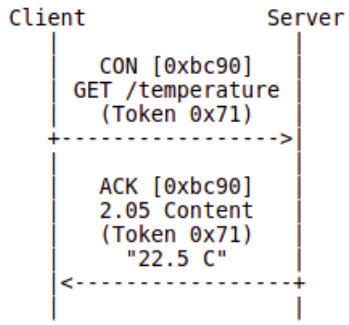


Figure 4: A GET request with a confirmable message

4.4.3c Asynchronous requests

If the server is not able to respond immediately to a request, it can send a simple acknowledgement to the client to indicate that it has received the request and is trying to get a response. Once it has the necessary data, it will send it to the client in a message containing the same token. In this way, the client can identify the original request and interpret the response correctly. Figure 5 illustrates this case. Since the response containing the requested value is a confirmable message, the client returns an acknowledgement after receiving it.

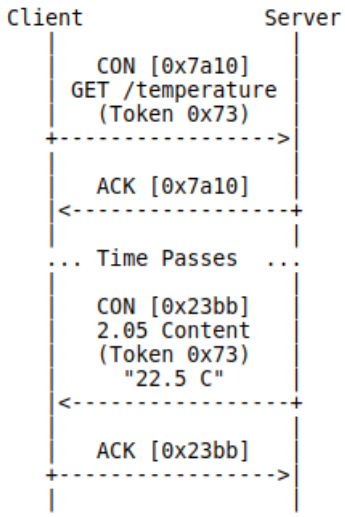


Figure 5: confirmable messages with empty acknowledgements

We can also add that, in the case of a server that is unable to process a message, it can respond with a reset message (RST). The reset message must simply also contain the Message ID for duplication detection.

4.4.4. Resource discovery

In an IoT context where machines can interact with other machines (M2M), devices must be able to discover the resources exposed by others. The main function of such a discovery mechanism is to provide URIs (called links) for the resources hosted by the server, complemented by attributes about these resources and possible other link relationships. A well-known URI **/.well-known/core** is defined as the default entry point for requesting the list of links to resources hosted by a server. This link list is represented with a compact and extensible "Core Link" format that is specified in ABNF (Augmented Backus-Naur Form) notation. Thus the client can send a GET request to the server on this URI and in return receive the list of links to available resources such as a temperature or brightness sensor.

Here is an example of a request whose response contains two sensor resources:

- **REQ:**

```
GET /.well-known/core
```

- **RES:**

```
2.05 Content
</sensors/temp>;rt="temperature-c";if="sensor",
</sensors/light>;rt="light-lux";if="sensor"
```

In the answer we can see that a description of multiple resources is separated by a comma. There are also attributes that describe information useful for accessing the link such as **rt** (Resource type) which identifies the service and **if** (Interface) which contains a generic definition of the service.

Then you can access the temperature sensor resource in the following way:

- **REQ:**

```
GET /sensors/temp
```

- **RES:**

```
2.05 Content
22.5 C
```

4.4.5. Resource observation

A useful extension to CoAP allows you to observe changes in the state of a server over time. This is done through a "subscription" mechanism, like the [Observer] design pattern (https://en.wikipedia.org/wiki/Observer_pattern):

1. the client sends to the server a GET request on the URI it is interested, adding the option “Observe”
2. the server replies with the current status of the resource and registers the subscription. Each time the status of the resource changes, a notification will be sent to the clients registered for the resource.

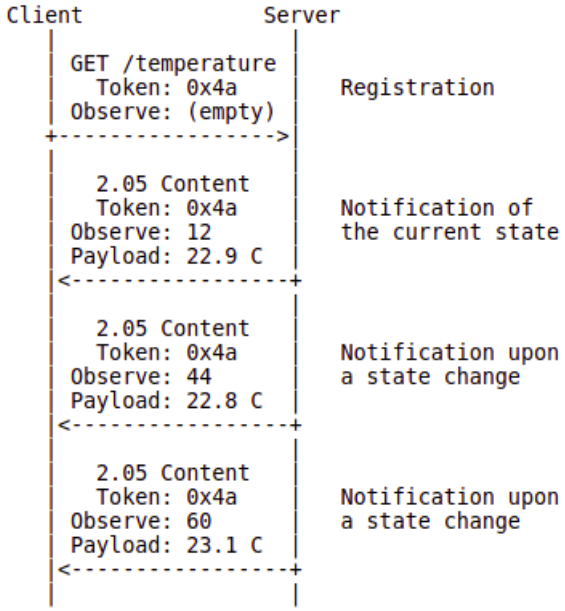


Figure 6: Resource observation

The "Observe" option is added for subscription, but also for value notification. In the latter case, a value is assigned to the option by the server in order to indicate the order for transmitting to clients.

This extension makes it possible, on the one hand, to optimize the use of bandwidth, since multiple requests are no longer necessary, and, on the other hand, the possibility of sending messages only at a given period and/or when there is a significant change in value, making it possible to make significant energy savings on the server side.

4.4.6. CoAP installation on FIT IoT-LAB

Video demonstration of the CoAP (Constrained Application Protocol) installation on the FIT IoT-LAB testbed.

4.5. Routing in Low-Power Wireless Networks

- 4.5.0. Introduction
- 4.5.1. The RPL routing graph
- 4.5.2. Creating and maintaining topologies
- 4.5.3. Managing the impact on memory
- TP13. RPL

In constrained networks, limitations in terms of radio range or transmitting power mean it is not always possible for emitting devices or receivers to communicate directly with each other. Their neighbours within the network will be called upon to relay the message gradually: this is known as multi-hop communication. However, in such cases, where multiple neighbours are available, there are a variety of possible paths to choose from. What this means is that it is necessary to define a set of mechanisms for selecting a relevant route/path for delivering the message, which is called routing.

In order to define a routing solution, we must focus on information exchanges, which can be categorised in accordance with different flows of traffic:

- point-to-point, for exchanging between two points on the network (*P2P*)
- point-to-multipoint, for sending from one original point to multiple recipient points (*P2M*)
- multipoint-to-point, for sending from multiple source points to the same collection point (*M2P*).

The figures below illustrate flows from one or more source nodes (S) to one or more destination nodes (D).

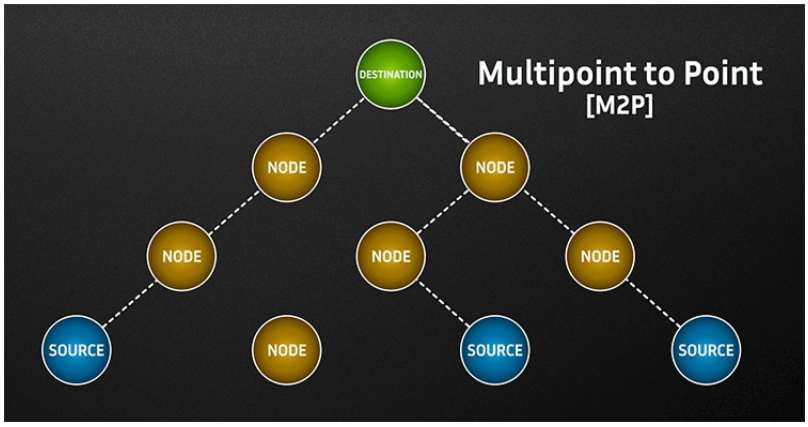


Figure 1: Multipoint-to-point (M2P)

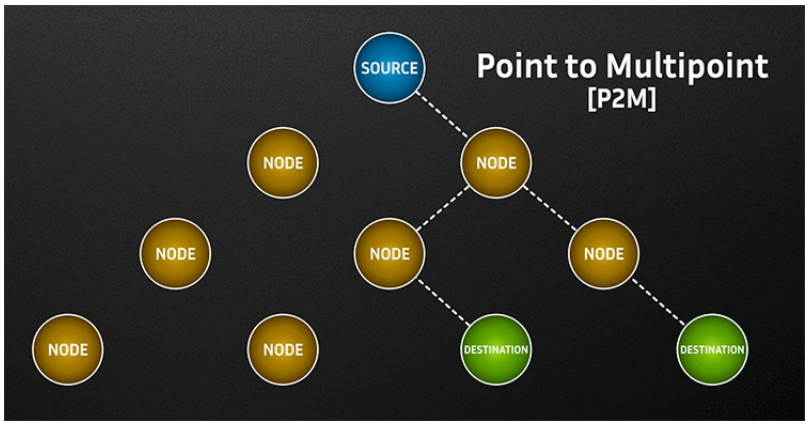


Figure 2: Point-to-multipoint (P2M)

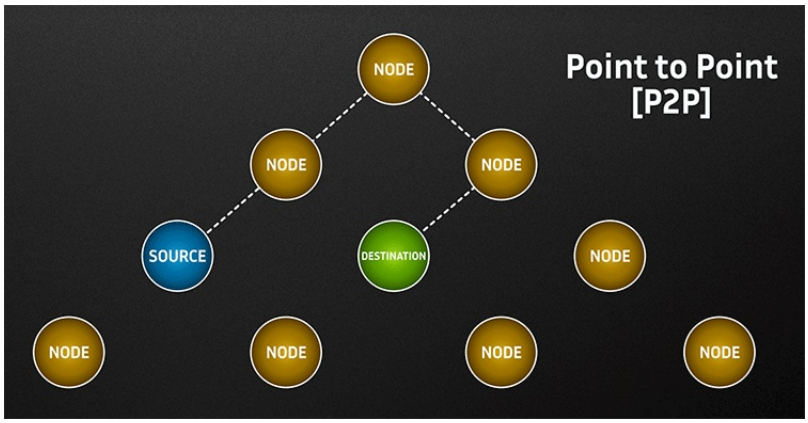


Figure 3: Point-to-point (P2P)

In the network types studied here, objects are most often deployed on an ad-hoc basis and retain a degree of instability as a result of using low energy-consumption wireless communications. What this means is that the network must be dynamic in order to be able to compensate for the disappearance, appearance or withdrawal of one or more nodes in the network. We also have to take into account the fact that this is reliant upon constrained material that is not particularly robust. Existing solutions for standard networks needed to be adapted in such a way that took these constraints into account.

The IETF (a global standardisation body) published the RPL standard, an IPv6 routing protocol for Low Power and Lossy Networks (LLNs), which was optimised for multi-hop and M2P communication, the most common type of traffic flow in wireless sensor networks (it also defines mechanisms for P2M and P2P).

During the rest of this sequence, we will go over the routing principles established by the RPL standard.

References

- RPL's RFC - https://tools.ietf.org/html/rfc6550
- RPL, the Routing Standard for the Internet of Things . . . Or Is It? -https://hal.archives-ouvertes.fr/hal-01647152/
- Dynamic configuration and routing for the internet of things -https://tel.archives-ouvertes.fr/tel-01687704/

4.5.1. The RPL routing graph

RPL defines a routing structure in the form of a*Destination-Oriented Directed Acyclic Graph* (**DODAG**). This is a graph directed towards a*root* node which does not form a cycle.

In order to prevent loops from forming, the standard is based on a "ranking" concept, e.g. the relative position of a node in relation to the root compared to other neighbouring nodes. If a node is able to communicate with more than one node of a lower rank, it will select one as a preferred parent for feeding back the message. This structure meets optimisation requirements for M2P communications, with information fed back from the root.

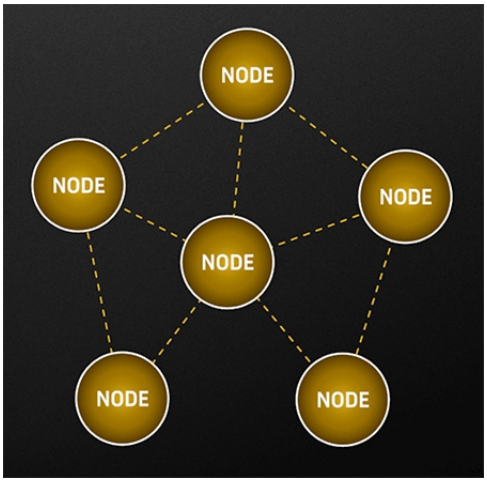


Figure 4: Communication links

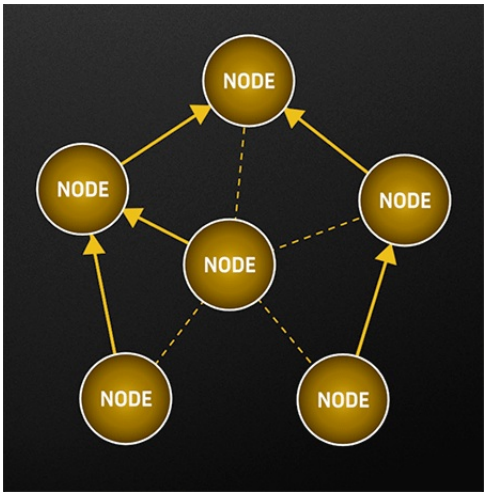


Figure 5: DODAG

Although we will describe the RPL mechanisms for a simple DODAG during the next part of this sequence, it should be noted that an RPL deployment may contain several instances. Each instance is defined by an objective function, which is used to define:

- how nodes select and optimise routes
- how one or more metrics can be used to calculate the rank
- how nodes select their preferred parent

An instance may contain more than one DODAG, which will share the same objective function - each node is only able to participate in one single DODAG. An RPL deployment, however, may contain more than one instance, while a node can participate in multiple instances. Consider a sensor network, for example, the main purpose of which is to feed back information on a regular basis for the purposes of monitoring equipment or areas. An RPL instance will be defined with an appropriate objective function in order to optimise the energy consumption of this network. However, although this same network must also be capable of sending alerts, messages must be sent as quickly as possible, and must be given priority. What this means is that its needs are different from the first. The network nodes will belong to a second instance, RPL, with a different objective function, which will concern itself less with energy consumption but which will favour quick, secure feedback.

4.5.2. Creating and maintaining topologies

The topology of the DODAG described previously is created and maintained using control packets.

The first of these are **DIOs** (*DODAG Information Objects*). Initiated by the root, which will indicate the routing metrics and the objective function, these are relayed to neighbouring nodes in order to build ascending routes. DIO packets coming from nodes of a equal or higher rank will be ignored in order to prevent loops from forming in the graph. In order to reduce the energy consumption of this multi-distribution system, which will repeat itself in order for the network to be maintained, this propagation is based on the Trickle algorithm, which provides a compromise with the reactivity to changes in topology. Announcements will be more frequent when the network is unstable, and more spaced out when the network is stable.

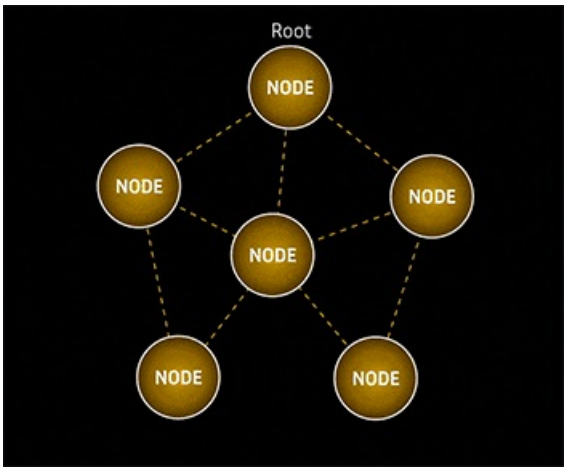


Figure 6: DODAG built using DIOs

The second of these are **DISs** (*DODAG Information Sollicitations*). These correspond to information requests made by nodes wanting to join the network or by nodes seeking more recent information. Any node receiving a DIS will respond directly to the source node using a DIO packet.

The last of these are **DAOs** (*Destination Advertisement Objects*). These enable RPLs to support descending traffic. Each node transmits a DAO towards its preferred parent, signalling to them that it may be contacted. Depending on the mode chosen (see following section), this information will either be stored and compiled with the preferred parent or fed back as far as the root. In cases where the application does not require P2P or P2M flows, these last control packets may be deactivated in order to reduce the impact on memory.

4.5.3. Managing the impact on memory

As we have just discussed, P2M and P2P flows are made possible through information being stored on DAOs, identifying routes leading to nodes of higher rank. This information can be stored in one of two ways.

Storing mode involves distributing the storage of this information in the graph, each node having a routing table between all destinations of its sub-tree. Messages will only go as far in the graph as the first common ancestor, thus optimising P2P flows. However, the protocol will only function if the objects have a sufficient storage capacity for the routing table of their sub-network.

Non-storing mode involves storing this information centrally in the root, making this the only node to retain routing information. As a result, it is necessary to have a root node with the requisite memory capacities. All messages will automatically go as far as the root before dropping back down towards their destination. For a P2P flow, even if the nodes are close together in the graph, this will be fed all the way back. For a P2M flow, the way in which the message drops back down the tree can be optimised.

The figure below illustrates the difference between these two modes when it comes to transmitting from a source node (S) to a destination node (D).

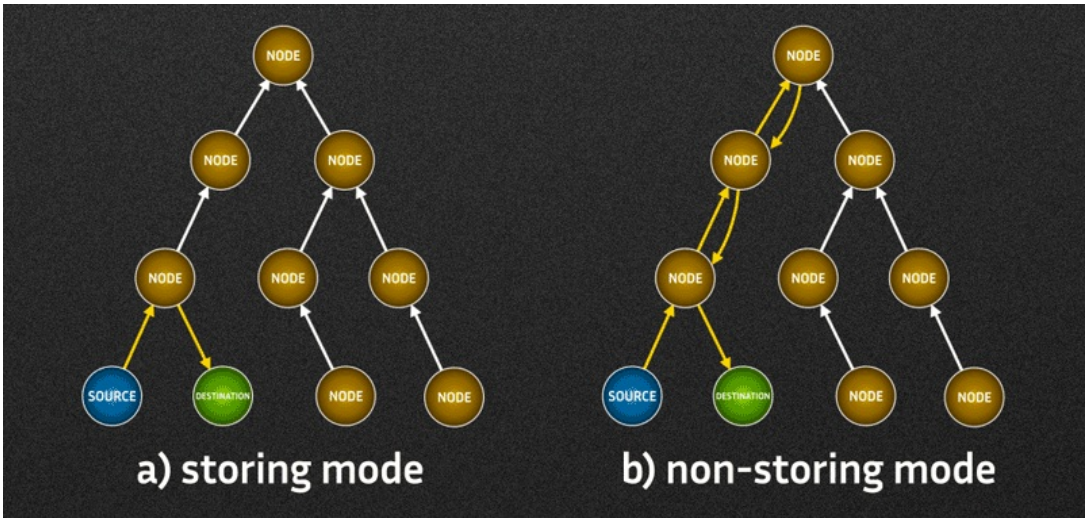


Figure 7: P2P communication comparison

The decision of whether to opt for*storing* or *non-storing mode* will depend on the memory capacities of the intermediary nodes and of the root node, as well as on the flows used most recently in the application.

Module 5. Securing Connected Objects

Objective: At the end of this module you will be able to identify the security problems of connected objects and the existing solutions to overcome them.

Hands-on activities (TP): TP14: Hash TP15: Encryption TP16: Signature TP17: DTLS communication TP18: Over the Air Firmware update

Contents of Module 5

5.1. Overview of Connected Objects Security Problems

- 5.1.0. Introduction
- 5.1.1. IoT and Security
- 5.1.2. Types of Attacks
- 5.1.3. In a nutshell

5.2. Cryptography for Connected Objects

- 5.2.0. Introduction
- 5.2.1. Crypto primitives for IoT security
- 5.2.2. Hash Functions
- TP14 : Hash
- 5.2.3. Encryption Schemes
- TP15 : Encryption
- 5.2.4. Message Authentication Codes
- 5.2.5. Digital Signature Schemes
- TP16 : Signature
- 5.2.6. Other Cryptographic Primtives

5.3. Network Security for Connected Objects

- 5.3.0. Introduction
- 5.3.1. Securing Communications
- 5.3.2. Local Communication Security for IoT
- 5.3.3. Transport Layer Security for IoT
- TP17 : Communication DTLS
- 5.3.4. Application Layer Security for IoT

5.4. Secured Update of a Connected Object Software (SUIT)

- 5.4.0. Introduction
- 5.4.1. The importance of Software Updates in IoT
- 5.4.2. SUIT-compliant IoT Firmware Updates
- TP18 : Over the Air Firmware update

Synopsis

Catalyzed by the availability of open-source, general-purpose, embedded IoT software and open standards for IoT communication, IoT is being increasingly deployed. As IoT is rolled out on billions of machines including very heterogeneous microcontroller-based devices, more reports pile up warnings about potential security threats -- and attacks which actually took place.

Cyberwarfare is nothing new: even before the advent of IoT, the Internet had become a battlefield. This cyberwar is in part government-driven (by geopolitics) and in part profit-driven (e.g. cyberpiracy).

In such a context, it is important to grasp what IoT currently offers in terms of functionality*vs* risk trade-off, what attack surface IoT needs to be secured against, and what mechanisms are used to mitigate attacks.

This module briefly overviews these aspects.

5.1. Overview of Connected Objects Security Problems

5.1.0. A Short Introduction to IoT Security

At the end of this sequence, you will be able to identifiy the different types of attacks targeting IoT devices, and the challenges these attacks incur.

Sources and references

- H. Tschofenig, E. Baccelli, "Cyber-Physical Security for the Masses: Survey of the IP Protocol Suite for IoT Security," IEEE Security & Privacy, 2019.
- S. Soltan et al. "BlackIoT: IoT Botnet of high wattage devices can disrupt the power grid" USENIX Security 18, 2018.
- E. Ronen, A. Shamir, "Extended functionality attacks on IoT devices: The case of smart lights" IEEE EuroS&P, 2016.

Video presenting IoT security challenges: compared to securing the Internet, what are the risks with IoT? What are the types of potential attacks? How to secure IoT devices? What are the constraints of IoT security?

5.1.1. IoT and security

Content

As IoT is rolled out on billions of machines including heterogeneous microcontroller-based devices, reports pile up, warning about potential security threats - and cyberattacks which actually took place.

Cyberattacks are nothing new: even before the advent of IoT, the Internet had become a battlefield, in part driven by governments (geopolitics) and in part driven by profit (cyberpiracy). In parallel, the emergence of surveillance capitalism also threatens end-users' privacy.

In this context, despite recurrent security and privacy breaches, the Internet continues to be used - because people consider that it still offers a reasonable tradeoff regarding functionality *vs* risks.

It is important to realise that it is generally impractical (if not impossible) to eliminate risk entirely. What security mechanisms aim for instead, is making risks *negligible*, given a set of assumptions.

So now, what is new with IoT?

5.1.1a IoT: New risks

IoT functionalities tend to improve faster than security, which becomes the main bottleneck. Compared to securing the rest of the Internet, aspects of low-end IoT security drastically modify both risks and constraints.

A hacked IoT system can cause direct physical harm, and may have severe safety implications. For instance, a hacked actuator such as a pacemaker or an insulin pump could kill, for example. A hacked smart lighting system could be weaponized to cause epilepsy. Hence, **the level of acceptable risk is changed**, compared to risks with a compromised email account, for instance.

With connected sensors everywhere, all the time, potential leaks of much finer-grained information are possible, potentially in real-time. Thus, **the scope of privacy breaches is changed**.

By maliciously combining IoT functionalities and increased network interdependence, catastrophic chain reactions could be triggered, and thus **entirely new types of attacks may be possible**. For instance, a relatively small botnet of smart heaters and coolers could disrupt a nation-wide powergrid, by maliciously synchronizing their maximum electricity demand.

5.1.1b IT security *vs* IoT security

Generally, compared to traditional IT security, solutions for the Internet of Things must be **significantly smaller** in order to fit microcontrollers' memory and CPU limitations,**but without compromises on execution times, or on security**, which is a challenge.

A typical example of this challenge is cryptography. Depending on the algorithm and on its implementation, a common issue on microcontrollers is outrageously long computation time, or a

memory footprint that is much too large.

5.1.2. Types of attacks

Content

To secure IoT devices, one must consider attacks of diverse natures. Let's start by distinguishing the main types of attacks.

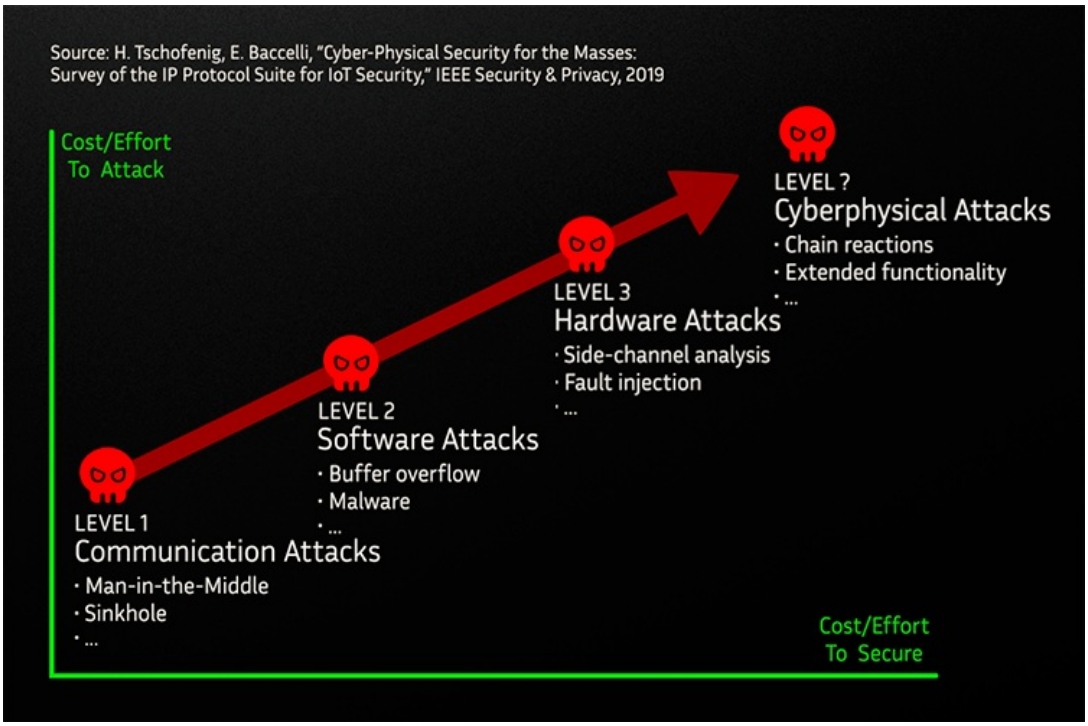


Fig. 1: Different types of attacks on IoT devices.

5.1.2a Network attacks

Network attacks aim to exploit the network as attack surface. During such an attack, a malicious intermediary inserts itself on the communication path and may:

- disrupt communication,
- analyse traffic sent over the network, or
- trick other entities on the network into amplifying another larger attack.

For instance, a *sinkhole* attack tricks elements into sending network traffic through a rogue node, which can dispose of the traffic altogether. A *man-in-the-middle* attack tricks end-points of a communication with a stealth intermediary, e.g. to maliciously bypass some access control checks.

This type of attack is cheap for attackers - who can easily set up tentative attacks, remotely, on a massive number of devices.

Specific network protocols must be used to defend against these attacks, while fitting both low-power communication constraints and resource constraints on microcontrollers.

5.1.2b Software attacks

Software attacks aim to exploit vulnerabilities in the embedded firmware. During such an attack, malicious logic input is fed to the software running on the IoT device which can lead to:

- The IoT device leaking data which should remain private, or
- An attacker covertly taking control of the IoT device.

For instance, *malware* is hostile software which is stealthily installed on the device, and which performs malicious operations. A *buffer overflow* exploits a software vulnerability (i.e. missing memory bound checks) to read or modify memory which should *not* be accessible normally.

These types of attacks are more cumbersome to put in place for attackers, but can bring higher value. For example, it can result in a botnet of compromised IoT devices.

Additional mechanisms must be used to defend against such attacks, while still fitting microcontrollers' resource constraints. For example, such mechanisms can protect reading, writing or executing some part of the memory. These are either themselves implemented in software, or are provided by specific hardware functionalities (e.g. a "trusted" execution environment such as ARM TrustZone).

As an example, before deploying and running some software, formal verification of this software can guarantee the absence of some vulnerabilities. After the software is deployed and running, if other vulnerabilities are discovered (which happens often down the line) secure updates over the network can patch the software to harden it, incrementally. Nevertheless, software updates themselves can become an attack vector, if a legitimate update is somehow laced with malicious logic.

5.1.2c Hardware attacks

Hardware attacks aim on the other hand to exploit vulnerabilities based on physical interaction with the IoT device.

For instance, a fine-grained analysis of power consumption during cryptographic operations may leak information that should remain private (for instance digits of a secret cryptographic key). This type of attack is called a *side-channel attack*.

Or, a malicious use of strong magnetic field around the device may cause this device to skip some critical checks. This type of attack is called a *fault injection* attack.

These types of attacks are very cumbersome as they require both physical access to a targeted device and, often, quite specific tooling.

Again, mitigating such attacks may require both modifying software (e.g. guarantee constant time execution of some cryptographic primitive) and modifying hardware (e.g. hardening the device).

5.1.2d Cyberphysical attacks?

On top of network, software and hardware attacks, new types of attacks might exploit

cyberphysical characteristics of the system.

A concrete example of such an attack is a *chain reaction* such as the one causing the electric power-grid disruption, using a smart heater botnet, which we described earlier in this section.

Another example is based on the principle of exploiting *extended functionality* on some elements in the system. The hacked smart lighting system we described earlier (weaponized to cause epilepsy) falls in this category.

In fact, such attacks are not easy to classify yet. Currently, research is only beginning to study them.

5.1.3. In a nutshell

In a nutshell, IoT security in practice must combine several defense mechanisms, working at different levels of the system, and protecting against different types of attacks. Each of these defense mechanisms is necessary, but not sufficient to achieve security overall.

In this module we focus mostly on mechanisms applicable on low-end IoT devices to:

- defend against network attacks, and
- defend against some software attacks (based on remote malware installation).

5.2. Cryptography for Connected Objects

5.2.0. Introduction

We are now going to briefly look into several crypto primitives and give you an overview of the typical solutions used in IoT on microcontrollers.

At the end of this sequence, you will be able to combine a variety of cryptographic mechanisms, applicable on-board microcontroller-based IoT devices.

Sources and References

- SHA-2 Specifications
- SHA-3 Specifications
- ECDSA with P-256 Specification
- RFC 8032 Edwards-Curve Digital Signature Algorithm (EdDSA)
- RFC 8439 ChaCha20 and Poly1305 for IETF Protocols
- FIPS 197 The Advanced Encryption Standard

5.2.1. Crypto primitives for IoT security

Cybersecurity currently relies on various combinations of mechanisms at work in hardware, and at different levels of the software and network stack.

Though diverse, all these mechanisms build upon on a set of common crypto*primitives*, i.e. specialized low-level operations on digital data. These primitives are used to provide basic guarantees on digital data, including:

- *Authenticity*: guarantee on the origin of the data;
- *Integrity*: guarantee that original data has not been tampered with;
- *Confidentiality*: guarantee that data is intelligible only for intended recipients.

Based on these, more elaborate guarantees can then be provided at higher level by the system, such as authorization, privacy...

IoT network and system security currently relies on various combinations of the following crypto primitives:

- Hash functions;
- Digital signature schemes;
- Message authenticators (MACs)
- Authenticated encryption;
- Key exchange schemes.

In the following part of these sequence, we will briefly look into each of the above categories, and give an overview of the typical solutions used in IoT on microcontrollers.

What is special about IoT?

Typically, cryptographic operations require rather intensive computation. On microcontrollers, computing power is (very) limited compared to microprocessors available on other types of machines (e.g. desktop, laptop, smartphone, tablet or even single-board computers such as a RaspberryPi).

In general, the challenge on low-end IoT is thus to achieve fast enough execution time with much smaller memory requirements, and without compromising on the primitive's security strength. In particular, for the most computation-intensive primitives (e.g. public key cryptography) this challenge is exacerbated.

One approach to overcome this challenge is to rely on very efficient hardware implementation of some primitives (e.g. a specialized crypto co-processor). However, if additional hardware modules are needed, the cost of devices increases. Furthermore, crypto valid today may not be valid tomorrow (because flaws are discovered, or because average attackers' computation power increases). Therefore, crypto often relies on logic implemented in software, and not only on hardware.

5.2.2. Hash functions

Content

5.2.2a What is a hash function?

A hash function is a function that maps data of arbitrary size to (smaller) fixed-size values, for example: 64 bytes. Such a value is called a *digest* or a *hash*.

A good cryptogaphic hash function should have a negligible risk of *collision*, i.e. that different input produce the same output. A better hash function provides the same or smaller risk, and requires less memory and/or less computation.

In a security context, hash functions are often used to provide a *one-way function*: deterministic and easy to compute for any input, but hard (computationally infeasible) to invert given the digest of arbitrary input. An important characteristic of hashing is that, the slightest change in the message to-be-hashed (for instance modifying the value of a single bit) typically results in a massive change in the resulting digest.

5.2.2b How/where is hashing used?

Since digests are *small*, they are nice to handle in a context where memory is scarce. Since digests are also of *fixed size*, handling a digest is simpler than handling the actual data, which allows implementation optimizations (e.g. avoids the complexity of handling larger, variable size input).

For instance, hashes are used to detect when data changed. In the context of secure IoT firmware updates, a hash function is used to produce a digest of the (binary) firmware image, and is sent together with the image. The receiver recomputes the digest locally (on the received firmware binary) and checks it against the received digest. If they do not match, an issue (corruption or tampering) is detected. Another example is how hashing data together with a shared secret key is used to create a message authenticator tag (see section 5.2.4. on MAC). Hashing is also used to to derive cryptographic keys from entropy sources.

5.2.2c SHA-3

SHA-3 (Secure Hash Algorithm 3) is the latest family of hash functions standardized by NIST (USA's National Institute of Standards and Technology). SHA-3 is based on a cryptographic primitive called Keccak, a so-called *sponge construction*.

Sponge functions compute hashes in two phases: (1) the absorption phase, and (2) the squeezing phase.

In the absorption phase, the data is split in blocks of fixed size r. For instance, for SHA3-256 the blocks are of size r=1088 bits. Bit scrambling is performed on the current block extended with

internal state of size *c*. A larger value of *c* increases security. For instance, for SHA3-256, *c*=512 bits.

Scrambling is performed on the *r*+*c* bits based on a series of bit permutations noted *f* composed only of simple operations (XOR, AND and NOT operations). The result is then XORed with the next block, and bit scrambling is performed again on the output. And so on until all blocks have been absorbed in the state (see Figure 1).

At this stage, the squeezing phase starts. This phase resembles the absorption phase, except rounds of bit scrambling (the same permutations) are performed directly one after the other. Each round produces a fixed-sized bit output: the first *r* bits are the result of this round of squeezing. The number of rounds can be chosen at will. Thus, Keccak is an extendable-output function (XOF) i.e. a cryptographic function producing arbitrarily long output (which can be used not only for hashing, but also for other purposes).

The hash is defined as the first output of the squeeze phase, truncated after the desired digest length. For instance, for SHA3-256, the output of the XOF is truncated after 256 bits.

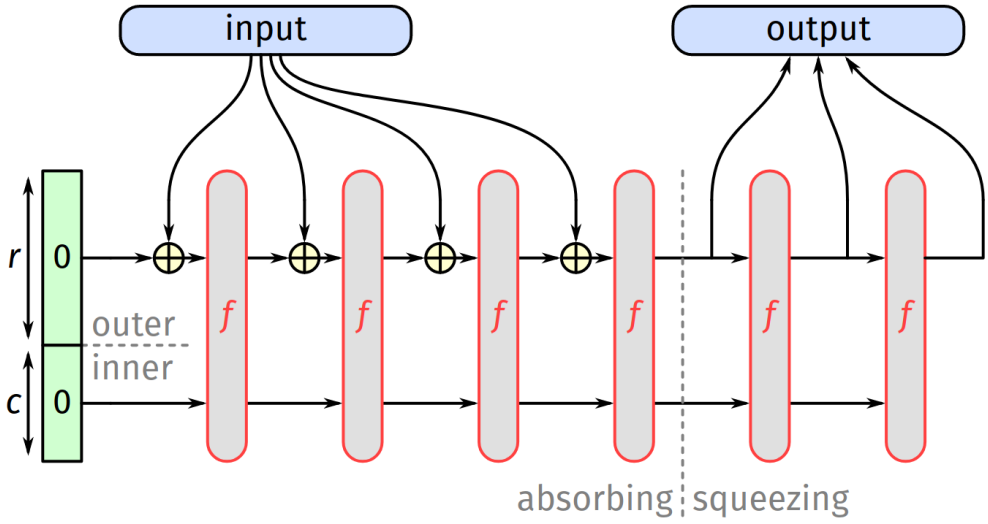


Fig. 1 : Sponge function used for SHA-3 hashing. (Source: G. Van Assche, RIOT Summit 2017).

5.2.2d SHA-2

SHA-2 is another family of hash functions previously standardized by NIST. SHA-2 is completely different from SHA-3 in its internal design: it is built using the Merkle--Damgard construction, and not a sponge function. It is not an XOF: it provides fixed-sized digests.

The most popular SHA-2 algorithm is without a doubt SHA256, which processes data in 512-bit blocks and produces a message digest consisting of 256 bits.

5.2.2e Other hashing primitives

There are many other hashing primitives. Some of them are deprecated in cryptographic contexts, because they have been shown to incur a non-negligible risk of collision. This is for example the case of MD5 message digest algorithm, which is considered unsafe in a cryptographic context nowadays.

5.2.3. Encryption schemes

Content

5.2.3a What is an encryption scheme?

A digital encryption scheme is a set of algorithms providing*a cipher*, i.e. a technique enabling the originator of digital data (referred to as plaintext) to encode this data using one or more keys. The encoded data, which we call *ciphertext*, reveals nothing about the plaintext (except for its length) unless it is decrypted with the key. Only authorized parties (who possess the required keys) can decrypt, i.e. recover the plaintext from the ciphertext.

A good encryption scheme ensures that it is practically impossible for unauthorized parties (who do not have the keys) to decrypt the ciphertext. A better encryption scheme provides the same or higher level of assurance, with less overhead in terms of memory and/or computation requirements.

5.2.3b How/where is encryption used?

For example, encryption is useful to protect the confidentiality of communciations over the network. As another example, in the context of secure IoT firmware updates, the firmware image binary could be captured by an attacker for vulnerability analysis and adversarial reverse engineering. To protect against such attacks, the firmware image can be encrypted before it is transferred to the IoT device targeted by the update, which can decrypt and recover the firmware image binary. In this case, the IoT device should be (somehow) provided with the necessary keys to decrypt.

5.2.3c Symmetric ciphers

Symmetric ciphers are based on the principle that the entity which encrypts a message and the entity that decrypts this message use the same key.

The Advanced Encryption Standard (AES) is probably the most popular symmetric cipher, to date. AES was standardised by NIST, and defines a family of ciphers which work on a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. For instance, AES-128 is the variant using a key size of 128 bits. The longer the key, the stronger the security guarantees. AES uses an encryption algorithm based on a complex combination of permutations and substitutions which is often implemented (and optimized) in hardware, even on low-end microcontrollers. Hardware implementations typically reduce both memory and energy budgets, compared to software-only implementations of this crypto primitive. For more details, the reader can refer to FIPS 197.

On devices that do not provide AES in hardware, encryption must be done in software. An alternative to AES which is faster in software is ChaCha20, another symmetric cipher designed by D. J. Bernstein and standardized by the IETF. For more details, the reader can refer to RFC 8439.

One crucial issue with symmetric ciphers however, is that it assumes the encrypter and decrypter share a secret (the secret key). The establishment of a shared secret (i.e. key distribution over an insecure network) requires a separate, additional mechanism: asymmetric cryptography.

5.2.3d Asymmetric Cryptography & Diffie-Hellman Key Exchange

Asymmetric cryptography is based on the principle that each entity is associated with a pair of keys: a public key and a private key. The public key is openly distributed (it is public knowledge), while the private key must be known only to the owner of this key pair. Assuming the above:

- anyone can encrypt a message using an entity's (let's call it A) public key, but this encrypted message can only be decrypted by A, using its private key.
- using its private key, entity A can produce a signature which can be authentified by anyone (using A's public key).

Compared to symmetric ciphers, asymmetric ciphers typically are more computation-intensive, but provide authentication properties and enable key establishment mechanisms that are not possible with symmetric ciphers alone.

In particular, Diffie-Hellman (DH) key exchange is a method using public key cryptography to establish a secret (a symmetric key) shared between two parties communicating over a (still) insecure channel.

One crucial problem with asymmetric ciphers, is being reasonably certain that the public key of the receiver is indeed legitimate (i.e. is owned by the entity which claims so). To alleviate this issue, a complex complementary infrastructure is used in practice, implementing a chain of trust and certificates based on digital signatures: the so-called Public Key Infrastructure (PKI). In low-end IoT however, using PKI is so far being avoided because certificates represent a significant additional overhead. Instead, IoT devices are pre-provisionned with the public keys of end-points they will communicate with over their lifetime (which is a limitation).

5.2.4. Message Authentication Codes (MAC)

A Message Authentication Code (MAC, sometime named*tag*) is a short, fixed-sized bit string, generated using a secret (symmetric) key, which can be used to authenticate a message of arbitrary length. Verifiers (who must possess the secret key) can check the tag to validate the origin and the integrity of the message.

For instance, Hash-based Message Authentication Code (HMAC) is a category of MAC which uses two rounds of hashing on the secret (symmetric) key and the message piece by piece to produce a tag. For instance, HMAC-SHA256 produces tags using SHA256 for hashing. Basing HMAC on other hash functions is possible. The size of the output (and the cryptographic strength) is the same as that of the underlying hash function.

5.2.5. Digital Signature Schemes

Content

5.2.5a What is a digital signature scheme?

A digital signature scheme is in general a set of algorithms which enables:

1. *Key generation*: the algorithm outputs a (random) private key and its corresponding public key.
2. *Signature generation*: given a digital message and a private key, the algorithm produces a (digital) signature.
3. *Signature verification*: given the message, the public key and the signature, the algorithm either confirms or invalidates the authenticity of the message.

A good signature scheme offers the recipient of digital data a reasonably high assurance that, if the signature verifies, this data was indeed originated by the entity to which the public key is associated. A better signature scheme provides the same or higher level of assurance, using smaller keys, and less computation to generate/verify signatures.

5.2.5b How/where are digital signatures used?

For example, in the context of secure IoT firmware updates, a digital signature is used to authenticate the origin of the firmware image update (validating legitimacy), and to ensure its integrity (ensuring it has not been tampered with).

Frequently, however, data which must be signed exceeds the maximum input size of signature schemes. In such cases the data is first hashed, which produces a short digest over which signature generation/verification is performed (instead of over the data itself).

Signature generation and verification are typically computation-intensive tasks which take significant time on microcontrollers -- typically much longer than hashing for instance.

5.2.5c Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC), is a technique exploiting the properties of equations defining elliptic curves and group theory. The main principle is to provide a *one-way function* by multiplying a point on the elliptic curve by a number, which will produce another point on the curve, but it is computationally difficult to find what number was used, even if you know the original point and the result. For instance, ECDH (Elliptic curve Diffie-Hellman) is a variant of DH which uses elliptic curve theory to achieve this goal.

Compared to alternative asymmetric cryptography techniques based on different principles (such as RSA, Rivest–Shamir–Adleman, based on the difficulty of factoring the produce of large primes), ECC can provide equivalent security with much smaller keys, diminishing required computing and

energy resources.

For these reasons, ECC is widely used in microcontroller-based IoT. Different elliptic curves may be used, which define different ECC standards.

5.2.5d ECDSA with P-256

Elliptic curve Digital Signature Algorithm (ECDSA) is a digital signature algorithm using elliptic curve cryptography.

ECDSA with P-256 generates and verifies digital signatures using the elliptic curve named the *P-256 curve*, standardized by NIST. For more details, the reader can refer to FIPS 186-3.

5.2.5e EdDSA with ed25519

The Edwards curve Digital Signature Algorithm (EdDSA) is a variant of the Schnorr signature scheme. Among other differences, it uses a different elliptic curve: the so-called ed25519 curve.

The specifications of EdDSA used with ed25519 have been standardized by the IETF. For more details, the reader can refer to RFC 8032.

5.2.6 Other Cryptographic Primitives

Quantum computing is expected to emerge in the (near?) future. The advent of such computing would provide potential attackers with novel and significantly powerful means -- powerful enough to break many cryptographic solutions considered safe nowadays. A challenge which looms next for cryptography, is thus efficient quantum-resistant cryptography. By that we mean: cryptographic primitives using traditional (non-quantum) computing, which can resist attackers which have access to quantum computing power.

A particularly tough challenge in this domain is designing quantum-resistant public key cryptography for IoT. A competition launched by NIST is ongoing, aiming to identify candidate cryptographic primitives which can address this challenge. This competition is on-going (it's a long process, lasting years). Selected primitives having proven their worth would subsequently be standardized. The implementation of a number of candidates primitives able to run on microcontrollers are being gathered by the PQM4 project (work in progress).

5.3. Network Security for Connected Objects

5.3.0. Introduction

We are now going to focus on a selection of mechanisms applicable on low-end IoT devices which secure communication for medium access protocols, for transport protocols and for content-aware protocols.

At the end of this sequence, you will then be able to combine protocols securing IoT communication, working at different levels of the network stack, preventing different types of attacks.

Sources & References

- H. Tschofenig, E. Baccelli, "Cyber-Physical Security for the Masses: Survey of the IP Protocol Suite for IoT Security," IEEE Security & Privacy, 2019.
- IEEE Standard for Low-Rate Wireless Networks,IEEE Std 802.15.4-2015
- RFC 6347: DTLS1.2
- RFC 8446: TLS1.3
- RFC 7925: TLS and DTLS profiles for IoT
- RFC 8152: COSE
- RFC 8613: OSCORE

5.3.1. Securing Communications

A network stack is typically modular, consisting in a set of complementary mechanisms. Each mechanism is specified as a protocol, which defines the expected behavior of elements in the network. The reader may recall that protocols coarsely fall into the below categories.

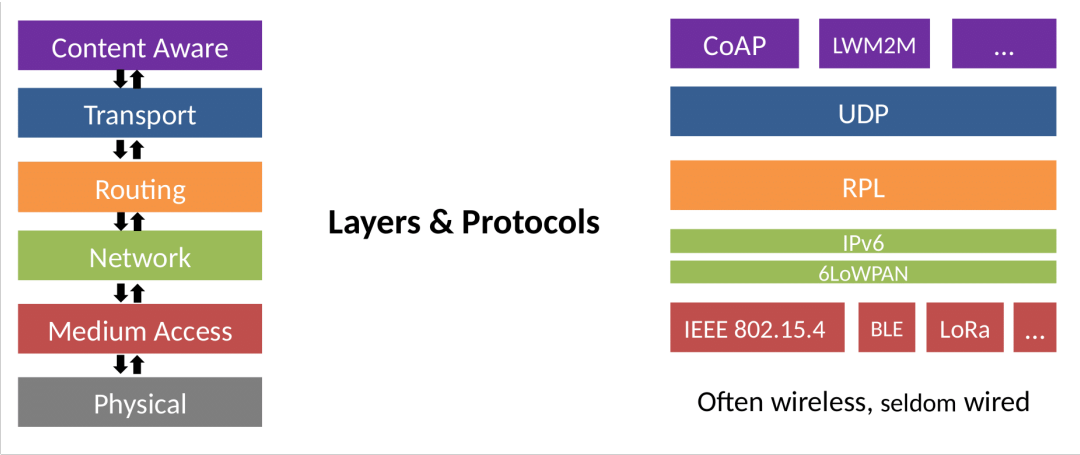


Fig. 1: Categories of protocols and a low-power IoT network stack.

Bottom-up, in terms of abstraction level:

- *Physical layer protocols* perform the transmission bit per bit over a communication medium which connects 2 computers;
- *Medium access layer protocols* realize a local network and the transmission of packets over a link shared by computers;
- *Network layer protocols* realize a global network, by defining packet formats & addresses usable across heterogeneous local networks;
- *Routing protocols* establish & use paths across the global network;
- *Transport protocols* adapt the rate at which data chunks are sent across the global network (& verify chunks went through);
- *Content-aware protocols* can work with user-facing names like "www.blabla.com/picture.jpg"

Each protocol provides a service, on which ultimately depend other parts of the system (e.g. higher layer protocols, operating systems or application logic). Disrupting a single protocol can thus disrupt the whole system, potentially. Hence, each category of protocol is prone to specific attacks, which call for specific mitigation. For instance:

- *Physical layer protocols* are prone to passive attacks such as *wiretapping*, whereby an attacker eavesdrops on the communication medium, or to active attacks such as *jamming* whereby an attacker saturates the communication medium, which thus becomes unusable;
- *Medium access protocols* are prone to attacks such as *spoofing*, whereby an attacker attempts masquerading another entity, e.g. faking a legitimate access point. Communications of nodes who attach may be disrupted or eavesdropped);
- *Network layer protocols* are prone to attacks such as a *replay attack*, whereby valid data

transmission is maliciously repeated or delayed. A replay attack can for instance be used to fake the identity of a legitimate entity on the network;

- *Routing protocols* are prone to attacks such as a *sinkhole*, whereby an attacker maliciously attracts all the traffic from neighbor nodes by advertising an unbeatable (fake) routing metric. Traffic can then be disrupted or analyzed;
- *Transport protocols* are prone to attacks such as tricks making them send packets at a rate much higher than sustainable by the receiver or the network, resulting in dropped packets, overflowing buffers and/or network congestion;
- *Content-aware protocols* are prone to attacks such as *Distributed denial-of-service (DDoS) attacks*, whereby several compromised nodes in the network collude to trick instances of the protocol into simultaneously send large, irrelevant content towards a server, which is overwhelmed and becomes unusable;

In a nutshell, communication security in practice must combine several defense mechanisms, working at different levels of the network stack, and protecting against different types of attacks. Each of these defense mechanisms is necessary, but not sufficient to achieve communication security overall.

In this module we focus on a selection of mechanisms applicable on low-end IoT devices which:

- secure communication for medium access protocols;
- secure communication for transport protocols;
- secure communication for content-aware protocols.

What is specific to IoT?

As already seen in Module 4, compared to the protocol stacks available on usual types of networks (e.g. TCP/IP over Ethernet or WiFi), a low-power IoT network stack typically consists in different protocols (see Fig. 1). For instance: CoAP is typically used instead of HTTP, hence mechanisms securing CoAP are necessary (instead of mechanisms securing HTTP). Thus, security mechanisms must be designed and applicable for these specific protocols.

In particular, security mechanisms and protocols for IoT must be able to function with low-cost, low-power devices. Roughly, compared to mechanisms running on other machines on the Internet, this means that IoT security mechanisms and protocols must be much more frugal, and in particular must:

- require 10^6 less (RAM and Flash) memory;
- require 10^6 less (control and user) traffic throughput;
- support much smaller packet size (very efficiently);
- function with much weaker CPU, 10^3 slower or less;

Another IoT specificity which must not be underestimated is a difference concerning the human factor, compared to securing a mobile phone or a PC. For the latter, a convenient human-computer interface (on-board screen/keyboard) and a 1:1 relationship with a human user can be assumed most of the time, and can thus be relied upon e.g. to provide credentials for bootstrapping, or to make common sense decisions such as approving a software update. On the contrary, in the case of low-power IoT devices, not only are typical human-computer interfaces typically absent, but also humans often have a 1:N relationship with IoT devices. These characteristics yield the need for much more automation, which is a challenge, especially at

bootstrap phases.

Last but not least, secure storage of secrets (e.g. cryptographic keys) is a challenge on simplistic low-end IoT devices which may not provide traditional hardware mechanisms preventing unauthorized access to sensitive data stored on-board (e.g. credentials).

5.3.2. Local Communication Security for IoT

Content

Assume nodes A and B (e.g. an IoT device and a wireless access point) can communicate with a MAC/PHY protocol over a shared medium -- a local network. In this context, Medium Access layer security aims to protect such communications against attacks such as eavesdropping, spoofing or replay.

Security mechanisms are specified differently depending on the actual protocols used at the MAC and physical layers. In the following we look more specifically into security mechanisms for IEEE 802.15.4 radios.

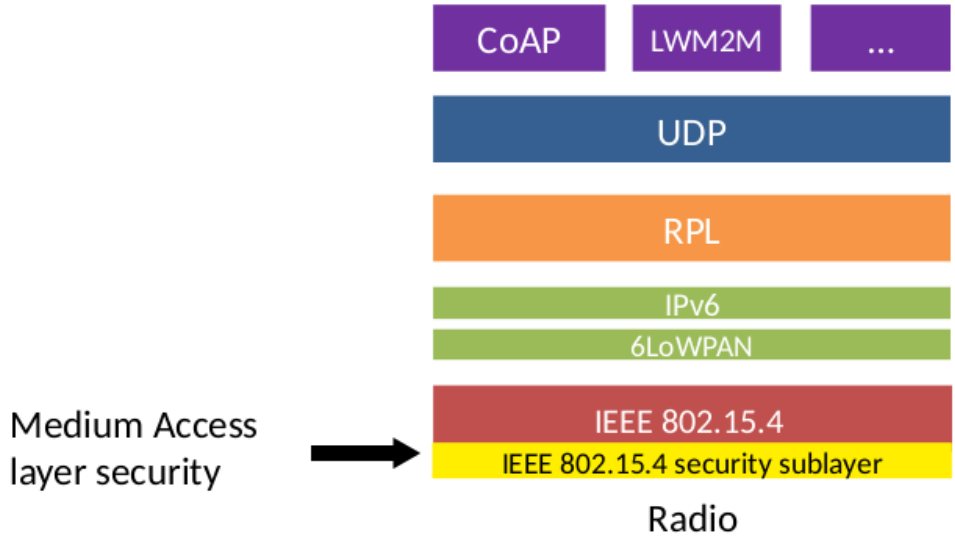


Fig. 2: MAC layer security within a typical IoT network protocol stack.

5.3.2a IEEE 802.15.4 Security

IEEE 802.15.4 security is based on symmetric cryptography with AES. Key distribution, establishment and maintenance is outside of the scope of the IEEE 802.15.4 standard.

Typically keys are pre-provisionned, and AES encryption/decryption is performed in hardware (generally much more efficient than in software).

Payloads (clear text messages) are broken into blocks of fixed size. Each block is encrypted/decrypted using AES. Different configurations are possible, which specify security "modes" detailing how AES is used exactly (see below).

Furthermore, packets sent over the radio must carry additional control information (header and suffix) as described in the following.

Security sublayer

Encryption/decryption is performed by a security sublayer, which is inserted in the stack, between the physical layer and the normal IEEE 802.15.4 packet processing.

Additionally to performing cryptographic operations, the security sublayer also extends outgoing packets with a supplementary header and a suffix dedicated to security.

Incoming packets are validated by the security sublayer using the supplementary header and suffix. If validated by the security sublayer, the decrypted packet is stripped from the supplementary security fields and passed on to normal IEEE 802.15.4 incoming packet processing.

Encryption modes

The basic encryption primitive used in IEEE 802.15.4 is typically AES 128. However, this primitive is used slightly differently depending on the mode of operation indicated in the security header.

Counter (CTR) mode -- In this mode, for each block, the successive values of a pre-agreed counter are used to cryptographically derive a bit mask that is specific to this block.

More precisely, the value of the counter is concatenated with a nonce (a non-secret pseudo-random number used once). This result is first encrypted with the (AES 128) symmetric key and then XORed with the (clear text) block, the result of which is the cipher text.

The receiver can decrypt and recover the clear text because it knows the key, the nonce and the current counter value, which is all that is needed to construct the bit mask.

Chained Block Cipher (CBC) mode -- In this mode, the clear text of each block is first XORed with the cipher text of the *previous* block, the result of which is then encrypted with the (AES 128) symmetric key.

The special case of the first block is XORed with a so-called Initialization Vector (IV), a non-secret pseudo-random number.

The receiver can decrypt with the key, assuming it temporarily keeps in memory the ciphertext of the immediately preceding block(s).

CTR with CBC (CCM) mode -- In this mode both CTR and CBC are used. For each block, CBC is first computed on the message to obtain a tag; the message and the tag are then encrypted using CTR.

Auxiliary Packet Header and Suffix

The security sublayer manages a table in memory which matches crypto keys with nodes identifiers (indicating which key to use, to communicate with whom, as well as current counter value and last IV used for each key).

Furthermore, the security sublayer manages appending, processing and validation of an additional header, and/or a suffix in IEEE 802.15.4 packets, dedicated to security control.

Several configurations are possible. The configuration used is indicated in the security header. Configurations include:

- AES-CTR-128: in this configuration, the payload is encrypted using the CTR encryption mode with AES 128.

- AES-CBC-MAC-128: in this configuration a Message Authentication Code (MAC) is suffixed to the payload. This MAC is computed using the CBC encryption mode with AES 128 (and truncated at 4, 8 or 16 bytes, depending on the configuration). This configuration guarantees message authentication and integrity, but not confidentiality of the payload.
- AES-CCM-128: in this configuration, the payload and the MAC are encrypted using CTR (the MAC is computed using CBC). This configuration guarantees message authentication and integrity as well as payload confidentiality.

The security sublayer validates incoming packets according to the configuration and cryptographic key currently in use for the originator of the packet. If checks pass, security header and suffix are stripped, and the packet is passed on for normal IEEE 802.15.4 frame processing. Else it is dropped.

5.3.2b Security Mechanisms Overhead

Note that *there is no free lunch*. While the above mechanisms provide guarantees in terms of confidentiality, integrity and authenticity of the data transmitted over the radio, these security mechanisms come at the price of increased frame length, CPU usage and memory footprint, which negatively impact performance regarding latency, throughput and memory usage.

Also note that security is *optional* in the standard.

5.3.3. Transport layer security for IoT

Content

One limitation of security at the link layer (e.g. encrypting with AES on IEEE 802.15.4) is that communications are decrypted at each intermediary on the path through the network. This caracteristic introduces vulnerabilities. Complementary security mechanisms must be used to guarantee security over a communication channel crossing several network intermediaries, e.g. transport layer security as described below.

Assume nodes A and B (e.g. an IoT device and a remote server on the Internet) can communicate end-to-end using a transport protocol. In this context, transport layer security aims to establish and maintain a secure communication channel between A and B over this transport protocol.

5.3.3a DTLS

DTLS (Datagram Transport Layer Security) is a protocol standardized by the Internet Engineering Task Force (IETF) which secures communications over UDP.

DTLS guarantees the integrity, authenticity and confidentiality of the data flowing through the secure channel it establishes from A to B, over UDP.

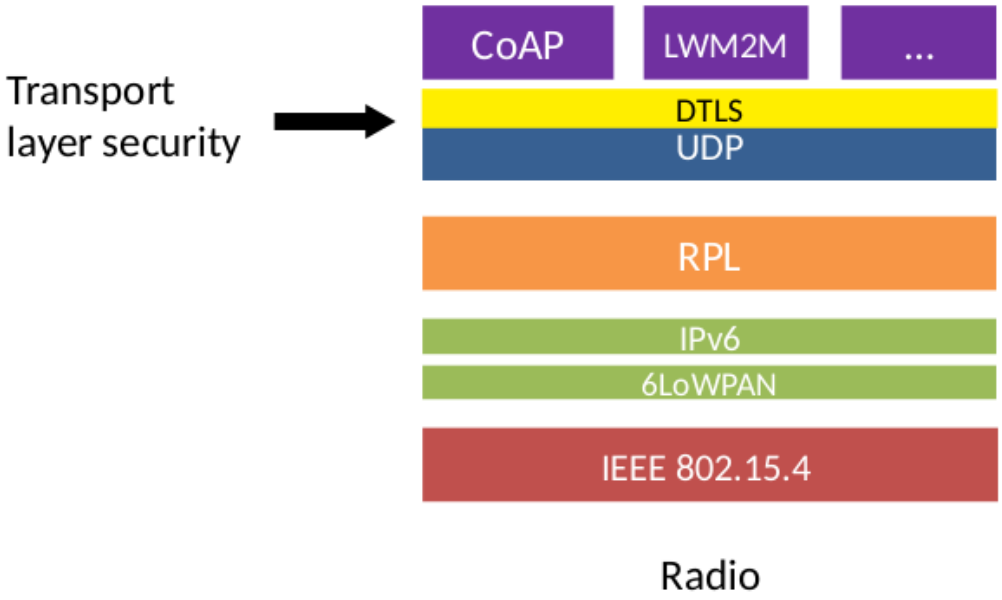


Fig. 3: Transport layer security within a typical IoT network protocol stack.

DTLS Handshake

DTLS relies on an initial bootstrap phase called the*handshake* layer. The handshake sets up a security context between two endpoints A and B. The end point initiating the handshake is called the *client*, while the other end point is called the*server*. The handshake results in the

authentication of A and B (using asymmetric or pre-shared keys) and the establishment of a master secret known only to A and B: a *symmetric key*.

DTLS Record Layer

In a second phase, called the*record layer*, the parameters and the symmetric key established by the handshake layer are used to encrypt and decrypt data (i.e. so-called *records*) sent from A to B, and vice-versa, ensuring confidentiality, integrity and authenticity. This key is used until the channel gets torn down or a maximum limit of records have been exchanged. The maximum limit depends on the algorithm. Beyond that point, should A and B need to communicate securely again, a new handshake is performed. Note that in particular, DTLS records contain a sequence number which is used to provide protection against replay attacks.

Ciphersuites

Within these principles, DTLS allows various configurations. Among others DTLS allows the use of a variety of different crypto primitives: different *ciphersuites*. During the handshake, prior to establishing the master secret, client and server automatically negotiate the cipher suite parameters to be employed. DTLS operation is more or less computationally-heavy depending on the ciphersuite used. The heaviest computation happens in the handshake, when the shared symmetric key is derived from A and B's public/private keys. However, on the one hand handshakes happen rather infrequently, and on the other hand specific ciphersuites can be used to decrease computation load on low-end IoT devices (see RFC 7925). For instance:

- TLS-PSK-WITH-AES-128-CCM is a ciphersuite which can be used in DTLS which is only based on symmetric key cryptography (AES 128 in CCM mode). It is relatively lightweight in terms of computation and memory requirements, at the price of less protection (prone to dictionary attacks, and weak forward secrecy).
- TLS-ECDHE-ECDSA-WITH-AES-128-CCM is another ciphersuite which can be used in DTLS, which is relatively more resource-consuming, but which provides stronger guarantees against dictionary attacks and forward secrecy. This configuration uses elliptic curve (asymmetric) cryptography during the handshake to derive symmetric keys used temporarily at the record layer.

5.3.3b CoAPs, DTLS versions and TLS

A typical use of DTLS for IoT is to secure CoAP communications over UDP. This construct (CoAP over DTLS over UDP) is sometimes named CoAPs or the coaps scheme, (whereby the "s" is for secure).

Currently, the most widely used version of the DTLS protocol is DTLS 1.2. However, a new version of the protocol (DTLS 1.3) is being finalized, and is expected to take over soon.

For sake of completeness, let's also add a few words about TLS. The Transport Layer Security (TLS) protocol is the equivalent of DTLS designed to operate over reliable, connection-oriented transports such as TCP, instead of operating over UDP.

Compared to DTLS, most of the mechanisms of TLS are the same, but some auxiliary mechanisms are absent, e.g. mechanisms detecting message loss and duplication are removed (as TCP guarantees reliable and in-order delivery of data). TLS 1.3 is the most recent version of this protocol.

5.3.4. Application layer security for IoT

Content

While security using CoAPs works well in a number of IoT use cases, it is not efficient for some IoT traffic patterns, and if a proxy intermediates, DTLS stops at the proxy and end-to-end security breaks. To cater for such use cases, application layer security can be used instead of transport layer security, as described next and shown Fig. 4.

Assume nodes A and B (e.g. an IoT device and a remote server on the Internet) can communicate end-to-end using a content-aware (application layer) protocol. In this context, transport layer security aims to establish and maintain a secure communication channel between A and B over this application layer protocol.

In the following we briefly overview two mechanisms used to secure communications over the application protocol CoAP.

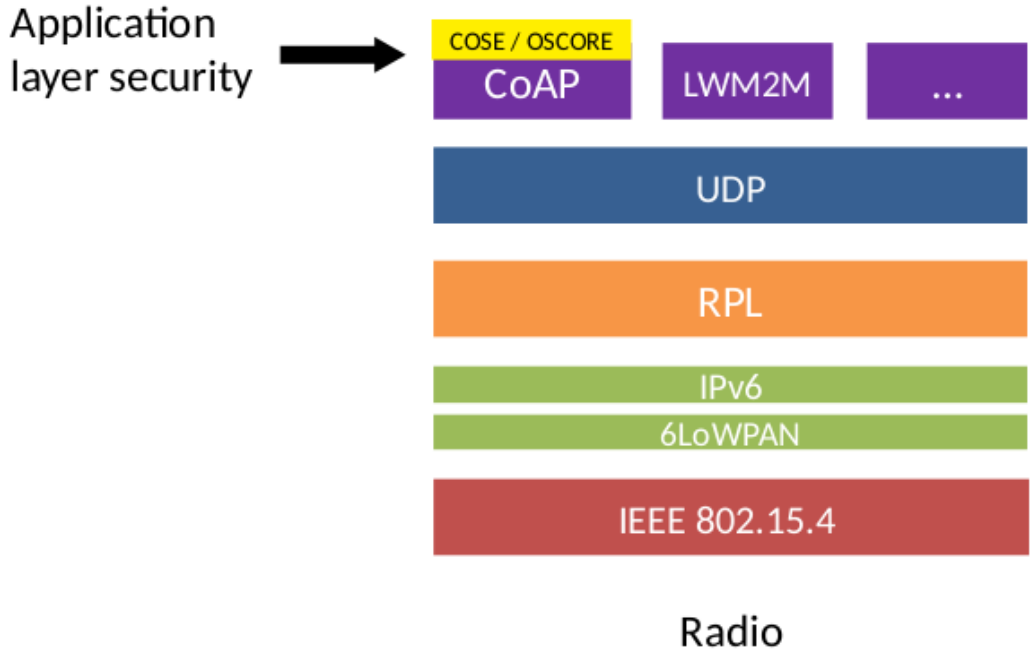


Fig. 4: Application layer security within a typical IoT network protocol stack.

5.3.4a COSE

Some traffic patterns in IoT incur infrequent, asynchronous communication, and "one-shot" payloads, e.g. software updates over the network. To secure such communication end-to-end, TLS/DTLS can be inefficient because of the handshake overhead. The IETF thus standardized a different security mechanism, COSE, which can be used over CoAP, and which we briefly cover in this section.

A recent trend in IoT is to use the Concise Binary Object Representation (CBOR) encoding and

serialization format. For reference, compared with the JavaScript Object Notation (JSON) format, CBOR was designed with much smaller code and message size in mind. COSE offers several security services, including digital signatures, counter signatures, Message Authentication Code (MAC), encryption as well as rudimentary key distribution methods using CBOR for serialization.

COSE can be seen as a set of building blocks that applications use in the way they find useful (keeping an eye on code size). A device implementing a firmware update solution may, for example, rely on asymmetric crypto and would, therefore, implement the signature verification capability offered by COSE and none of the other security services (e.g. no symmetric key crypto).

Securing IoT Data at Rest

An interesting characteristic of COSE is that it can be used to protect either synchronous data transmissions over the network, or asynchronous data transmissions, whereby transmitted data temporarily rests somewhere on the path towards its destination(s). An example of synchronous transmission is an IoT device sending its current sensor values to a dashboard application visualizing real-time data. An example of asynchronous transmission is the publication of a software update, which is first uploaded to a (untrusted) repository, and eventually downloaded by IoT devices, individually, when they are ready.

5.3.4b OSCORE

When CoAP is deployed with CoAP proxies and application layer gateways TLS/DTLS cannot provide end-to-end security because a classical DTLS exchange terminates at the gateway. To secure CoAP messages the IETF defines another communication security solution called Object Security for Constrained RESTful Environments (OSCORE).

OSCORE defines a CoAP option and a mechanism reusing COSE to protect CoAP messages, providing end-to-end encryption, integrity, replay protection, and binding of response to request (thus countering some related attacks).

Specific CoAP Option

An OSCORE packet is essentially a CoAP packet with an OSCORE Option, and an OSCORE Ciphertext as payload.

The OSCORE option is an OSCORE-specific header extending the CoAP packet header, which contains information such as a (sender) sequence number, and an ID which enables the receiver to figure out the security context (i.e. which pre-shared key to use) to decrypt the packet.

COSE Encryption

OSCORE plaintext consists of the CoAP payload, CoAP options and CoAP method code which, when encrypted with a COSE-defined encryption procedure, results in OSCORE ciphertext. The encrypted CoAP options (i.e. the inner options) are used to communicate with the OSCORE-aware endpoint, and are opaque to the intermediary proxies.

Note that other options (so-called outer options) are not encrypted and are thus not protected. The reason for this is that CoAP proxies must be able to inspect part of the message (including outer options).

Also note that OSCORE does not offer key management itself, it has to rely on a separate key

management mechanism. For each end-point with whom secure communication is required, OSCORE thus assumes that a shared secret (a symmetric key) is pre-established.

As such, OSCORE requires only symmetric cryptography operation. The default encryption scheme is AES 128 bit in CCM mode.

5.4. Secured Update of a Connected Object Software (SUIT)

- 5.4.0. Introduction
- 5.4.1. The importance of Software Updates in IoT
- 5.4.2. SUIT-compliant IoT Firmware Updates
- TP18 : Over the Air Firmware update

5.4.0. Introduction

Enabling software updates is necessary to secure IoT devices. Hence, we will now present aspects of securing software updates for microcontroller-based IoT devices.

At the end of this sequence, you will then be able to apply secure IoT firmware updates, combining mechanisms for hashing, digital signature, and secure low-power networking which we introduced in the previous sequences of this module.

Sources & References

- H. Tschofenig, E. Baccelli, "Cyber-Physical Security for the Masses: Survey of the IP Protocol Suite for IoT Security," IEEE Security & Privacy, 2019.
- SUIT Architecture Specifications.
- SUIT Manifest Specifications.
- K. Zandberg, K. Schleiser, Francisco Acosta, H. Tschofenig, E. Baccelli, 'Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check," IEEE Access, 2019.

5.4.1. The importance of software updates

Software in the Internet age has proven the maxim "you can't secure what you can't update". For instance, many IoT devices compromised by the Mirai botnet are still (!) unpatched to this day, and have become permanent liabilities. And yet, software updates are attack vectors themselves This duality makes secure software a challenge in general.

Beyond patching newly discovered bugs and vulnerabilities, modern software must be updated regularly to keep up with the continuing evolution of legal requirements, crypto deprecation, and threat models. Last but not least: software updates are also crucial to facilitate deploying new (or improved) functionalities, on IoT devices during their life time.

Hence, it is crucial to enable and secure IoT software updates. We can distinguish several cases of software updates:

- Model 1: monolithic software, single stakeholder: for example a firmware binary update;
- Model 2: multiple modules, single stakeholder: for example a run-time configuration update;
- Model 3: multiple modules, multiple stakeholders: for example a smartphone with multiple apps developed and maintained/updated by different software companies.

So far, in practice, microcontroller-based IoT devices mostly follow Model 1. IoT firmware updates take advantage of a combination of mechanisms we described in the previous sequences of this module: in particular, hashing, digital signature, and secure low-power networking.

What is special about IoT?

Obviously, modern Internet software is closer to Model 3, which fosters quicker innovation and better maintenance. This was most recently evidenced by smartphone software when it transitionned from Model 1 to Model 3 (before/after Android and iOS). Increasing complexity due to cybersecurity, interoperability, device management requirements might push low-power IoT software maintenance towards resembling more Model 3 in the future. But for now, "monolithic" firmware updates is the dominant software update technique for low-end IoT devices.

Secure IoT firmware updates employ a combination of crypto primitives (hashing, digital signature) and low-power networking protocols we introduced in the previous sequences of this course. Hence, secure IoT firmware updates are subject to the same constraints and specificities (see previous sequences of this module).

However, firmware updates add two important twists:

1. the memory budget is typically even more challenged because, instead of storing a single image, it must store multiple images and a bootloader. Furthermore, as a whole firmware typically does not fit in RAM memory, some operations (e.g. hashing) may have to be performed on-the-fly, as the binary is received and stored somewhere in flash before validation.

2. the execution of crucial low-level systems primitives should be secured (such as writing the new image to executable flash memory). Specific mechanisms are needed for that, on-board the device, which differ from mechanism applicable on less constrained computers -- which have a memory management unit (MMU) for instance.

5.4.2. SUIT-compliant IoT firmware updates

Content

The IETF recently formed the Software Updates for Internet of Things (SUIT) working group to standardize a secure IoT firmware update solution. The standardization process has not yet finished, but draft specifications already provide useful elements.

SUIT describes both a generic architecture for secure IoT software updates and a concise metadata format which can be used in conjunction with the firmware update binary.

5.4.2a Standard metadata for IoT firmware update

This metadata (called a*manifest*) is contained in a data structure that is protected, at least against tampering, end-to-end from the software developer, to the device itself.

The manifest contains several elements to instruct an IoT device to install only firmware that comes from an authorized source, has not been modified, is (optionally) confidentiality protected, is suitable for the hardware, and meets various other conditions.

The security end-to-end (authenticity, integrity and, if needed confidentiality) is performed using COSE (see previous sequence in this module). Low-power protocols such as COAP (or LwM2M) can then be used to deliver the manifest and the firmware image to IoT devices.

In particular, the cryptographic primitives used to assert authenticity and integrity of the received firmware update binary is a digital signature, performed on a hash of the firmware binary.

5.4.2b Manifest generation and validation

The device is assumed to be preprovisionned with the public key of the authorized software developer (or the authorized software publisher). This entity can produce a manifest indicating a hash of firmware binary and a signature of this hash. Upon receiving the manifest and the firmware update, the device can both verify the signature and compare the received hash value and the locally computed hash value. If they match and the signature is valid, the received firmware binary is considered legitimate (see workflow shown Figure 1.)

Several SUIT configurations are possible e.g. using different types of digital signatures and hash functions. For instance, one possible configuration is using ed25519 digital signatures, and SHA256 for the hash function.

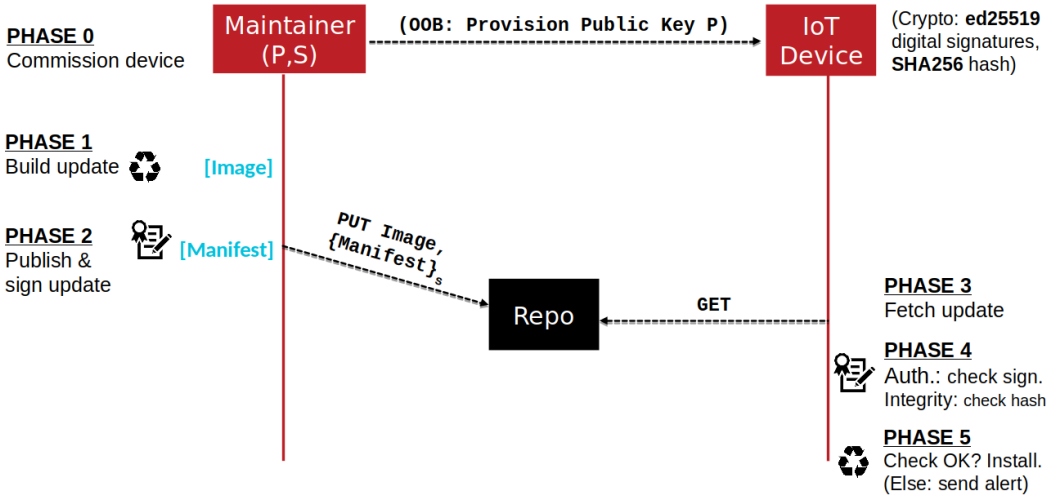


Fig. 1: Phases of a SUIT update workflow, using ed25519 digital signatures and SHA256 hash.