Universidade do Minho
Escola de Engenharia

Ângelo André Oliveira Ribeiro

Deploying RIOT Operating System on a Reconfigurable Internet of Things End-device

Guimarães, Outubro de 2017

Universidade do Minho
Escola de Engenharia

Ângelo André Oliveira Ribeiro

Deploying RIOT Operating System on a Reconfigurable Internet of Things End-device

Dissertação de Mestrado
Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Tiago Gomes

Guimarães, Outubro de 2017

Acknowledgments

First of all, I express my gratitude to the best advisor Dr. Tiago Gomes, for his friendship, patience, priceless motivation and for the huge transmission of knowledge in such an adventure through IoT. I also wish to express my gratitude to the most challenging, motivational and inspiring professors Dr. Adriano Tavares and Dr. Jorge Cabral that were fundamental in my development of knowledge in the embedded systems field.

To all the companions in this journey Hugo Araujo, Ailton Lopes, Sérgio Pereira, Nuno Silva, José Silva, Pedro Machado, Pedro Ribeiro, Francisco Petrucci and Ricardo Roriz I express my deepest gratitude for all the friendship, craziness and company on those long days and nights of work. I present also my gratitude to Cristiano Rodrigues and Ivo Marques for the collaboration and help in the development of the XIoT.

A special thank you to Guilherme Raimundo, Barbara Rodrigues, João Torre and Tiago Ribeiro.

To my family, specially to my mother, for all the support during these course, for never let me give-up, for making the impossible possible, and for sure, if I wright this words is thanks to you. Last but not least to my grandmother for such an example of perseverance and love.

To my grandfather Chico, and to you Astronauta.

"The present is theirs; the future, for which I really worked, is mine."

—Nikola Tesla

Abstract

The Internet of Everything (IoE) is enabling the connection of an infinity of physical objects to the Internet, and has the potential to connect every single existing object in the world. This empowers a market with endless opportunities where the big players are forecasting, by 2020, more than 50 billion connected devices, representing an 8 trillion USD market.

The IoE is a broad concept that comprises several technological areas and will certainly, include more in the future. Some of those already existing fields are the Internet of Energy related with the connectivity of electrical power grids, Internet of Medical Things (IoMT), for instance, enables patient monitoring, Internet of Industrial Things (IoIT), which is dedicated to industrial plants, and the Internet of Things (IoT) that focus on the connection of everyday objects (e.g. home appliances, wearables, transports, buildings, etc.) to the Internet.

The diversity of scenarios where IoT can be deployed, and consequently the different constraints associated to each device, leads to a heterogeneous network composed by several communication technologies and protocols co-existing on the same physical space. Therefore, the key requirements of an IoT network are the connectivity and the interoperability between devices. Such requirement is achieved by the adoption of standard protocols and a well-defined lightweight network stack. Due to the adoption of a standard network stack, the data processed and transmitted between devices tends to increase. Because most of the devices connected are resource constrained, i.e., low memory, low processing capabilities, available energy, the communication can severally decrease the device's performance.

Hereupon, to tackle such issues without sacrificing other important requirements, this dissertation aims to deploy an operating system (OS) for IoT, the RIOT-OS, while providing a study on how network-related tasks can benefit from hardware accelerators (deployed on reconfigurable technology), specially designed to process and filter packets received by an IoT device.

Resumo

O conceito Internet of Everything (IoE) permite a conexão de uma infinidade de objetos à Internet e tem o potencial de conectar todos os objetos existentes no mundo. Favorecendo assim o aparecimento de novos mercados e infinitas possibilidades, em que os grandes intervenientes destes mercados preveem até 2020 a conexão de mais de 50 mil milhões de dispositivos, representando um mercado de 8 mil milhões de dólares.

IoE é um amplo conceito que inclui várias áreas tecnológicas e irá certamente incluir mais no futuro. Algumas das áreas já existentes são: a Internet of Energy relacionada com a conexão de redes de transporte e distribuição de energia à Internet; Internet of Medical Things (IoMT), que possibilita a monotorização de pacientes; Internet of Industrial Things (IoIT), dedicada a instalações industriais e a Internet of Things (IoT), que foca na conexão de objetos do dia-a-dia (e.g. eletrodomésticos, *wearables*, transportes, edifícios, etc.) à Internet.

A diversidade de cenários à qual IoT pode ser aplicado, e consequentemente, as diferentes restrições aplicadas a cada dispositivo, levam à criação de uma rede heterogénea composto por diversas tecnologias de comunicação e protocolos a coexistir no mesmo espaço físico. Desta forma, os requisitos chave aplicados às redes IoT são a conectividade e interoperabilidade entre dispositivos. Estes requisitos são atingidos com a adoção de protocolos *standard* e pilhas de comunicação bem definidas. Com a adoção de pilhas de comunicação *standard*, a informação processada e transmitida entre dispostos tende a aumentar. Visto que a maioria dos dispositivos conectados possuem escaços recursos, i.e., memória reduzida, baixa capacidade de processamento, pouca energia disponível, o aumento da capacidade de comunicação pode degradar o desempenho destes dispositivos.

Posto isto, para lidar com estes problemas e sem sacrificar outros requisitos importantes, esta dissertação pretende fazer o porting de um sistema operativo IoT, o RIOT, para uma solução reconfigurável, o CUTE mote. O principal objetivo consiste na realização de um estudo sobre os benefícios que as tarefas relacionadas com as camadas de rede podem ter ao serem executadas em hardware via aceleradores dedicados. Estes aceleradores são especialmente projetados para processar e filtrar pacotes de dados provenientes de uma interface radio em redes IoT periféricas.

Contents

List of Figures

List of Tables

Listings

Acronyms

6LoWPAN IPv6 over low-power wireless personal area networks

ADC analog to digital converter

AES advanced encryption standard

AI artificial intelligence

AMBA Advanced Microcontroller Bus Architecture

APB3 Advanced Peripheral Bus v3

API application programming interface

ARQ Automatic Repeat Request

ASIC application-specific integrated circuit

BLA Black List Address

BLIP Berkeley low-power Internet stack

BSP board support package

CCA Clear Channel Assessment

CoAP constrained application protocol

COTS commercial of-the-shelf

CPS cyber-physical system

CPU central processing unit

CRC cyclic redundancy check

DFD duplicate frame detector

DFF D flip-flop

DPA	differential power analysis
DPM	dynamic power management
DSP	digital signal processor
DVFS	Dynamic Voltage and Frequency Scaling
ECC	elliptic curve cryptography
EDF	earliest deadline first
EM	evaluation module
eNVM	embedded non-volatile memory
ERCC	error correction codes
eSRAM	embedded static random-access memory
ETM	embedded trace macrocell
FEC	forward error coding
FIFO	first-in-first-out
FPGA	field-programmable gate array
FPSoC	Field-Programmable System-On-Chip
FPU	floating-point unit
FSM	finite state machine
GPIO	general-purpose input/output
GPOS	general propose operating system
GPU	graphics processing unit
HAL	hardware abstraction layer
HDL	hardware description language
HTTP	hypertext transfer protocol
I/O	input/output

I2C inter-integrated circuit

IC integrated circuit

ICMP Internet control message protocol

ICMPv6 Internet control message protocol version 6

ICN Information Centric Networking

IDE integrated development environment

IETF Internet engineering task force

ILA integrated logic analyser

IoE Internet of Everything

IoIT Internet of Industrial Things

IoMT Internet of Medical Things

IoT Internet of Things

IoT-ARM Internet of Things - Architectural Reference Model

IP Internet protocol

IPC interprocess communication

IPHC Internet protocol header compression

IPv4 Internet protocol version 4

IPv6 Internet protocol version 6

IRQ interrupt request

ISR interrupt service routine

LAN local area network

LE logic elements

LED light-emitting diode

LLN low-power and lossy network

LPDDR low-power double data rate

LUT lookup table

M2M machine-to-machine

MAC medium access control

MCU microcontroller unit

MEMS microelectromechanical systems

MLA MAC Layer Accelerator

MMU memory management unit

MPDU Mac Protocol Data Unit

MQTT Message Queuing Telemetry Transport

MSS microcontroller subsystem

NAT network address translation

NPF Network Packet Filtering

NR Network Router

NRE non-recurring engineering

OS operating system

PAN personal area network

PHY physical layer

PSR packet sending rate

PUF physically unclonable function

RAM random-access memory

RCU reconfigurable computing unit

RF radio frequency

RFID radio-frequency identification

RFSoC	Radio-Frequency System-on-a-Chip
RNG	random number generator
ROM	read-only memory
RPL	IPv6 Routing Protocol for Low-Power and Lossy Networks
RTC	real-time clock
RTL	register-transfer level
RTOS	real-time operating system
SerDes	fullduplex Serializer/Deserializer
SEU	single event upset
SHA	secure hash algorithms
SoC	system on chip
SPI	serial peripheral interface
SPPS	Secure Production Programming Solution
SRAM	static random-access memory
TCB	task control block
TCP	transmission control protocol
TEE	trusted execution environment
TI	Texas Instruments
UART	universal asynchronous receiver-transmitter
UDP	user datagram protocol
uIP	microIP
WLA	White List Address
WLAN	wireless local area network
WSN	wireless sensor network

WuR wake-up radio

XMPP Extensible Messaging and Presence Protocol

Chapter 1

Introduction

"The day science begins to study non-physical phenomena, it will make more progress in one decade than in all the previous centuries of its existence."

—Nikola Tesla

The Internet of Things (IoT) is a new paradigm were the surrounding objects are part of the Internet. This new paradigm allows to sense and actuate on the physical world with ubiquitous and transversal wireless motes, mainly low-end IoT devices that reside in the network edge. Traditionally, the motes were aggregated in a wireless sensor network (WSN) composed by homogeneous devices that could even be disconnected from the Internet. With the rise of IoT, a variety of heterogeneous devices have flooded the network, creating the necessity for connectivity and interoperability, together with processing features capable of handle the ever-growing amount of data transmitted over the network. However, due to the scarce resources and common energy-efficiency constraints of the low-end IoT devices, the accomplishment of connectivity and interoperability are not straight forward, requiring new solutions at the architectural level of the motes.

This first Chapter introduces the content of this dissertation, beginning with an introduction to the Internet of Things in Section 1.1, and followed by a description of the low-end IoT devices at the network edge in Section 1.2. A market analysis is presented in Section 1.3 based on the forecasts of the electronic and communication industries. Section 1.4 presents a general architecture to IoT regarding the flow of information and the embedded system in IoT. In Section 1.5 the two main requirements for IoT low-end devices, connectivity and security are explained. Section 1.6 discussed the shift from WSN to IoT and presents a standard network stack for IoT. Considering the increase complexity on the network edge, Section 1.7 presents the requirements for an IoT-enabled OS suitable for low-end devices. The dissertation structure is explained Section 1.8 and Section 1.9 concludes the Chapter.

1.1 The Internet of Things

The concept of Internet of Things (IoT) was first introduced by Kevin Ashton in 1999 [1], and later clarified in [2]. It consists of a major technological revolution that has updated the current Internet infrastructure to a much more advanced computing network. Furthermore, all the physical objects around us will be uniquely identifiable and ubiquitously connected to each other.

The IoT achieved such a magnitude that is now incorporated in a larger concept, the Internet of Everything (IoE). The applications of IoT not only include home appliances, wearables and everyday objects, but can be expanded to a countless number of applications. IoE can be used in cities, enabling monitoring of road traffic, trash, pollution, etc., empowering the concept of smart cities [3]. Smart buildings are another application of the IoE, allowing the monitoring and control of lights, air conditioning, etc. [4]. On the automotive industry, IoE empower not only the existence of connected cars but also the existence of full autonomous vehicles [5]. Other applications of IoE include the monitoring in real time of water quality [6], improvement of supply chains [7], food traceability [8], agriculture [9], and much more.

Nowadays, IoT can be considered a dynamic global network infrastructure that extends the classic Internet. It is composed by myriads of connected devices that can interact and communicate between them, extract information and react to the physical world. Most of the devices that compose this infrastructure reside in the network edge and are the so called "things". They can be defined as physical or virtual objects that have an identity, attributes, and are endowed with communication interfaces. The "things" can acquire and propagate information in the network, communicate with each other, and may have sensors and actuators to interact with the physical world. Some of these "things" may be endowed with intelligence by means of artificial intelligence (AI), allowing them to adapt to the environment and learn from other "things".

The most important activity of the "things" in the network edge is to collect valuable data. In fact, it is only economically viable to turn an object into a "thing", when the value of the data generated is higher than the cost of endowing the object with the needed functionalities. The data value is increasing in today's world, because with reliable data is possible to improve business models, supply chains, transportation systems, etc. In the past data was acquired manually by people, although, they have limited attention and accuracy making this data less reliable. Therefore, empowering objects and machines with computers capable of collect reliable information and make this data available over the Internet, allowing to see the world as never before and empower endless businesses and services.

The connection of physical objects to the Internet requires the incorporation of suitable computational systems, able to empower such objects with the appropriate processing and communication capabilities. These computational systems fall in the category of low-end embedded systems, mostly tiny small energy-efficient and wireless devices empowered by low-end microprocessors with scarce resources. Even extremely constrained in terms of computing power, available memory, communication, and energy capacities, it is expected that they fulfil the requirements of a cyber-physical system (CPS): (i) reliability, (ii) real-time behaviour, and (iii) an adaptive communication stack to integrate the Internet seamlessly [10].

1.2 Low-end Devices at the Network Edge

The typical embedded system that enables the connection of objects to the Internet can be designated as low-end devices. Due to the increasing value of data and due to the characteristics of the low-end devices (small fabric, low-cost and energy efficiency) they are spreading rapidly in this new Internet era. The emergence of devices with these features can be explained resorting to Moore's and Koomey's laws. The Moore's law specify that the transistors number will double every two years on an integrated circuit (IC) [11], which enables the integration of more transistors in the same space and consequently, the existence of more powerful devices occupying less silicon area. Hence, due to the high cost of silicon the price per transistor will also decrease, allowing the existence of an IC with the same performance in a smaller fabric form and lower price.

Related to energy consumption, Koomey's law explicit that the number of computational operations per joule dissipated in an IC is been doubling every 1.57 years, this means that with a fixed computing load, the amount of energy needed will decrease by a factor of one and a half [12]. Usually, low-end devices are deployed in scenarios with limited access and due to the existing number of devices maintenance should be avoided by its difficulty and cost associated. Wherefore, to avoid maintenance, energy-efficiency is required in wireless motes, which commonly run their entire lives in a single charge of a storage device (e.g. batteries, micro-fuel cells, capacitors, etc.) or resorting to harvesting solutions (e.g. solar, flow systems, thermal, etc.). The reduction of power consumption at the IC level has making possible the deployment of myriads of devices, with low maintenance cost and using tiny amounts of energy.

Due to the importance of the "things" on the IoT, the hardware that endows the object with the processing and communication capabilities is a preponderant choice when designing an IoT ecosystem. The constraints of the scenario where the system is deployed largely impact the selection of the hardware platform that can be made, mainly, from three different groups:

- **commercial of-the-shelf (COTS)**: Already-made devices that integrate a central processing unit (CPU) usually equipped with several peripherals such as timers, analog to digital converter (ADC), external memories and even communication interfaces. A COTS solution speeds up the development cycle and requires less non-recurring engineering (NRE) effort. By the other side traditionally COTS devices are not customizable, have lack of extensibility, and usually brings unnecessary hardware to the application, increasing cost, area, and power consumption;
- **Custom Platform**: Devices that target a particular application, designed and built from the scratch (e.g. application-specific integrated circuit (ASIC)). This solution can achieve the highest performance, and can be built to optimize a desired constraint. More suitable for production in high scales, or when the price per unit is a constraint due to the highest NRE cost in comparison with the other solutions;
- **Hybrid Platform**: Conjugates the best of the two worlds, providing a built in system and allowing flexibility to the final solution. The NRE cost depends on the level of customization required. These platforms are of special interest to extend an already developed system, implemented in a COTS. Platforms that combine a microcontroller unit (MCU) and field-programmable gate array (FPGA) technology on the same system on chip (SoC) (known as Field-Programmable System-On-Chip (FPSoC) [13]), allows the usage of mature software and hardware, integrating easily new hardware peripherals with minimum efforts. The FPGA technology has gained special attention lately, because they turned more cost-effective, with a smaller footprint, better power consumption efficiency, and higher performance.

1.3 IoT Market

The emergence of new business models and applications has been increasing the potential growing of IoT. Subsequently, this market captivated the attention of big players in the industry, such as Arm, Intel, Qualcomm, Microsoft, IBM, and many others. In the last years from companies to universities, multiple entities have concentrated their efforts in develop solutions for this emerging concept, ranging from the edge devices to the cloud, developing new hardware, protocols and services.

Back in 2009, Cisco presented the IoE, a vast network where all the physical things are connected, Cisco anticipated that this network would potentially rise the number of connected devices to 50 billion, creating a USD 14.4 trillion in Value at Stake [14]. Ericson back to that time, supported the forecast of Cisco presenting the same value of 50 billion connections by 2020 [15]. Later in 20015, Cisco reviewed that the global Internet protocol (IP) networks will increase from 16.3 billion in 2015 to 26.3 billion by 2020 [16]. In 2017 Ericson presented a new forecast, proposing that by 2023 there will be 30 billion connected devices, with 20 billion related to IoT [17]. Furthermore, Ericson with DHL published a report [18] in 2015, stating that IoT by itself, will generate USD 8 trillion worldwide in Value at Stake over the next decade (Figure 1.1).

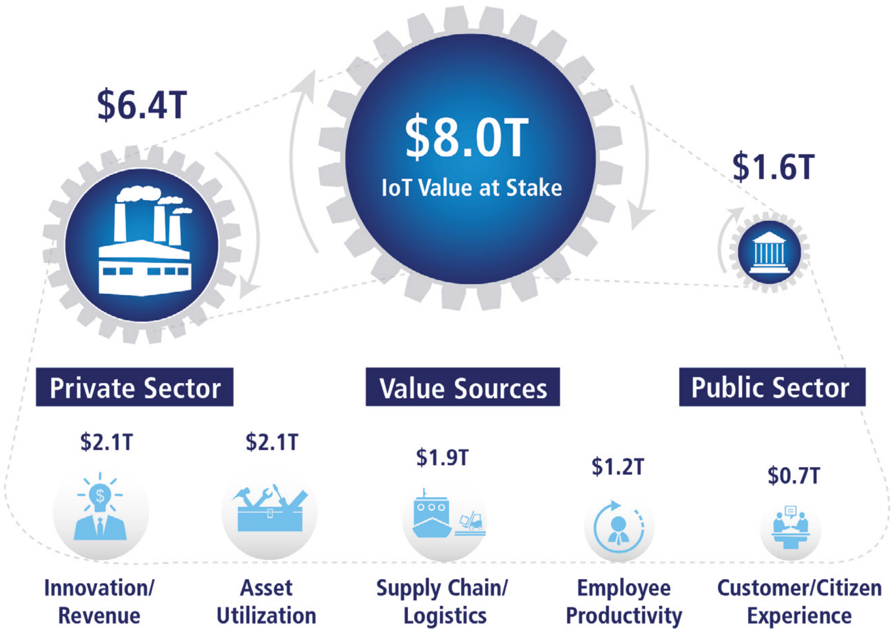

Figure 1.1: IoT Value at Stake [18].

With a more optimistic forecast, Intel predicts 200 billion devices in the near year of 2020. They expect a fast-growing pace in the IoT world, rising in just five years from 15 billion in 2015 to 200 billion in 2020, pointing out that in 2025 the total worth of IoT technology will reach the USD 6.2 trillion [19].

IHS Markit, a recognized financial services company, envisioned in 2017 that more than 31 billion devices would be connected in 2018. Furthermore, stated that while the local area network (LAN) and personal area network (PAN) solutions present de facto standards and market stability, the new IoT market presents duplicated and overlapping wireless solutions. Therefore, this fragmented market, leads to IoT vendors increase their shipments in over 20% year-on-year [20].

Futurum, a research and analyst firm, stated in 2017 that the global number of connected devices and objects in operation in 2020 will be between 40 and 50 billion. As a point of reference, they illustrate that Qualcomm, alone, ships roughly a million chips for IoT every single day. Furthermore, their estimations envision a spending in IoT of USD 250 billion in 2020, with 40-45% of that spent in products, and 55-60% in services [21].

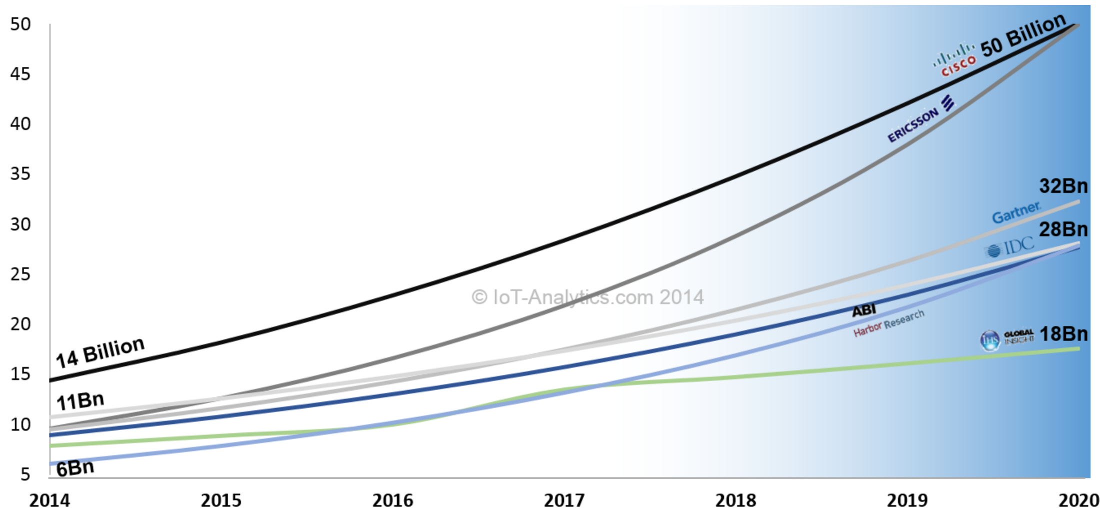

Figure 1.2: IoT Market forecast: Connected devices 2014-2020 [22].

In 2017, Arm stated in a white paper [23] their forecast to the IoT world by 2035. They estimate a production of 1 trillion devices between 2017 and 2035, reaching in 2035 a USD trillion per annum of spends in IoT hardware and services.

The market predictions diverge largely from entity to entity as seen in Figure 1.2. However, what could be extracted from the different envisions is that for every person living on earth, there will be at least 2, maybe even 6 connected things by 2020. The IoT arena will create a revenue opportunity for companies beyond what Apple, Google, and Facebook are selling together today, surpassing the economic output of Germany within the next 10 years [22].

1.4 IoT Architecture

The IoT presents endless opportunities, consequently it has been applied to diverse scenarios such as smart homes, wearables, smart cities, smart grids, automotive industry, agriculture, retail, to name a few [24]. The different scenarios present unique characteristics that constrain the used technologies, network architectures and design approaches.

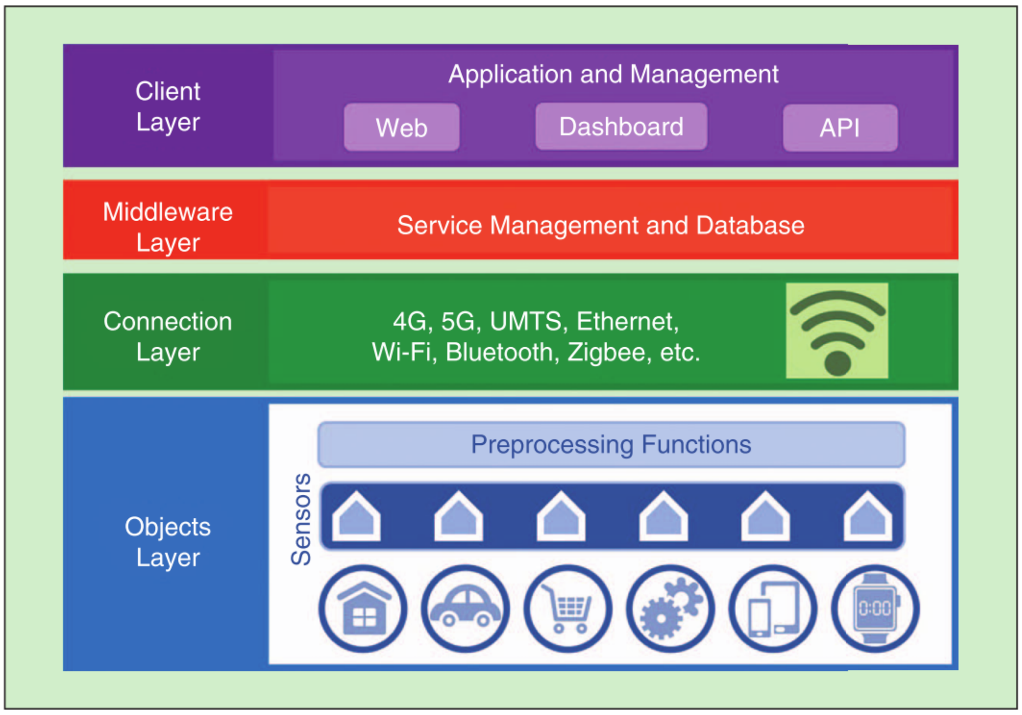
Client Layer
Application and Management
Web
Dashboard
API
Middleware Layer
Service Management and Database
Connection Layer
4G, 5G, UMTS, Ethernet, Wi-Fi, Bluetooth, Zigbee, etc.
Objects Layer
Preprocessing Functions
Sensors
0:00

Figure 1.3: The general architecture of the IoT [25].

A general architecture for IoT is shown in Figure 1.3, which consists of a four-layer vertical architecture with bottom-up data flow [26–28]. The layers that compose this stack can be described as: (i) Objects layer, where the physical items reside, they are identifiable and able to acquire data generated by them or by their surroundings. These objects are identifiable with resources such as radio-frequency identification (RFID) tags, and with ability of auto-identification trough medium access control (MAC) addresses or an ID stored in a non-volatile memory; (ii) Connection layer, that connects the physical objects with the middleware layer. The connection resources used are influenced by the communication systems and protocols used, the network topology, the type of data acquisition systems, and the communication capabilities of the devices; (iii) Middleware layer, stores the information acquired in databases, related with the services provided by each physical object. Data mining functions are applied on this layer and decisions made in accordance. Due the big amount of data generated, some IoT applications use proper firmware to pre-process and filter data before streaming; (iv) Client layer, corresponds to a specific target application. Generates the proper results and actuation in accordance with the middleware layer information. This layer is also responsible for the connection of the object to the environment, users or provide machine-to-machine (M2M) communication. It is on this layer that the IoT is able to create added value, trough providing valuable information, improving safety, mange supply chains and many more depending on each application scenario.

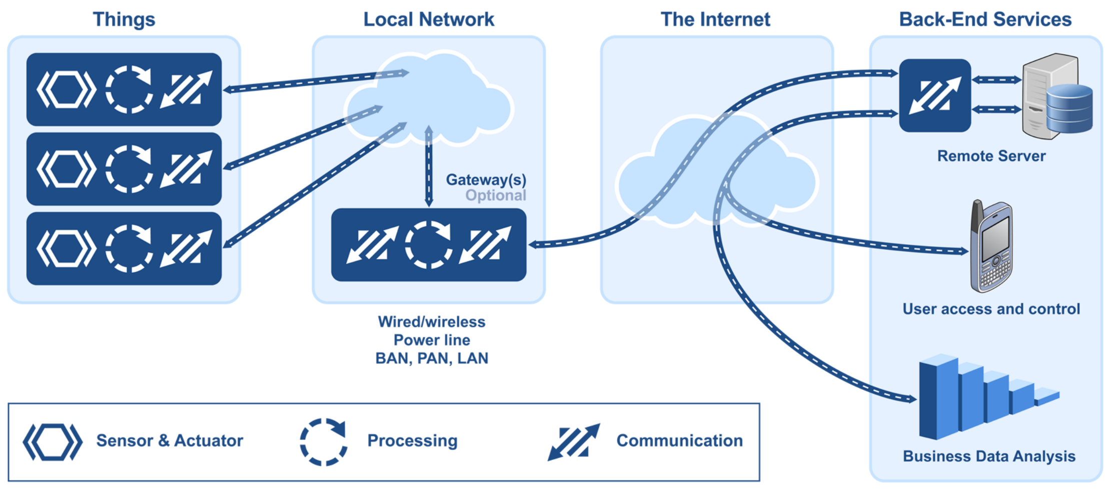

Figure 1.4: The IoT from an embedded systems point of view [29].

The work of this dissertation will focus on the embedded system device that empower IoT at the network edge. Figure 1.4 shows a common topology for IoT from an embedded system point of view. It is composed by four main components: (i) the "thing", an embedded device endowed with processing capabilities, that is able to communicate, sense the physical environment and/or actuate; (ii) the Local Network, that can include a gateway, responsible by perform connectivity aggregation trough collection of data-in-transit, processing, verification and further retransmission to the Internet; (iii) the Internet; (iv) the Back-End Services that comprehend enterprise data systems (Cloud) or private repositories, responsible for saving data for further processing and making it available to authorized users [29].

1.5 Connectivity and Security

The IoT is not an interface by itself, it is more a set of frameworks connected in the vast Internet. Even more, the endpoints are built in embedded devices that are getting cheaper, smaller and by its nature run applications with strong communication requirements. The prevalence of such a big network of interconnected devices will impose new security and privacy threats putting all those devices at a high risk [30]. For these reasons, strong communication and security constraints are applied to these devices.

The communication requirements for the IoT endpoints devices include:

- **Connectivity**: Possibility of all the endpoint devices to be connected to the network. The transition from Internet protocol version 4 (IPv4) to Internet protocol version 6 (IPv6) allows the connection of 2^{128} devices with a unique IP address. The usage of IPv6 augments the security over IPv4 because the first does not use network address translation (NAT), avoiding unnecessary addresses translation and packet modification;

- **Interoperability**: Ability of exchange and make use of information regardless the used technology. To achieve interoperability is needed the (i) incorporation of multimode radios that allow diverse IoT devices talk to with each other; (ii) software flexibility that enables the support for different standard communication protocols; (iii) strong hardware-based security, as described later;

- **Reachability**: Availability of a node over time on the network. The IoT nodes will be full reachable, with a wide deployment of IPv6, thus allowing the unequivocal identification of the endpoints on the network.

To ensure proper security of the endpoint devices is needed to secure not only the device, trough hardware security, but also the data contained and exchanged by it. The security requirements for the endpoint devices are described as follows:

- **Data Security**: the security triad or CIA triad stands for confidentiality, integrity and availability. It constitutes the root of trust, in data security for the IoT devices, and is summarized below:
  - Confidentiality: Ability to provide confidence on user about information privacy. It is done by preventing unauthorized people to access information. The data privacy can be guaranteed by, for example, encryption and two-step verification;
  - Integrity: Guaranty that data will remain unchanged during transmission and reception. Perhaps, if data is modified it must be noticed. Commonly used methods to verify data integrity are checksum and cyclic redundancy check (CRC);

  - Availability: Capability to give immediate access to data, to authorized parties, in any condition. It guarantees that the information will be properly saved and accessible. Two methods to guaranty availability are redundancy and failover backup.

- **Hardware Security**: The cornerstone of embedded system security is to guarantee that only authorized microprocessor code is loaded and executed, consequently, a hardware root of trust is required. Furthermore, a hardware root of trust should provide secure monitoring, secure validation/authentication, storage protection, secure communication and key management. A hardware root of trust can be defined by four basic blocks:

  - Protective hardware: Provides a trusted execution environment (TEE) [31, 32] to run only privileged software. This is achieved through the implementation of secure measures on the silicon manufacturing phase, with the implementation of unique security keys. Protection to the runtime memory should be implemented to protect the STACK, HEAP and global data;
  - Tamper detection and countermeasures: Code loaded from the outside is validated before running on the secure CPU. When a validation fails a tamper is detected triggering the countermeasures, that may include device zeroization, used to permanently erase sensitive data, such as cryptographic keys, device lock, among others;
  - Crypto-engines: Dedicated hardware cryptographic accelerators to allow data and communication channels protection, without overload the CPU.

Notwithstanding security being a major concern in IoT, it is let out of the scope of this dissertation. Therefore, this work will only focus on the connectivity of low-end IoT devices.

1.6 IoT Network Stack

The IoT is the expansion of the current Internet services, so it as to accommodate each and every object which exists in this world or likely to exist in the coming future [33]. Although, today's Internet is dependent on IPv4 that is based on a 32-bit addresses, and therefore able to generate only 4,294,967,296 unique addresses. The IPv4 address exhaustion occurred in Feb.3 2011, however it had been delayed using remapping methods such as NAT [34]. Considered this limitation, IPv6 was developed aiming to have enough addresses to scale the Internet for decades to come, which according the estimates may include 50 billion connected devices by 2020 [34]. IPv6 uses a 128-bit address that empower the generation of 3.4×10^{38} unique addresses, opening the Internet to a new era and allowing the connection of countless devices [34, 35].

WSNs at the IoT edge, are mainly composed by oodles of inexpensive low-power devices, with small computing capabilities, scarce memory, and mainly relying on IEEE 802.15.4-based networks. Essentially this network supports low bandwidths, small packet sizes and typically suffers from severe and frequent packet loss [36, 37]. In order to endow such devices with IPv6 connectivity, the network stack deployed in IoT require an adaption layer to deal with the constraints of the IEEE 802.15.4 MAC/physical layer (PHY) and thus achieve IPv6 compliance. The IPv6 over low-power wireless personal area networks (6LoWPAN) deals with such limitations, through the introduction of mechanisms of header compression, fragmentation, reassembly, and stateless auto configuration as well as implementing modifications to the neighbour discovery, reducing bootstrapping complexity [38, 39]. This adaptation layer allows the big shift from traditional WSN to IoT, thus, enables the usage of IPv6 in such constrained devices, and due to its interoperability and lightweight implementation is being while deployed on constrained embedded systems [40].

Figure 1.5 presents the network stack proposed by the Internet of Things - Architectural Reference Model (IoT-ARM) [41], which is a standard stack that aims to increase the connectivity and the interoperability among IoT devices [42].

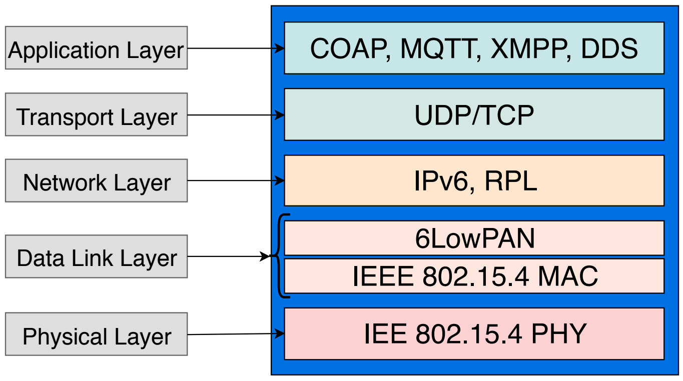

Figure 1.5: 5-layer IoT-ARM Communication Model [42].

Other protocols could be adopted to the network stack, however, the presented in Figure 1.5 are recommended to be used on low-end devices endowed with a low-power radio interface.

1. **Physical Layer**: Is defined on IEEE 802.15.4 standard and suits the needs of the target devices.

2. **Data link layer**: The MAC layer provides access to the media and is part of IEEE 802.15.4 protocol. In this layer resides also the 6LoWPAN adaptation layer allowing the usage of IPv6 in resource constrained devices.

3. **Network layer**: Address and routes data through the network, traditionally enabled by IPv4, and recently upgraded to support IPv6. The IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [43] is a routing protocol, that improves the route of data between autonomous devices in the Internet.

4. **Transport layer**: The user datagram protocol (UDP) and transmission control protocol (TCP) are the transport protocols used in the Internet. Often, TCP is considered too complex for low-power and lossy network (LLN), thus, not suitable for devices with few resources, and demanding for low power consumption. UDP is the most suitable and deployed on IoT scenarios.

5. **Application layer**: The hypertext transfer protocol (HTTP) that is used on the Internet is too complex for constrained nodes, thus alternative mechanisms have been developed for LLN. One of the most important mechanisms is constrained application protocol (CoAP), a web transfer protocol targeting M2M applications, and capable to interoperate with HTTP. CoAP runs over UDP while Extensible Messaging and Presence Protocol (XMPP) and Message Queuing Telemetry Transport (MQTT) run-over TCP.

1.7 Operating Systems for Low-end IoT Devices

The complexity of the IoT arena is rising rapidly and the amount of data and protocols that need to be managed is pushing IoT low-end devices out of them comfort zone, thus the integration of a full-fledged OS is required. A general propose operating system (GPOS) such as Linux is so far considered too complex for IoT low-end devices [10]. Besides, the usage of lightweight OS dedicated to WSN in more powerful IoT devices result in less energy-efficient implementations and do not exploit all the devices capabilities [10].

Furthermore, it is desirable that the OS provides capabilities of a modern full-fledged OS, such as native multi-threading, hardware abstraction, dynamic memory management, a friendly application programming interface (API) and at least C or C++ programming languages. Portability and interoperability are important at the OS level, thus open standards and APIs such as POSIX should be adopted. A complete and comprehensible documentation is required to make the best use of the OS and ease application design. Another important point when choosing an OS for IoT is the maturity of the code, widely deployed and tested code with commercial applications and a large community of contributors are more unlikely to have undetected errors, thus open source code should be always a first choice [44].

1.7.1 Requirements for an IoT OS

Due to the characteristics of the IoT low-end devices, an OS suitable for this field of embedded systems must fulfil several requirements and provide modularity and configurability, to be possible his deployment in different hardware platforms and applications. The widespread requirements are:

- **Small Memory Footprint**: Low-end devices exhibit scarce memory resources, usually a few kilobytes of random-access memory (RAM) and read-only memory (ROM) [45]. Consequently, the OS should have a small footprint maintaining an acceptable performance and convenient API;
- **Support for Heterogeneous Hardware**: The OS suitable for the IoT need to be able to deal with a heterogeneity of hardware architectures, is desirable to support various MCUs architectures and families from 8 bit to 32-bit architectures, enabling support to devices with different capabilities such as memory management unit (MMU) or floating-point unit (FPU);
- **Energy Efficiency**: Due to the power constraints of IoT low-end devices, the OS should make use of the power saving modes available on the MCUs, resorting to efficient schedulers policies to switch to sleep modes as soon as possible;
- **Real-Time Capabilities**: Various applications require precise timing and timely execution, thus the OS should guarantee worst-case execution times and worst-case interrupt latencies;
- **Network Connectivity**: The need to support multiple link layer technologies and provide communication with other Internet hosts, led to the use of network stacks based on IP protocols directly on IoT devices [46]. Thus, a key requirement for an IoT-OS is offer support for heterogeneous

link layer technologies, such as low-power radio and wired technologies. Furthermore, should provide a modular network stack based on standard (open standards are mandatory, such as those specified by Internet engineering task force (IETF)) IP protocols relevant for IoT [46] (a full-fledged TCP/IP implementation as well as a 6LoWPAN stack) that support evolution and integration of multiple network protocols;

- **Security**: IoT devices can be deployed in critical or industrial infrastructures with life safety implications (in this case certification of all the application including the OS, may be mandatory), in other hand the communication capabilities of this devices entail dangerous, so it should meet high security and privacy standards. Often deployed in critical applications with difficult physical access and high failure cost, these systems must be robust and thus the OS reliable. From a security data point of view the IoT OS should fulfil the CIA triad. Secure mechanisms for software updates must be incorporated and open source used as much as possible [47].

1.8 Dissertation Structure

The remaining of this dissertation is structured as follows:

- Chapter 2 presents an analysis of prominent OS and platforms that enable the development of this dissertation. Also, analyses the evaluation tools that allow the development of such a work and presents a literature review of relevant state-of-art in the development of heterogeneous solutions for IoT devices.

- Chapter 3 analysis heterogeneous architectures for low-end devices with special focus on an in-house project the CUTE mote. The structure of RIOT-OS is presented in this chapter and described the deployment of such OS on a COTS platform with the usage of IAR Workbench. Furthermore, is presented the heterogeneous architecture developed and the deployment of RIOT-OS in this platform. The chapter concludes with the evaluation of the COTS and heterogeneous solution.

- Chapter 4 presents a discussion of hardware acceleration on low-end IoT devices, followed by the analysis of the MAC Layer Accelerator (MLA) deployed in CUTE mote. Then, several improvements to the hardware accelerator are presented, emerging XIOT, a refactoring of the MLA. Furthermore, this chapter describes the integration of XIOT in RIOT-OS and finalizes with the evaluation of the architecture.

- Chapter 5 concludes the dissertation discussing the obtained results in the different platforms and analyses the advantage of hardware network-related accelerators in low-end IoT devices. For last, are presented future developments for this work.

1.9 Conclusion

This Chapter has introduced the IoT world, with focus on this dissertation core, the low-end devices at the network edge. It was analysed the bottom-up flow of information in IoT and how embedded systems play a critical role in the generation and streaming of information. The two main requirements for embedded systems in the IoT were also discussed, with special attention to the connectivity challenges. To solve those challenges was presented a network stack for the low-end devices in the IoT edge. Knowing that the complexity on the edge is increasing, the low-end IoT devices can benefit from an embedded OS, thus, the requirements for an OS suitable for these devices were discussed. Finally, was described the overall structure of this dissertation. The next Chapter presents relevant state of art and the tolls used during this dissertation.

Chapter 2

State of Art

"If I have seen further than others, it is by standing upon the shoulders of giants."

—Isaac Newton

This Chapter analyses the existing tools and hardware solutions that allow to develop the work of this dissertation, as well as the current state of art in heterogeneous architectures for low-end devices. Section 2.1 analyses the most prominent and suitable OS for low-end IoT devices in the network edge. Then, Section 2.2 describes the hardware platforms used for the development of the heterogeneous architecture and the COTS solutions used for performance comparison. Section 2.3 presents the main tools that allow the development, test and evaluation of the developed work. The state of the art of heterogeneous architectures is presented in Section 2.4, with special focus on CUTE mote project. Finally, Section 2.5 concludes this Chapter.

2.1 Operating Systems for IoT

Traditional WSN were mainly homogeneous networks with a single connection point, or in some cases even disconnected from the Internet. As seen previously in Section 1.5 and Section 1.6 of Chapter 1, the transition from WSN to IoT requires wireless motes to guarantee connectivity, interoperability and reachability. The accomplishment of these requirements can only be achieved with the integration of an IP compliant network stack in the motes. However, the incorporation of a network stack largely increases the complexity of the software deployed in the IoT devices. To manage such complexity, the integration of an OS in these devices is mandatory, however, due their scarce resources this is not straightforward.

The OSes available for IoT present a variety of features, and even that all of them guarantee in somehow the basic features to create at least a node in a WSN, not all of them fulfil the requirements presented in Subsection 1.7.1. Therefore, the design choices of the OS will impact its behaviour and applications. A summary of the design options of an OS for resource constrained (and connected) embedded device is presented below.

- **Architecture**: The design structure and modularity of the OS are determined by the adopted architecture, which may be monolithic, microkernel or layered architecture. The monolithic architecture is all compiled into a single system image, resulting in a small OS footprint with reduced overhead in the interaction between modules. Consequently, the low fragmentation and isolation of this architecture makes it difficult to maintain and more prominent to system failures, due to errors in single modules. On the microkernel approach the kernel size is shortened, providing a minimum set of functionalities inside the kernel. The interaction between user and kernel space is provided by the usage of user-level servers. By the fragmentation achieved, the reliability, customization and maintainability are improved in comparison with the monolithic architecture. However, the communication between user and kernel space introduces a degradation of performance. In the layered approach the OS is split into various layers, implementing each of them a functionality. This architecture presents less modularity that the microkernel, however, it is more flexible and reliable than the monolithic design;

- **Memory Allocation**: As described before, IoT low-end devices have scarce resources, including memory. Therefore, the strategy for memory allocation must have in consideration the available resources, and it can be either static or dynamic. The static memory allocation is a simple and a useful technique to deal with reduced memory. No memory allocation is done in run time, as drawback it results in an inflexible device. The dynamic strategy makes the system more flexible and allows a better management of the memory space. However, it can be a hard task to be performed by an MCU with reduced resources, mainly because of the absence of MMU that makes the memory management difficult, and decreases the system's security;

- **Network Buffer Management**: Regarding the specificity of an OS for IoT devices, the usage of a network stack built in a layered fashion creates the necessity of exchanging information between each layer of the stack. This can be done either by reference, or by memory copy. Some OSes present a new approach, where a central memory manager is used to handle the data between layers in a more efficient manner;

- **Driver Model and Hardware Abstraction Layer**: A clear-cut driver model enables the integration of different peripheral drivers into an application, facilitating further integration of new drivers into the OS. A well-defined hardware abstraction layer empowers the portability between different MCU families and architectures;

- **Scheduling Strategy**: IoT devices are applied to a wide range of scenarios, both with real-time and non real-time requirements. This enforces the OS to implement proper scheduling policies to fit the application requirements. The scheduler applied to IoT devices should have in consideration the energy consumption as well as the memory usage;

- **Programming Model**: Two models are dominant in the IoT arena and can be defined either as event-driven or multi-threading. On a programming model based on event-driven, the execution of a task is triggered by an event, e.g., an interrupt. This model is more suitable for devices with scarce resources, and is often implemented by the execution of an event loop, instead of a more complex scheduler and a shared-stack model [44]. In a multi-thread environment, the usage of a scheduler is imperative to manage the execution of threads. This approach enables de development of more complex systems, but requires more powerful hardware resources than the event-driven model;

- **Programming Languages and Debugging Tools**: The usage of standard programming languages such as ANSI C or C++ simplifies the portability between platforms, and enables the usage of well-established toolchains and tools (development and debugging). Besides, dialects or OS-specific languages may be used to solve performance and safety-relevant issues that low level languages such as C are not able to solve. Due to the constrained resources of the low-end devices, some of them do not offer debugging interfaces. Alternatively, it can be used the `printf()` function over a UART or even a light-emitting diode (LED) connected to a general-purpose input/output (GPIO) port to provide debug information;

- **Testing**: A crucial phase in the development of any application is the testing process, thus the OS should provide facilities to help in this process. The hardness of testing IoT devices arise from their distributed, deeply embedded and often very constrained nature. An extensively used technique to test hardware-related parts, such as device drivers, is the usage of hardware emulation tools, e.g., MSPSim or Emul8 [48]. Network emulators and simulators such as Cooja or ns-2/ns-3, that allow for the integration of OS code, are an asset for such a task [49];

- **Feature Set**: Two main groups compose the set of features provided by the OS. One comprehends the kernel functionalities, such as timers and synchronization mechanisms, and other the high-level features of the system that might include a shell or cryptographic libraries;

Some available OS solutions to deploy in an IoT device at the network edge include Contiki-OS [50], Tiny-OS [51], Linux, RIOT-OS [52], FreeRTOS [53], Amazon FreeRTOS [54], Mbed OS [55], Mantis [56], Nano-RK [57], NutOS [58], MansOS [59], MicroC/OS-III [60], μClinux [61], etc. However, most of them does not fulfil the requirements described in Section 1.7 of Chapter 1. In the remaining of this Section are described in detail the most prominent OS solutions for the low-end IoT devices.

2.1.1 Contiki-OS

Contiki-OS [50] is an OS developed around the microIP (uIP) stack, targets the resource-constrained embedded systems, and it is currently applied in fields such as WSN or IoT. Architecturally, Contiki-OS is implemented in a modular approach, that is close to the layered architecture [62]. OS facilities are provided in the form of services, having their own implementation and interface, which increases the granularity of the OS and decreases the interdependence between system modules.

The programming language adopted to develop Contiki-OS was C, a highly portable language. The supported platforms cover a wide range of resource-constrained devices, including 8-bit AVR platforms, 16-bit and 20-bit MSP430 platforms, and 32-bit Arm Cortex M3 architecture [44].

An event-driven programming model is used in the kernel space, where the event handlers run in the same context and to completion. No priority is assigned to the event handlers, however, they can use internal mechanisms for preemption [63]. In the application space, Contiki provides partial support for multithreading by implementing a lightweight form of threads, which are designated by Protothreads [64]). Those lightweight threads are stack-less and have a very small memory overhead. Besides the partial multithreading support, none sophisticated scheduling algorithm is materialized, consequently the OS does not fulfil the requirements for real-time applications. Contiki provides power saving modes, by switching to the deepest sleeping mode supported, when the OS finishes all the pending tasks.

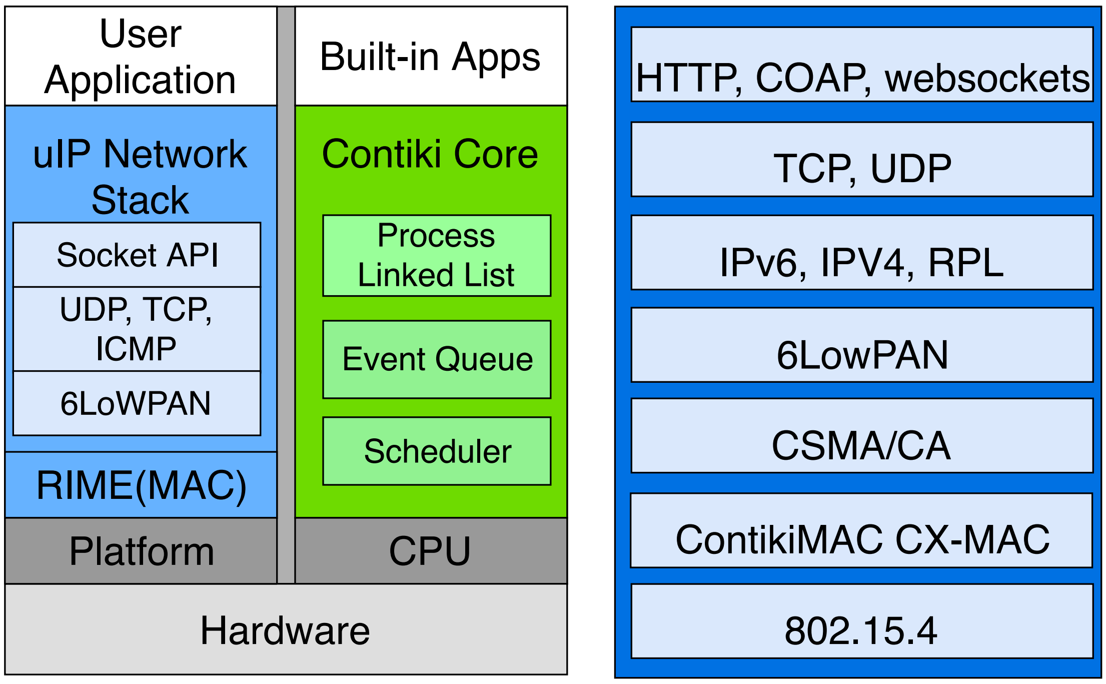

Figure 2.1: Contiki-OS architecture stack and supported IoT stack.

Contiki offers support for dynamic memory allocation. To ensure that no fragmentation happens in the memory, a memory allocator manager is used for compacting memory when there are free blocks in the system. The absence of a memory protection mechanism between different processes makes the overall system unprotected [63].

Contiki provides support to a wide range of protocols used in the IoT communication stack such as IPv6, 6LoWPAN, RPL, and CoAP [44]. An implementation of uIP [65] is provided with a minimum set of features needed to have a minimal TCP/IP stack, compliant with its full implementation on more powerful devices and OSes. This way, low-end devices can interoperate and communicate with remote servers without the need of protocol translation/adaptation in the middle. Another stack supported by Contiki is Rime [66], a lightweight stack able of provide a set of distributed programming abstraction layers. Such stack can be used when a simple and isolated network is needed by the final application. Simulation capabilities are provided in Contiki by Cooja [67], a simulation tool for WSN. Contiki enables the load and unload of individual applications or services at run-time [50], which makes possible the remote upgrade of the system.

The source code of Contiki is available under a BSD license on GitHub with a large community of contributors and forks. A rich API and documentation are available enhancing the development process.

2.1.2 TinyOS

TinyOS [51] is a component-based and application-specific OS designed for WSN, with a low OS footprint targeting very constrained platforms. The OS implements a monolithic architecture and uses a component-based model. According the specificity of the application, the components will compose a static OS image, to be deployed on the mote. A component is an entity that exposes one or more interfaces, consisting of three computational abstractions: commands, events and tasks. This OS provides a single stack and there is no isolation between kernel and user spaces.

The programming model followed by TinyOS is the event driven model, with the addition of TOSThreads [68] in version 2.1, which enabled the support for multithreading. TOSThreads, a preemptive application level thread library for TinyOS, provides the ease programming of a threading model and the efficiency of an event-driven kernel. For these threads a cooperative scheduling algorithm is used, thus the kernel relies on the threads to yield the processor. Consequently, the programmer is responsible by managing concurrency. In TinyOS each thread has to allocate a task control block (TCB) with fixed memory space.

The scheduling algorithm employed by earlier versions of TinyOS relied on a non-preemptive first-in-first-out (FIFO) scheduling algorithm, thus the tasks on this FIFO ran to completion. The task execution is not an atomic process, and it can be interrupted by an interrupt handler, command or event. To be compliant

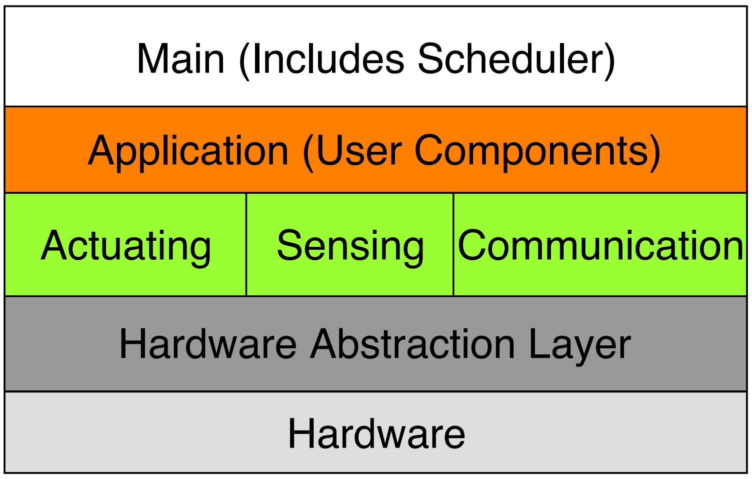
Main (Includes Scheduler)
Application (User Components)
Actuating
Sensing
Communication
Hardware Abstraction Layer
Hardware

Figure 2.2: TinyOS architecture stack and supported IoT stack.

with the requirements for real-time applications, later versions of TinyOS implement an earliest deadline first (EDF) scheduling algorithm. However, in this policy there is no guaranty that the deadlines will be meet when the system is overload, compromising the deployment in systems with strong real-time requirements.

Since low-end IoT devices are resources constrained, MMUs are not always present in every device. Thus, TinyOS presents an efficient memory safety method [69] incorporated in the latest versions. In this OS only static memory allocation is used, consequently no heaps, function pointers or virtual memory mechanisms are used.

The OS communication capabilities encompass a Berkeley low-power Internet stack (BLIP) [70], an implementation of the 6LoWPAN stack that include TCP, UDP, Internet control message protocol version 6 (ICMPv6), IPv6, RPL and CoAP. The OS also uses Hydro routing protocol [71] to ensure reliable unicast communication within an IPv6 sub network with lossy links [72].

The set of features of TinyOS comprehend a file system, database [73], security [74] and simulation support provided by TOSSIM [75]. TinyOS uses a C dialect called NesC [76], which is a component-based and event-driven programming language [44]. The source code is provided under a BSD license, with extensive documentation, tutorials and examples.

2.1.3 RIOT-OS

RIOT-OS [52] is an OS developed to fill the gap between WSN OS and full-fledged OS. Aiming to allow easy portability among a wide range of devices, the OS is implemented in two distinct parts, hardware-dependent and hardware-independent code, with well-defined interface and independence of the modules.

RIOT-OS implements a modular architecture built around a microkernel inherited from FireKernel [77]. It uses a multi-threaded programming model similar to Linux, on which each thread has a unique TCB and its own memory space. The development of the multi-thread model gives special attention to the memory usage, by designing a very small TCB and minimizing the usage of the stack during run time. The use of multi-threading on RIOT-OS is optional, thus, it can be removed and the user application be the only thread running on the system decreasing this way the memory requirements. For thread synchronization and communication RIOT-OS offers mutex, semaphore and messaging queues.

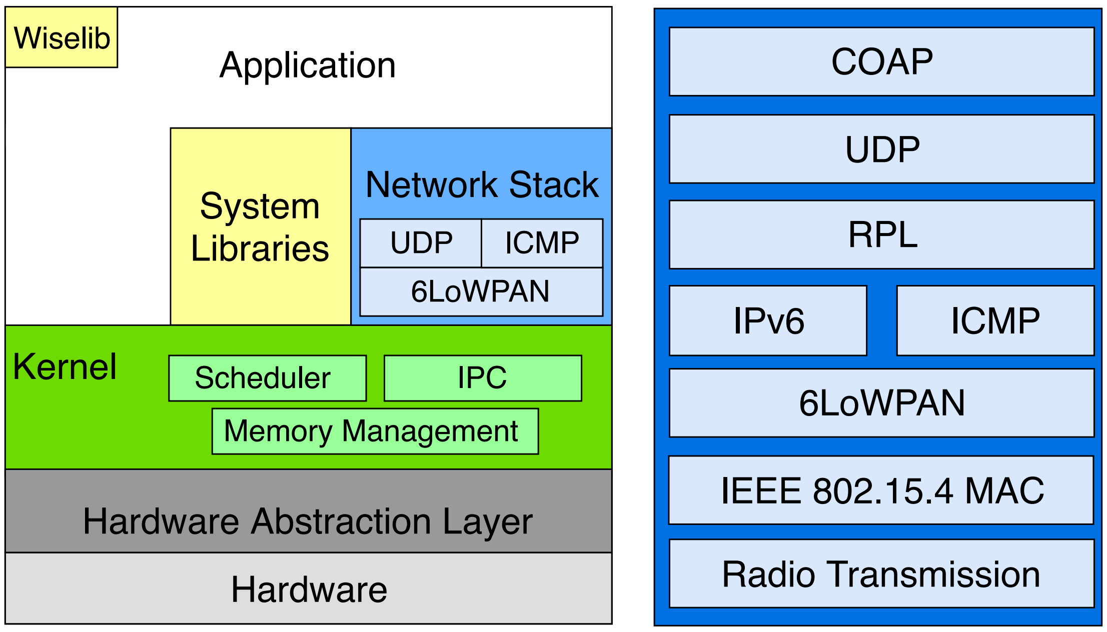

Figure 2.3: RIOT-OS architecture stack and supported IoT stack.

The scheduler algorithm implied in RIOT-OS is based on fixed priorities and preemption. The scheduler policy enforces the thread with the highest priority to run-to-completion, that can only be interrupted by an interrupt request (IRQ). In order to promote energy efficiency, RIOT-OS proposes a tickless scheduler, i.e., it works without any periodic event, aka tickless. The system enters in sleep mode after switching to the idle thread, that will determine the deepest sleep mode possible, and only external events will wake up the system. In brief, RIOT-OS presents soft real-time capabilities [52].

On this OS the use of dynamic memory allocation is only allowed on the user application space. On the kernel side, runtime of O(1) is guaranteed by the exclusive use of static memory allocation [10].

The OS is written in standard ANSI C language, with some hardware dependent parts written in assembly [44]. RIOT adds support for C++ programming language enabling this way the usage of powerful libraries. The use of standard languages allow to employ well-known toolchains and debugging tools, as well as increase portability features.

RIOT defines a flexible layer-separating architecture [78], incorporating two programming interfaces for the network subsystem, labelled as (i) netdev and (ii) sock. In (i) is offered a generic network driver interface, whereas in (ii) a high-level network interface is provided for applications access [52]. One of the implemented stacks by RIOT is GNRC, the default IP stack [78], based on the standard IP protocols, that supports 6LoWPAN, IPv6, RPL, UDP and CoAP implemented with well-defined interfaces and interprocess communication (IPC) [40]. A port of the full 6TiSCH stack is implemented in the form of OpenWSN [79] and CCN-lite is implemented as a port of the Information Centric Networking (ICN) [44]. On the default network stack, GNRC, a centralized network buffer is used to store packets, headers and other networking meta-data, allowing to transmit only the addresses of the information between the network layers, without need for data copy or duplication of data.

As described in Section 1.7, update over-the-air is an important feature, thus it can be included in RIOT with the integration of a boot-loader and two firmware slots. Through the 6LoWPAN network stack, a RIOT image (featuring a software update module) can be downloaded into the system allowing the reconfiguration. RIOT goes even further by providing a secure update with integrity and authenticity of the downloaded firmware using state of the art public-key crypto and hashing algorithms [52].

RIOT provides a hardware visualizer that allows the compilation and execution of RIOT applications on a host OS. Other hardware simulators can be used to compile and test RIOT such as MSPSim & Cooja [52], IoT-Lab [80] or DES-Testbed [81]. A wide set of features is provided by RIOT, of which stand out a powerful command-line interpreter (a Linux-like shell), a virtual file system and cryptographic libraries. The source code of RIOT is available on GitHub under a LGPL v2.1 license. A consistent documentation is provided in a standard form with the usage of Doxygen. The community around RIOT-OS is quite active through the development of code updates, bug detection and publication of scientific papers and academic works.

2.1.4 FreeRTOS

FreeRTOS [53] is a popular real-time operating system (RTOS) widely deployed in industrial and commercial applications as well as used in several research projects. A large community use this OS and it has been ported for many MCU architectures and platforms. The architecture of FreeRTOS is fairly simple, all the OS is splitted in four files written in C, usually is considered more a threading library than a full-fledged OS [44]. FreeRTOS employs a multi-threading programming model based on tasks, with each task being instantiated statically with its own stack.

The OS presents real-time features provided by the implementation of a preemptive, priority-based round-robin scheduler, and by synchronization objects such as mutexes and semaphores. Since version 7.3.0 a optional tickless mode is supported in the scheduler [44]. Dynamic memory allocation is supported by the OS in five different modes: (i) allocate only, memory cannot be freed, (ii) allocate and free, without coalescent of adjacent free blocks, (iii) wraps of the standard C library `malloc()` and `free()` for thread safety, (iv) an algorithm similar to (i), but with coalescent of adjacent free blocks to avoid fragmentation and, (v) an improvement of (iv), with the ability to scatter the heap over several memory sections.

FreeRTOS does not natively provide a network stack, , however multiple stacks from third-party can be used as a supplement to the OS. Ports of network stacks were developed to match with FreeRTOS, of which stand out the adaptation of IwIP and Nabto.

The OS does not define a driver model or abstraction layer, rather it works with vendor supplied board support package (BSP). FreeRTOS does not provide any debug or testing capabilities, instead it resides in third-party solutions, due to the portability of the OS it can easily integrated in standard tools. FreeRTOS offers rich documentation, compromising books, trainings and a broad API. The source code is provided under a GLP license.

2.2 Hardware Platforms

The development of a heterogeneous solution for low-end IoT devices, with the integration of hardware accelerators presents various challenges, as described in [36]. The common constraints for IoT devices, that reside on the network edge include low-price, power consumption, processing capabilities, memory resources, among others. Thus, the following requirements should be considered, not only in the choice of a platform, but also is desired their accomplishment in the final solution:

- Implement an **IoT-enabled network stack**, that promotes the communication requirements for an IoT device: (i) connectivity, (ii) interoperability, and (iii) reachability;
- Provide **scalability**, to allow the inclusion of new hardware accelerators;
- Provide **availability**, an important feature for sensitive applications that have the potential to cause harm on persons or goods;
- Empower design **modularity and customization** to facilitate the efficient integration of only the required components, optimizing the usage of resources and power;
- Facilitate **portability**, enabling the fast integration of a heterogeneous set of devices, including different communication interfaces and OS;
- A **secure hardware architecture**, in a commercially available low-power SoC.
- **Data protection mechanisms** that enable the accomplishment of the CIA triad requirements identified in Section 1.5.

Having in consideration the requirements previously presented and the hardware solutions described in Section 1.2 of Chapter 1, relevant platforms for this dissertation are analysed in the next two subsections.

2.2.1 Texas Instruments

Texas Instruments (TI) offers a broad portfolio of wireless connectivity devices including the lowest power and longest-range solutions across 14 wireless connectivity standards, certified and third-party modules that allow the quick and efficient development of IoT motes. The more relevant TI solutions for this dissertation are:

- **Wireless MCU SoC**: The CC2538 and the CC2650 are wireless SoC that combine a powerful Arm Cortex-M3-based MCU and a robust IEEE 802.15.4 radio transceiver. This combination enables the device to handle complex network stacks with security, heavy processing applications, and over-the-air download. Powerful hardware security accelerators enable quick and efficient authentication and encryption, while leaving the CPU free to handle application tasks. The multiple low-power modes with retention enable quick startup from sleep and minimum energy spent to perform periodic tasks.

- **SmartRF transceivers**: The CC2420 and CC2520 are true single-chips 2.4 GHz IEEE 802.15.4-compliant RF transceiver designed for low power and low voltage wireless applications, enabling through serial peripheral interface (SPI) interface connectivity in MCUs that do not provide it natively. Extensive hardware support for packet handling, data buffering, burst transmissions, data encryption, data authentication, clear channel assessment, link quality indication and packet timing information are included in the chip.

- **Development Platforms**: SmartRF05EB, SmartRF06EB offer support for the two MCU described before providing hardware that enables the fast development of IoT solutions. The SmartRF06 Evaluation Board is designed to the CC2538 and other evaluation module (EM) from the same family, providing an integrated XDS100v3 debug probe that allows the download and debugging of software on the CC2538 SoC. This debugger is supported by several integrated development environments (IDEs) such as IAR Workbench and Keil for Arm Cortex-M.

2.2.2 Microsemi SmartFusion2

The FPGA market is accounted for USD 63.05 billion in 2017 and growing at a fast pace, being expected to reach USD 117.97 billion by 2026 [82]. This ebullient market offers a variety of solutions from different suppliers and targets the specificities of a wide range of scenarios. Therefore, a variety of development boards could have been considered to fulfil the requirements stated for low-end IoT devices. From Microsemi, was considered the PolarFire and IGLOO2 FPGA Families [83, 84], but both failed in provide a hard-core MCU. The same fact is observed in the LatticeXP2 [85], where only a soft-core can be included. From Xilinx, the Zynq-7000 series comes with an Arm Cortex-A9 [86] that is not appropriated for a low-end device. The low-cost family from Altera the Cyclone V [87] includes also an Arm Cortex-A9, what for the previous reason does not fulfil the requirements specified before. Nevertheless, Microsemi produces the SmartFusion2 [88], a powerful FPSoC that combines an Arm Cortex-M3 MCU and FPGA fabric with power efficiency, exceptional reliability and proven security. This SoC is included in the SmartFusion2 Security Evaluation Kit [89], and provides an easy and secure platform for the development of cost-optimized designs suiting perfectly the stated hardware requirements.

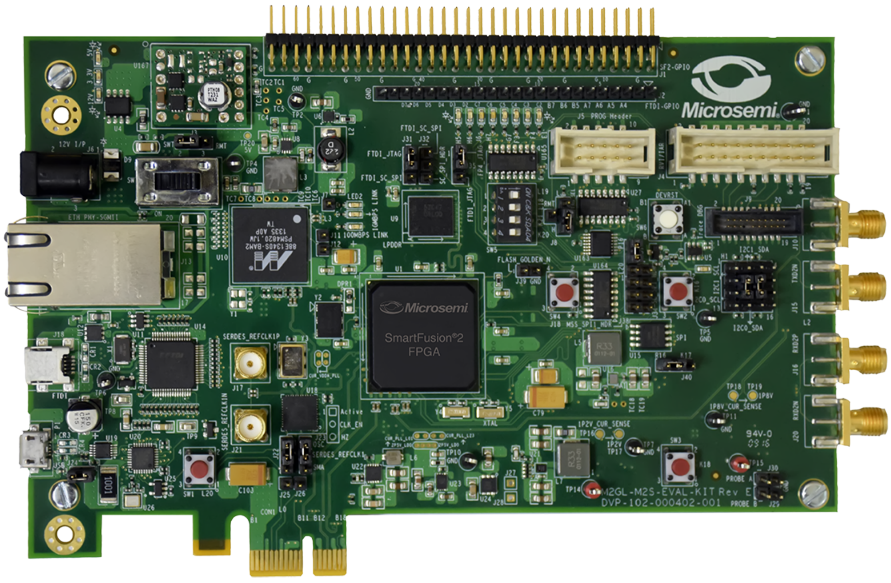

Figure 2.4: SmartFusion2 Security Evaluation Kit.

This platform includes 90K logic elements (LE) in the FGG484 package (M2S090TS-FGG484), a hard-core up to 166 MHz Arm Cortex-M3 processor, 64 Mb SPI Flash memory and 512 Mb on-board low-power double data rate (LPDDR) static random-access memory (SRAM). Is strengthened with industry-required high-performance communication interfaces such as PCI Express Gen2, fullduplex Serializer/Deserializer (SerDes), RJ45 for 10/100/1000 Ethernet and JTAG/SPI programming interfaces. The board capabilities are extended with embedded trace macrocell (ETM), embedded static random-access memory (eSRAM), embedded non-volatile memory (eNVM), and an extensive cluster of peripherals including CAN, TSE, USB, inter-integrated circuit (I2C), SPI and universal asynchronous receiver-transmitter (UART).

The SmartFusion2 brings to market a total power reduction of approximately 20% to 40% with a typical standby power consumption of 7mW. Improves security with implementation of protection against overbuilding and cloning, and secure boot for both the FPGA and processor. It enables data security by providing elliptic curve cryptography (ECC), SRAM-physically unclonable function (PUF), random number generator (RNG), advanced encryption standard (AES), secure hash algorithms (SHA), differential power analysis (DPA) and by implementing anti-tamper measures. It offers an exceptional reliability for safety critical and mission critical systems through the provision of single event upset (SEU) immune Zero FIT Flash FPGA configuration. More architecture highlights include hardware based 667 Mbps DDR2/3 controllers and integrated DSP processing blocks. Other utilities and features can be found on the board such as push-buttons, switches and LEDs for demo purposes, as well as test points for current and power consumption measurements.

2.3 Evaluation Tools

The inherent complexity of developing IoT-enabled devices requires powerful tools that allow a fast and efficient development cycle. Even more, the co-design with integration of hardware accelerators requires simulation features and the possibility to debug the hardware and software developed. The usage of industrial compliant, standard and well-known tools with a large community of developers are less likely to have undetected bugs can provide a better support. Next subsections describe the selected tools used throughout the development of this dissertation.

2.3.1 IAR Workbench

The IAR Workbench IDE from IAR Systems provides a complete development tool for embedded systems. Contain tools for software compiling, linking and debugging specially designed for embedded processors. To easy the development this IDE offers preconfigured files for a wide range of devices, including header files for peripheral I/O, linker configuration files, device description files and flash loaders. Also, it allows the usage of customized toolchains, by invoking external tools and offers a command line environment.

The debugger capabilities offered by IAR are one of the most extensive in the market, thus they provide a great analyser to debug all system components, including power analysis tools, and even RTOS middleware analysis. The debug capabilities of IAR are especially important in this dissertation work, due to the complex and extensive tasks of receiving and sending packets through a radio interface. This IDE comprises rich documentation and a responsive technical support, speeding up the development cycle.

2.3.2 Libero SoC Design Suite

Libero SoC Design Suite offers high productivity with its comprehensive, easy-to-learn, easy-to-adopt development tools for designing with Microsemi's IGLOO2, SmartFusion2, RTG4, SmartFusion, IGLOO and ProASIC3 families. The suite integrates industry standard Synopsys Synplify Pro synthesis and Mentor Graphics ModelSim simulator, programming and debugging tools capabilities and secure production programming support.

The design in Libero can be made using SmartDesign (a visual block-based design creation tool, that helps to assemble and connect cores from different providers), System Builder (design tool that uses a set of high-level questions to define the system), hardware description language (HDL) or embedded design flows. Simulation can be done at functional, gate-level, and timing verification using Mentor Graphics ModelSim Pro ME.

ModelSim Pro ME provides a mixed-language simulator for Verilog, SystemVerilog, and VHDL. This source level verification tool allows the verification of VHDL code, line by line. It enables the execution of simulation at all levels: behavioural (pre-synthesis), structural (post-synthesis), back-annotated, and dynamic simulation. It also allows the simulation of peripherals connected to the MCU through the Advanced Microcontroller Bus Architecture (AMBA) bus, thus reveals very useful for the work developed during this dissertation.

Synplify Pro ME is included for synthesis proposes, enabling the fully optimization of an HDL design for any Microsemi device. Synphony Model Compiler ME, part of the Microsemi Digital Signal Processing Solution, enables the digital signal processor (DSP) designer to evaluate an algorithm at a higher level of abstraction using MATLAB and Simulink along with an exhaustive set of DSP blocksets and Microsemi intellectual property.

For power analysis, SmartPower allows the identification of static and dynamic power consumption problems, estimation of power consumption of the overall system or individual components. For battery-powered design, it allows the definition of power modes and the estimation of the battery lifetime. Similarly, to SmartPower, it is provided the SmarrTime, a tool that allows a complete design timing analysis. Microsemi offers the first industrial programming solution to prevent overbuilding with Secure Production Programming Solution (SPPS). This enables the control of the number of devices programmed, certifies the programmer and ensures that only original Microsemi devices are programmed.

The debug solutions offered by Libero include SmartDebug and Synopsys Identify ME. SmartDebug provides a new approach to debug FPGA fabric and SerDes without the usage of an integrated logic analyser (ILA). The SmartDebug captures the FPGA device status and flash memory content, that gives access to any LE, and check the inputs and outputs in real time. Finally, Synopsys Identify ME, an on-chip debugging tool allows the detection of design bugs, by providing internal signals directly from the FPGA at system speed, these signals are insert into the register-transfer level (RTL) files.

2.3.3 Thread-Metric Benchmark Suite

The Thread-Metric Benchmark Suite is an open-source, vendor-neutral, free benchmark suite used to measure RTOS performance [90]. Thread-Metric is composed of eight distinct tests designed to highlight commonly used aspects of an RTOS, measuring each of them a specific low-level service of the kernel. In more depth, the benchmark measures cooperative and preemptive context switch, interrupt processing with and without preemption, message passing, semaphore processing, memory allocation and deallocation mechanisms. A single evaluation measure the total number of RTOS services that can be handled during a predefined timer interval. To obtain normalized values, a calibration test can be performed to determine the processor speed [90]. In the context of this dissertation, the benchmarks are used to evaluate the impact of offloading software-based functionalities to dedicated hardware accelerators.

2.4 Heterogeneous Architectures

Due to the ubiquitous deployment of IoT endpoint devices arises diversified requirements for these motes. Consequently, the motes can incorporate a variety of sensors and communication interfaces and their demand for performance is increasing proportionally with the ever-growing amounts of data collected, processed and transmitted over a network. This variety of requirements and solutions adopted creates a large heterogeneity in the IoT edge. However, at the network edge a good balance between performance and power consumption should be achieved [25], guarantying easy integration of new hardware and low-cost price per device. Therefore, three hardware architectures are analysed having in consideration the previously stated:

1. **A COTS MCU** is the traditional solution for WSN due to the homogeneity of those networks. Several COTS examples can be found in [91–98] however, they fail in provide flexibility, customization or in enabling easy integration with new hardware;

2. **A graphics processing unit (GPU)** can be used to increase performance due to the natural parallelism features of these platforms. Although, due to

the common power consumption constraints, solutions such as GPU are not suitable for endpoint devices [25];

3. **An ASIC**, a custom platform, provides high performance with reduced power-consumption, but presents drawbacks, being the high cost to volume ratio, high time-to-market, and lack of flexibility (once fabricated cannot be changed). Thus, the application of an ASIC on customized edge-devices with a small production may not be viable, due to the high NRE value, manufacturing cost and long time-to-market;

None of the presented solutions fulfil the common requirements for an IoT platform and neither the proposed in Section 2.2. Therefore, a new hardware architecture is required to improve the wireless motes, were in the current state of the artFPGA technology is being used to overtake these challenges. However, FPGA was not a choice in the past to develop wireless motes, mainly due to their high power consumption and associated costs.

Although, in recent years FPGA suppliers addressed the energy consumption problems, and are turning their platforms SoC-oriented (FPSoC). Xilinx and Intel (former Altera) are the main supplier of FPGAs, with Microsemi providing the latest security technology innovations. In the last years, these suppliers have released FPSoC platforms with several of them including Arm CPU's [99]. In Figure 2.5 is presented the evolution of FPSoC in the market, regarding the number of processor cores used. The difference of heterogeneous and homogeneous architecture, in this case, refer to the use of different CPUs in the same SoC.

Nevertheless, usually FPGA chip unit price is higher than ASIC, the development period is much shorter due to the unnecessary manufacturing process, and the NRE cost reduced due to the incorporation of on-board facilities that enable a fast development such as an ILA. FPSoC are of special interest to deploy CPU accelerators, enabling the implementation of protocols and hardware blocks designed by users, integrating new functionalities with widely deployed hardware and software with minor changes. Therefore, these recent advances allow the usage of FPGA enabled platforms, specially FPSoC for prototyping new functionalities or develop heterogeneous endpoint devices. Some examples of FPGA applications, are the improvement of hardware security [100–102], implementation of cryptography algorithms [103–107] and compression/decompression mechanisms [108].

An emergent solution for wireless motes is the conjugation of a reconfigurable computing unit (RCU) (built in FPGA fabric), with a hard-core MCU and an 802.15.4 compliant radio transceiver [109]. These heterogeneous wireless motes enable the integration of compatible IP network stacks, allowing to endow the

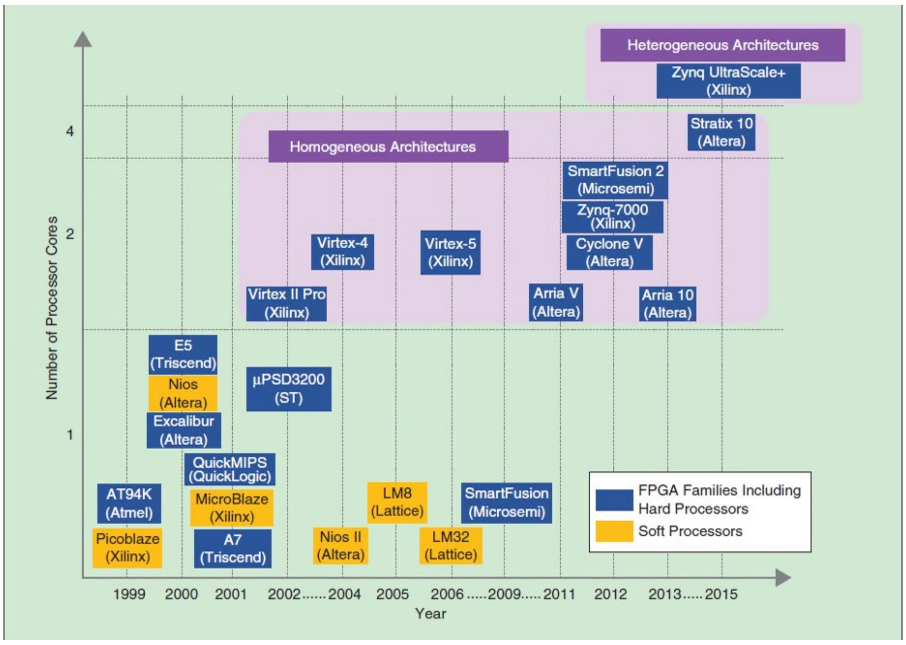

Figure 2.5: The evolution of FPSoCs [25].

devices with reconfigurability and customization capabilities. These architectures including an RCU present already various applications in the IoT arena, being the most relevant state-of-art research related with the core of this dissertation the HaLoMote [110], the PowWow Mote [111], the CookiesWSN [112], and the in-house project CUTE mote [113] that are discussed below.

In [110], aiming to address the challenges of computationally intensive distributed applications, with limited energy budgets is proposed the heterogeneous Hardware-Accelerated Low-Power Mote (HaloMote) as a more energy-efficient approach to the common homogeneous node architectures. The mote is composed by an MCU with integrated IEEE 802.15.4 radio transceiver and a Microsemi Igloo M1AGL1000 FPGA used as RCU. The first version adopted a CC2538 radio frequency (RF)-SoC as MCU while, in the latest versions was used an ATmega256RFR2 RF-SoC. Aiming to reduce power consumption, the RCU hosts dedicated hardware accelerators designed to perform dynamic power management (DPM) mechanisms and efficient data compression algorithms. For the same mote, a forward-adaptive differential pulse code modulation, with a Rice symbol coder was used to compress vibration data from microelectromechanical systems (MEMS) sensors in [114]. The first published version integrated in the RCU data aggregation functionalities, for a structural health monitoring system, aiming to reduce the data sent through the wireless interface [115].

PowWow [111], a power optimized hardware and software framework for wireless motes, aiming to provide energy efficiency on the WSN motes. The heterogeneous solution is composed by a TI CC2420 IEEE 802.15.4 compliant RF transceiver, a MSP430 SoC running Contiki-OS and an Igloo FPGA platform from Microsemi as RCU. To achieve the energy efficiency, low-level network accelerators were deployed in the RCU, to assist Contiki-OS in link-layer tasks such as error correction codes (ERCC). The goal of this layer is to manage the Automatic Repeat Request (ARQ) and forward error coding (FEC), in order to guaranty a reliable link between two nodes. These low-level accelerators allow to maintain the MCU in sleep mode, waking-up only when an IEEE 802.15.4 frame need to be forward to Contiki-OS. Even more, for energy saving proposes PowWoe adds a Dynamic Voltage and Frequency Scaling (DVFS) mechanism, allowing to dynamically adapt the voltage and the frequency of the processor and consequently minimize the power consumption.

CookiesWSN [112] mote uses the same COTS used in PowWow for the heterogeneous solution development. CookiesWSN uses an ultra low-power FPGA to implement wake-up radio (WuR) and achieve ultra-low energy in WSN, taking advantage of their speed, flexibility and low-power consumption. Other hardware accelerators were developed in [116–118] to improve security functionalities.

Recently a novel architecture has been proposed in [113, 119], this in-house work proposes a heterogeneous architecture for IoT edge devices, using a FPSoC to offload critical features of the communication stack to hardware. The platform selected was the SmartFusion2 Security kit from Microsemi, that includes a hard-core processor (Arm Cortex-M3) and FPGA fabric in the same SoC. As OS was used Contiki-OS and to provide communication capabilities integrated an IEEE 802.15.4 radio transceiver with a SPI interface.

In this architecture a hardware accelerator is deployed between the radio and the MCU, allowing filter incoming frames are pre-process them in parallel with the CPU execution. This allows to keep the processor free to execute another task or keep it in low-power consumption modes, being only interrupted when a frame needs to be forward to higher layers for further processing. Is proposed that the filtering scheme can filter IEEE 802.15.4 fields such as, PAN_ID, SHORT_ADDR, and EXT_ADDR.

In [120], a work in progress, is proposed a hardware-based Network Packet Filtering (NPF) and an IPv6 Link-local address calculator which can filter IPv6 packets, offering nearly 18% overhead reduction. Aiming to obtain a SoC implementation that can be deployed in future IEEE 802.15.4 radio modules, the

NPF implements, in the presented state, (i) IP source address filtering, and (ii) IP destination address filtering. Other features are proposed to add extra filtering capabilities impacting the transport layer, by analyse the (i) UDP source and destination port, and the (ii) UDP source or destination address field. According the author, this allows to introduce a novel concept on packet filtering, by adding IP addresses to a White List Address (WLA) or to a Black List Address (BLA). The NPF will only interrupt the OS running on the MCU, when an IP packet has been verified and accepted, increasing the availability of the OS to perform other activities. The added value of the NPF is more important when applied to a Network Router (NR), due to the higher amount of data processed.

Due to the large number of received packets in IP-based IoT-enabled networks, a 6LoWPAN accelerator is proposed in [121]. This accelerator aims to process and filter IPv6 packets received by an IEEE 802.15.4 radio transceiver, without a CPU intervention. The 6LoWPAN packet filter is composed by a 6LoWPAN Internet protocol header compression (IPHC) module that performs the compression and decompression of the received packets, also, the accelerator verifies the local and remote ports of valid packets. Even more, the 6LoWPAN accelerator incorporates three filters for the IEEE 802.15.4 MAC layer, (i) PAN, (ii) duplicate frame detector (DFD), and (iii) source and destination address.

The results published in this work, shows an overhead reduction of 13.24% for packet processing and address filtering. Furthermore, allows full OS availability when an unwanted packet is received. In terms of resource utilization was used 6521 4lookup table (LUT) (7.57%) and 3974 D flip-flop (DFF) (4.61%). The hardware packet processing may reach a reduction between 56.01% and 72.39%, for packets to be accepted and discarded, respectively. Concerning the energy consumption, was described a reduction from 16503μJ to nearly 22μJ, comparing the software-based processing to hardware, when a packet is discarded.

An IEEE 802.15.4 accelerator was presented in [122] with the capability to perform (i) third-level of filtering, as specified by the IEEE 802.15.4 standard, (ii) packet handling, such as, detection of duplicated frames and (iii) control of the MAC protocol.

The architecture used was the same described in [120], deploying on the RCU (between the MCU and an IEEE 802.15.4 radio transceiver) an MLA, using an SPI interface to establish communication between the MLA and the radio. Furthermore, to exchange information betwixt the MLA, the microcontroller subsystem (MSS) and the SPI hardware blocks was used an AMBA - Advanced Peripheral Bus v3 (APB3) bus interface.

The MLA performs filtering and processing of incoming Mac Protocol Data Unit (MPDU), aiming to discard unwanted MPDU before they reach the MCU. Therefore, the MLA provides (i) runtime configuration of an addresses list for comparison with incoming packets, (ii) runtime number definition of packets to be held by the duplicate MPDU detector, (iii) reallocation of incoming MPDU from the radio RX FIFO to its own RX FIFO (for further processing), (iv) acceptance or rejection of MPDU according the defined addresses list, (v) dispose of duplicated frames according the defined rule, (vi) OS notification of an available MPDU to be forward to the network stack.

The resulted obtains by the author shows that the implementation of hardware filters can achieve 17% of overhead reduction on packets processing. With the MLA accelerator integration is prevented the triggering of unnecessary wake-up calls to the MCU and, consequently, interrupts to the OS achieving up to 59% increased system availability.

More recently the CUTE mote has been presented in [113]. This heterogeneous architecture proposes the integration of an RCU unit between an MCU and an IEEE 802.15.4 radio transceiver, in the following of the work developed in [120]. This solution specially designed for low-end and resource constrained network edge IoT devices, is described as a customizable and trustable solution targeting low-power IoT applications. Due to the heterogeneous architecture, the CUTE mote split functionalities between components. In the Arm Cortex-M3 MCU is deployed the Contiki-OS, an event-driven OS that incorporates an IP compliant stack, being responsible by handle low priority tasks. The TI CC2520 IEEE 802.15.4 radio transceiver is responsible by handle the physical layer. By its side, the RCU integrates: (i) a DPM system, responsible by performing power saving schemes; (ii) an AMBA bus protocol to interconnect memories and peripherals deployed on the FPGA; (iii) interfaces to access external devices, as the IEEE 802.15.4 radio transceiver; (iv) a data aggregation accelerator, that gathers and compresses data before providing it to the OS; (v) security-related hardware accelerators, enabling encryption/decryption mechanisms applied to the CIA triad processes; (vi) IoT network-related accelerators, specifically (i) a MAC sub-layer accelerator described in [122], (ii) a 6LoWPAN accelerator described in [121], (iii) an IPv6/UDP port filtering accelerator.

The CUTE mote configured with MAC, 6LoWPAN and IPv6/UDP port filtering, presents a hardware utilization of 6521 (7.57%) 4-LUT and 3974 (4.61%) DFF. The power consumption analysis revealed that in active mode the power consumed, on average, is 56.52 mW while in Flash Freeze mode is 8.23 mW.

Furthermore, the processing of an IPv6 packet exclusively in software, when is accepted or rejected, uses respectively 14356 nJ and 16503 nJ, while with the hardware accelerators the values drop to 12168 nJ and 22 nJ.

2.5 Conclusion

This Chapter described and discussed the tools and platforms needed for the development and evaluation of the heterogeneous low-end IoT solution. The state of the art in the development of heterogeneous architectures, with the integration of hardware accelerators shows unequivocal advantages either in performance or power consumption. Therefore, to develop a heterogeneous architecture for the IoT edge will be combined the Microsemi SmartFusion2, that includes FPGA fabric and the widely deployed low-end Arm Cortex-M3 CPU, with an IEEE 802.15.4 compliant radio. Next Chapters describe the deployment RIOT-OS and the integration of the acceleration support provided by the mote. Moreover, it is also compared the performance increase of using such platforms over traditional COTS solutions.

Chapter 3

RIOT-OS Deployment on a Reconfigurable IoT Mote

"Strive for perfection in everything you do. Take the best that exists and make it better. When it does not exist, design it."
—Sir Henry Royce

The transition from the traditional WSN to IoT is bringing new challenges in developing low-end devices. The usage of IP network stacks and the increasing number of devices exchanging information in the network is forcing the low-end devices to perform heavy and complex tasks. Such condition demands for increased processing capabilities, while maintaining the low-cost and low-power requirements. Yet, increasing performance while accomplishing the remaining constraints in low-end devices is not straightforward, requiring new architectural solutions, both at the software and hardware levels.

This Chapter starts by introducing the heterogeneous architectures in Section 3.1 together with an explanation of the in-house project CUTE mote. Section 3.2 presents the structure of RIOT-OS and the integration issues at the compiler level to enable the usage of IAR Workbench. The development of the heterogeneous architecture is presented in Section 3.3, with the deployment of RIOT-OS in such mote. Section 3.4 shows and discusses the results obtained from a performance evaluation realized to the system, while Section 3.5 concludes this Chapter.

3.1 Heterogeneous Architectures for IoT Low-End Devices

The boundless deployment of IoT low-end devices is making the concept of ubiquitous computing presented by Mark Weiser back in 1988 a reality. This concept consists on the deployment of pervasive, invisible and enhanced computers into the background of human activities, improving human performance in a non-intrusive way. Furthermore, this new trend is favouring the integration of WSN devices into the IoT ecosystem. Mainly, the WSN use the IEEE 802.15.4 standard that was developed targeting low-power and low-cost wireless local area networks (WLANs), addressing the energy consumption with its low-power transmissions and low-data rate links [123]. The IEEE 802.15.4 was not conceived targeting IoT networks and consequently is not able to handle the large amount of data required by such a complex network, which difficulties the integration of traditional WSN devices in IoT.

In resource constrained application scenarios, such as IoT, high traffic on the network heavily decreases the overall energy consumption, system availability and device's lifetime. Traditional strategies to reduce the amount of data handled by low-end devices include the reduction of packet size, compression mechanisms, and cutback of transmissions. Known strategies to tackle the energy consumption on the low-end devices may include the reduction of the node's operation and radio duty cycle, and the disabling of system components when not in use. On the other hand, increasing the performance of low-end devices, while maintaining the low-power and low-cost constraints, and aiming to fulfil the connectivity and interoperability requirements stated in Section 1.5 from Chapter 1, can be a hard task to achieve, requiring new solutions at different levels of the mote itself.

In recent years, FPGA manufacturers have addressed the high-power consumption of such technology. Furthermore, turn their architectures SoC oriented, with the inclusion in a single SoC of a hardcore MCU and FPGA fabric. These platforms are known as FPSoC and due to their versatility, flexibility, high performance, security, scalability, ease of design and short time to market, are expected to play a key role in development of IoT technology [25, 124].

Legacy FPGA-enabled devices were based in SRAM memories. Such memories are volatile, thus its content is lost when they are powered off, which makes them impossible to be used in low-power modes on the RCU deployed in this type of FPGA. Recent flash-based FPGA solutions offer several advantages over its predecessor, such as the reduced energy consumption and the ability to retain its content when the FPGA is turned off. Therefore, this enables the development of energy aware architectures based on FPGA.

Heterogeneous solutions incorporating an RCU based on flash memory and a wireless transceiver have already been proposed [125–127]. However, these solutions do not suite the power consumption constraints of the IoT low-end devices. In the solutions already proposed all the tasks are processed in the RCU, even long-term and low-intensity computational tasks, and at least one of the devices should be always awake [110]. Based on the recent state of art, the best solution for the development of heterogeneous solutions for low-end IoT devices relies on the usage of FPSoC platforms that include a low-power hard-core MCU with a FPGA flash based RCU [110, 113].

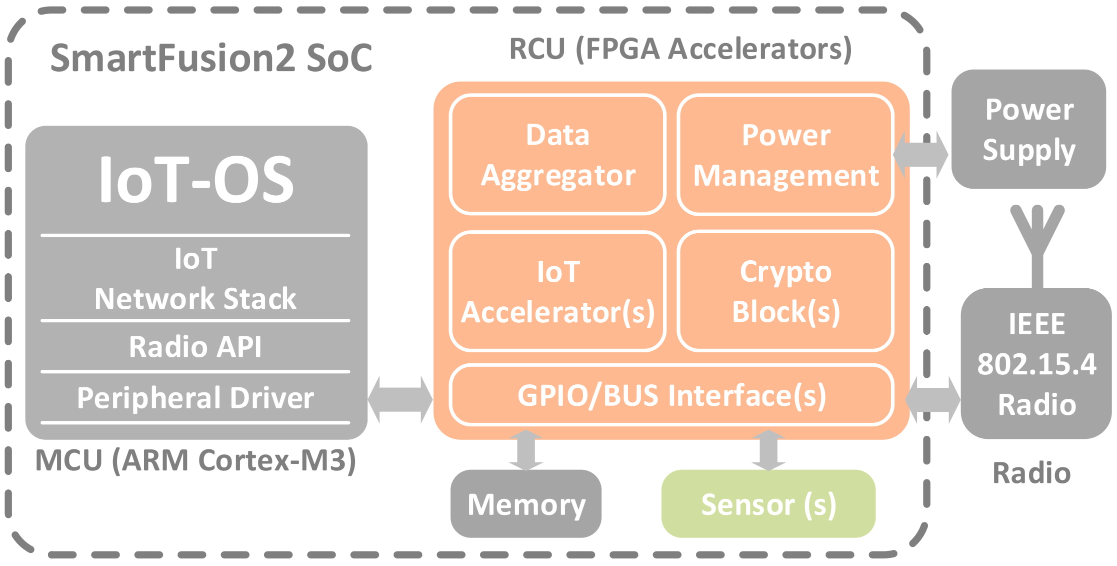

Figure 3.1: CUTE mote architecture [113].

The in-house project CUTE mote [113] presents at the time of the development of this dissertation, and for the best of author's knowledge, the state-of-the-art in low-power heterogeneous solutions for IoT low-end devices. Therefore, the CUTE mote architecture presented in Figure 3.1 is used in this dissertation.

The selected platform to deploy CUTE mote was the Microsemi SmartFusion2 FPSoC, which includes a hardcore Arm Cortex-M3 MCU and FPGA fabric. In this architecture the RCU is strategically positioned between the radio transceiver and the MCU to intercept incoming data from the radio transceiver before it reaches the MCU. This way, it is possible to process the streaming of data in hardware accelerators and then make it available to the MCU through an AMBA protocol only when needed.

The design of the CUTE mote requires the integration of three fundamental components, as stated in [113]: (i) an IEEE 802.15.4 radio transceiver, used to fulfil the communication requirements; (ii) an RCU unit, responsible to enable the integration of hardware accelerators; and (iii) an MCU, which is responsible to host the OS and running software-only tasks.

In the CUTE mote architecture, the RCU is responsible to perform more intensive and high priority tasks, while the low-power MCU handles less intensive and low priority ones. Thus, to perform the intensive tasks the RCU integrates diverse hardware blocks, namely: (i) a DPM hardware block to accomplish the low-power requirements of the CUTE mote; (ii) an AMBA bus interface that allows access peripherals and establish communication between the MCU and the RCU; (iii) a module that allows access to external devices such as sensors or the radio transceiver; (iv) a data aggregation accelerator to compress/aggregate data before being dispatch to the OS; (v) an IoT network accelerator to handle the data exchanged by nodes on the network; and (vi) security-related accelerators that perform encryption/decryption algorithms as well as security protocols [113].

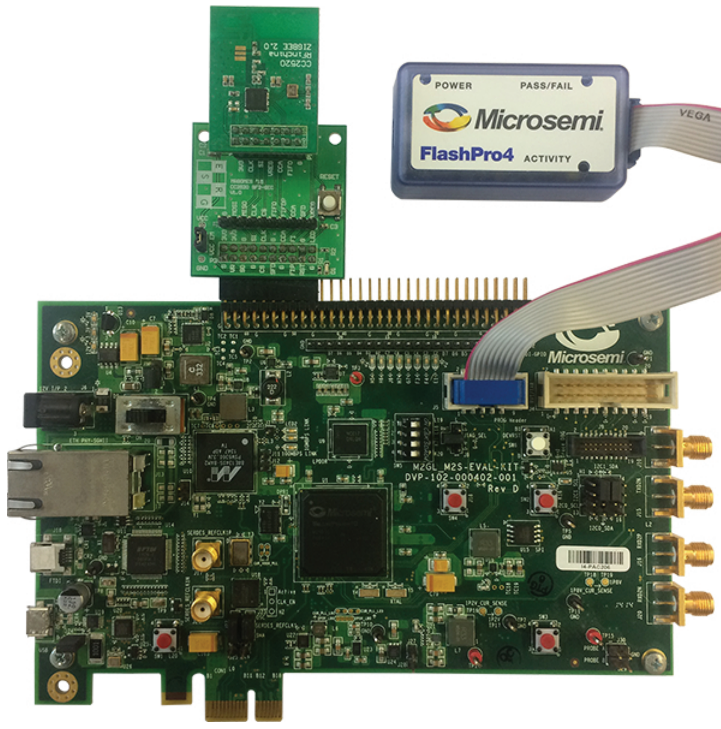

Figure 3.2: CUTE mote hardware prototype [113].

The communication between blocks in this architecture benefits from the usage of AMBA buses in such a way that the hardware accelerators, that are memory mapped, can be accessed by the MCU as common peripheral devices. The integration of such soft-peripherals in an OS is also facilitated, by only requiring the development of a device driver.

Figure 3.2 the prototype used to develop the CUTE mote [113]. The prototype integrates a CC2520 IEEE 802.15.4 radio transceiver that solution the platform with the required connectivity requirements, as stated in Section 1.5 from Chapter 1.

3.2 RIOT-OS on a COTS Solution

As stated in Section 1.7 from Chapter 1, an OS suitable for IoT should, in brief, provide a small memory footprint, energy efficiency, real-time capabilities, network connectivity, security and support heterogeneous architectures.

Contiki-OS is the embedded OS used in CUTE mote because it offers support for Arm Cortex-M3 that is present in Microsemi SmartFusion2, presents a small footprint IoT-enabled network stack, good on-line support and a broad development community [113]. However, Contiki-OS presents an event-driven programming model, and even with the partial multithreading support, the OS does not offer real-time capabilities, failing in fulfiling all the desired requirements when conceiving an IoT-enabled OS.

Table 3.1: Embedded OSes features comparison.

Name	***Architecture***	***Programming Model***	***Scheduling Strategy***	***Real-Time***	***6LoWPAN***
Contiki-OS	Modular	Event-driven	✗	✗	✓
TinyOS	Monolithic	Event-driven	EDF	✗	✓
RIOT-OS	Microkernel	Multi-threading	Preemptive	✓	✓
FreeRTOS	Library	Multi-threading	Preemptive	✓	✗

Section 2.1 from Chapter 2 presents, besides Contiki-OS, three prominent state-of-the-art OS suitable for low-end devices. In Table 3.1 is presented a resume of the key features of those OSes. From the analysed, TinyOS fails in provide real-time capabilities and consequently is not suitable. FreeRTOS one of the most deployed OS in industrial applications offers a large support and documentation, a predictive and low-latency real-time scheduler and one of the smallest OS memory footprint. However, this OS fails in provide a native network stack that fulfil the connectivity requirements stated in Section 1.5 from Chapter 1 and for this reason is not used in this dissertation.

RIOT-OS follows a microkernel architecture with high granularity that allows the customization of memory footprint through the selection of specific software blocks for each application. To achieve energy efficiency, it adopts a tickless scheduler with soft-real time characteristics. The network connectivity is provided with a full modular communication stack, supporting a variety of standard protocols such as IPv6, RPL, 6LoWPAN, UDP, CoAP, etc. RIOT-OS provides security by design due to the microkernel architecture that avoids the propagation of errors or attacks. Moreover, it provides memory isolation where each thread has its own address space. Offers rich documentation, scientific publications, and a large community of contributors that increase continuously the number of supported architectures and platforms. RIOT-OS fulfils the requirements for an OS suitable for IoT low-end devices and for this reason, it was selected to be used in this dissertation, in order to increase the OS support on the CUTE mote.

The microkernel architecture of RIOT-OS and its high granularity allows to divide all the OS components in small blocks called modules. The granularity of RIOT-OS achieved such a condition that each of these modules represent a functionality, protocol or hardware interface that can be added or removed at during compile time, according to the application requirements.

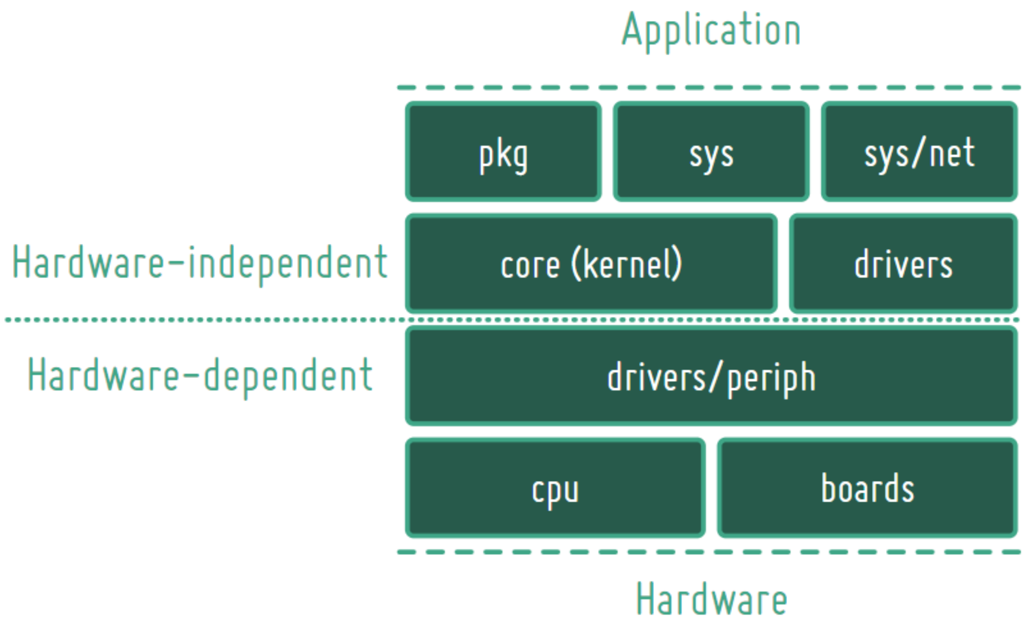

Figure 3.3: RIOT-OS structure [128].

The RIOT-OS organization is described in [128], at a top-level, it can be divided in two layers with one hardware-dependent layer that incorporate modules dedicated to the low-level implementation of hardware interfaces, and a hardware-independent layer that aggregates OS and application related modules. In general the RIOT-OS architecture can be categorized in five groups:

1. **Core**: It is the kernel of RIOT-OS, containing the kernel OS functionalities such as the scheduler, threading, synchronization (mutexes, mailboxes and messaging/IPC), IRQ-handling, startup and configuration code, support for kernel data-structures, type definitions, etc.;

2. **Platform specific code**: It can be divided in two submodules, CPU and board, maintaining a 1-to-n relationship where a board has exactly one CPU while a CPU can be part of many boards, they can be described as:

   - **CPU**: Contain all CPU specific configurations, such as implementations of power management, interrupt handling vectors, startup code, clock initialization and thread handling (e.g. context switching) code. Each CPU contains its own peripheral drivers implementation such as I2C, GPIO, SPI, UART, etc. Due to the common features of some processor families, e.g. Arm Cortex-M, low-level code like task switching and interrupt handling is shared among different CPUs, being these files denoted with a "common" suffix;
   - **Boards**: Present specific configurations for the CPU on each board, including pin-mapping, peripherals, on-board devices and CPU's clock configuration. On top of this, it can be included configuration files needed to interface the board, typically custom flash/debug scripts. Generally, a board consist of a fixed configuration of a controller and some external devices such as sensors or radios. Some configurations could be shared among vendors, e.g. stm32, with the notation of the "common" suffix;

3. **Device drivers or drivers**: Provide interface for external devices such as network interfaces, sensors, actuators, memories, etc. With the usage of a peripheral driver API and other RIOT-OS modules like xtimer, the OS is made completely platform agnostic, eliminating CPU or board dependencies;

4. **Libraries and network code**: Contain external, network and system libraries providing the tools and utilities that turn RIOT-OS into a full-fledged OS, this layer cam be divided in three sections:

   - **sys**: System library modules that provide tools and utilities, including data structures (e.g.color), crypto libraries (e.g. hashes, AES), high-level APIs (e.g. Posix), memory management (e.g. malloc), RIOT-OS shell, among others;

- **svs/net**: Provide the network stack implementations (e.g. GNRC stack) or agnostic network stack code as header definitions or network types;
- **pkg**: External libraries and applications supported by RIOT-OS (e.g. ccn-lite, microcoap). Such libraries are included through a custom Makefile that downloads the library and, if required, applies patches to enable their full integration with RIOT-OS;

5. **Application**: It is where the code for a specific application can be found. It runs in user-space, isolated from the RIOT-OS kernel.

To have a comparison platform with the work developed around CUTE mote, firstly, it was used the TI CC2538 Radio-Frequency System-on-a-Chip (RFSoC) device, previously described in Section 2.2 from Chapter 2 and that is considered one of the best devices in the market that can include on the same SoC an MCU and a radio transceiver. Furthermore, the CC2538 includes a low-end Arm Cortex-M3, which is the same MCU present in the Microsemi SmartFusion2. Despite being micro architecturally different, this can help in making closer comparisons between both systems.

RIOT-OS is a free distribution built around the open-source GNU toolchain without specifying any IDE or compliant framework. The development of a heterogeneous solution is more likely to contain errors, thus powerful software debugging tools are required to empower the development cycle. For these reasons, an IDE was adopted to develop the software code, being selected one of the most popular IDE specially targeting embedded devices, the IAR Workbench, which is described in Section 2.3 of Chapter 2.

The TI CC2538 SoC is already supported by RIOT-OS and due to the usage of GNU compiler, the compilation process is based on Makefiles. The usage of such Makefiles is not possible in IAR, thus this Makefiles were analysed and determined the specific modules that are necessary to run the full kernel in this platform with the inclusion of communication capabilities through the generic network (GNRC) stack with the protocols UDP, RPL, IPv6, Internet control message protocol (ICMP), 6LoWPAN and the IEEE 802.15.4 MAC layer.

To allow the usage o IAR Workbench over RIOT-OS some code was adapted to be compliant with the compiler and assembler, and some files were added as workaround to the Makefiles. In summary the modifications were:

- **irq_arch.c**: In this file were modified GNU compiler directives that are not compliant with IAR. For instance, Listening 3.1 shows where the attribute of the function `irq_enable used` was modified to `__root`, avoiding the linker to remove this function on optimizations;

Listing 3.1: `irq_enable` function of RIOT-OS's `irq_arch` file

```c
/**
 * @brief Enable all maskable interrupts
 */
__root unsigned int irq_enable(void)
{
    __enable_irq();
    return __get_PRIMASK();
}
```

- **thread_arch.c**: It was necessary the inclusion of IAR pragmas to allow calls to functions from the assembly code. Also, due to the differences between the IAR and GNU assembler, it was necessary to modify some instructions and function attributes. Listening 3.2 illustrates the use of the pragma `required` ensuring that a needed symbol can be found in the linked output. The `__noreturn` directive was also modified to be IAR compliant, indicating that this function will not return once is executed;

Listing 3.2: `cpu_switch_context_exit` function of RIOT-OS's `thread_arch` file

```c
#pragma required=irq_enable
__attribute__((naked)) __noreturn void cpu_switch_context_exit(void)
{
    __asm volatile (
    "bl     irq_enable                \n" /* enable IRQs to make the SVC
                                           * interrupt is reachable */
    "svc    #1                        \n" /* trigger the SVC interrupt */
    "unreachable:                     \n" /* this loop is unreachable */
    "b      unreachable               \n" /* loop indefinitely */
    :::);
}
```

- **spi.c**: Contains the SPI implementation for the TI CC2538 SPI peripheral, however, a bug that caused the malfunction of this peripheral was detected, and therefore, the source final was modified;
- **riotbuild.h**: Was added a header file to the OS specifying the modules that should be integrated in the OS during the compilation process;
- **types.h** and **unistd.h**: Such GNU libraries were modified with the correspondent definitions to be used with IAR toolchain;

- **startup_m2sxxx.s**: This file is typically generated by IAR and contains the low-level CPU and board configurations, such as clock frequency, and startup initialization. However, this file is created by IAR having in mind to run a bare-metal application and not an OS. One of the function implementations present in this file is the reset handler interrupt service routine (ISR) executed once the MCU is powered-up. To perform the RIOT-OS configurations for the TI CC2538, the reset handler was modified to call the board initialization routine and later, the execution jumps to the kernel initialization routine. This behaviour can be found in Listening 3.3.

Listing 3.3: `Reset_Handler` function of CMSIS's `startup_m2sxxx` file

```
;-----------------------------------------------------------------------
; Call SystemInit() to perform Libero specified configuration.
;
call_system_init
    LDR     R0,=SystemInit
    BLX     R0

    LDR     R0,=__low_level_init
    BLX     R0

    LDR     R0,=__iar_data_init3
    BLX     R0

    LDR     R0,=board_init
    BLX     R0

;-----------------------------------------------------------------------
; Call kernel_init() startup the kernel
;
    LDR     R0,=kernel_init
    BX      R0
```

In order to verify the correct execution of RIOT, several examples were executed and tested over the CC2538EM along with the SmartRF06EB evaluation boards. These examples include tests to the hardware peripherals, OS functionalities, network interfaces and communication stack.

3.3 RIOT-OS on a Heterogeneous Mote

The Microsemi SmartFusion2, described in Section 2.2 from Chapter 2, is the selected platform to deploy RIOT-OS over a heterogeneous architecture provided by the CUTE mote project. This FPSoC platform offers on the same SoC a hard-core MCU (Arm Cortex-M3) and FPGA fabric accessible through standard AMBA protocols. To endow the CUTE mote with the connectivity features stated in Section 1.5 of Chapter 1, it used the same radio transceiver as initially used in CUTE mote. This radio is the CC2520 from TI and is described in Section 2.2 from Chapter 2.

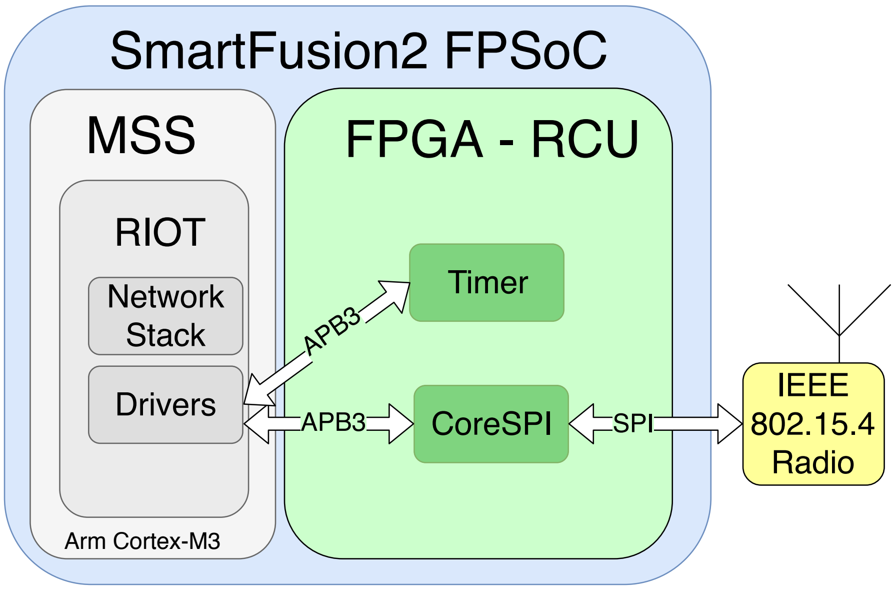

Figure 3.4: Architecture of the heterogeneous solution.

RIOT-OS offers support for a variety of platforms and architectures, supporting the Arm Cortex-M3 present in the SmartFusion2, however, does not offer support for this SoC. Consequently, a porting was made to provide full support of RIOT-OS over such hardware platform. Due to the characteristics of the platform, the process started with the development of the hardware, enabling in the SmartFusion2 the necessary facilities to deploy RIOT-OS and to enable the integration of the TI CC2520 radio transceiver. To configure and develop the hardware, it was used the Microsemi Libero described in Section 2.3 of Chapter 2. Therefore, besides the memory, clocks and pin-out configurations of the SmartFusion2, it were integrated two more hardware blocks, as depicted by Figure 3.4. These hardware blocks are described as follows:

- **Timer**: RIOT-OS requires at least a timer to be integrated in the OS timer subsystem (xtimer). However, the MSS does not provide a timer with the characteristics needed by RIOT-OS, thus it was necessary to integrate a hardware timer in the RCU. Libero's hardware catalog already provides hardware timers, yet, they also do not have the necessary features. Furthermore, it works with the same clock of the APB3 bus, in this case 88MHz, that is not a power of two and consequently makes difficult the usage of a prescaler and the counting of specific periods of time without additional processing. A real-time clock (RTC) with the resolution of one second is available by the MSS, but still, it is not suitable to be used by RIOT-OS. For these reasons, a hardware timer was developed in Verilog and deployed in the RCU, according to RIOT-OS needs.

- **CoreSPI**: The integration of the CC2520 radio transceiver in the heterogeneous architecture requires an SPI controller for the communication tasks. Despite the MSS providing an integrated SPI peripheral, it is deployed a hardware SPI IP core in order to make it possible to intercept the received/-transmitted IEEE 802.15.4 frames. The controller was selected from the Libero catalog that includes a core SPI capable of provide full communication with the CC2520 radio transceiver. To interface the core SPI deployed in the RCU with the external radio transceiver, it was also necessary to configure GPIO pins, required to read and write the radio input/output (I/O) pins as shown in Figure 3.5.

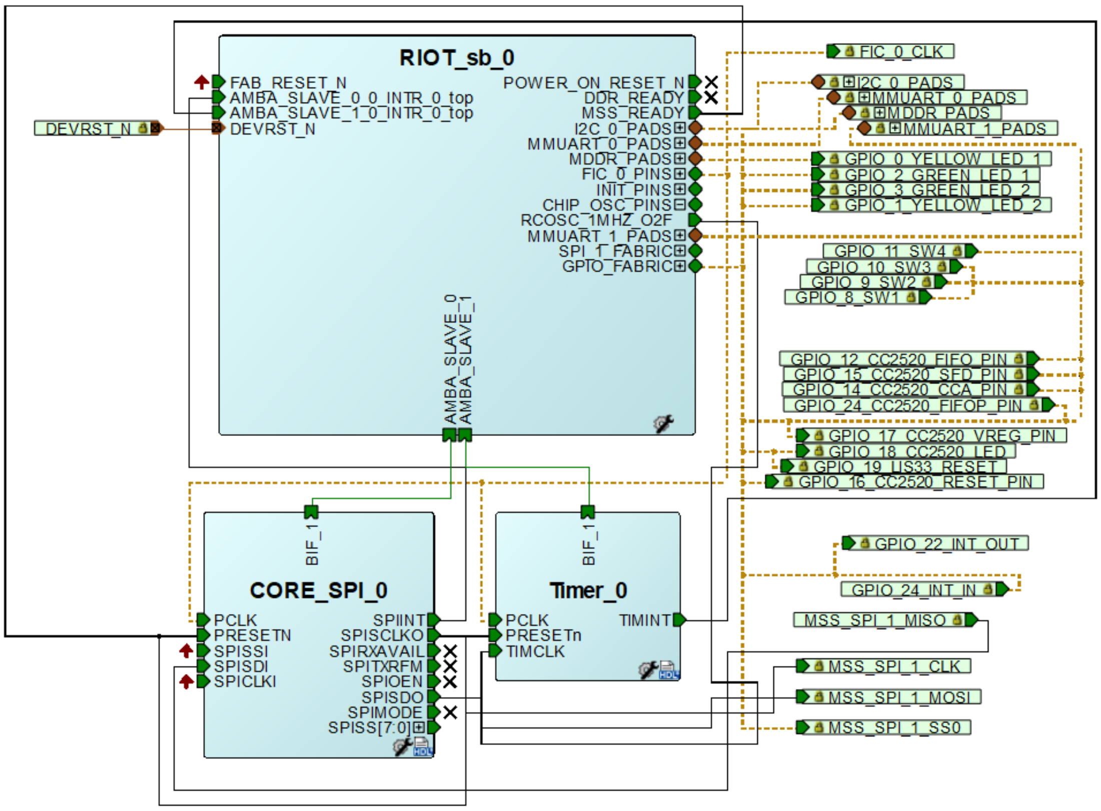

Figure 3.5: Heterogeneous architecture layout.

Due to the microkernel and consequent modular design fashion of RIOT-OS, the porting to SmartFusion2 was at this stage, after the resolution of compiler and assembler related issues in Section 3.2, made exclusively at the hardware dependent level, which includes CPU, board and peripheral drivers related files.

Microsemi provides their own software development environment, the SoftConsole. However, it still can generate and export firmware for well-known IDEs such as IAR. The Microsemi's Libero toll allows the generation of firmware to the core available on the catalog, therefore, this firmware is used to develop the peripheral drivers needed to deploy RIOT-OS in the heterogeneous platform, integrating the firmware in the RIOT-OS's API. The drivers developed were: (i) the GPIO driver used to access to the GPIO ports of the platform; (ii) the timer driver that interfaces the hardware timer deployed in the RCU with the OS; (iii) the SPI driver that remaps the functions provided by the Libero firmware to the SPI driver of the OS; and (iv) the CC2520 peripheral driver that enables the usage of such radio transceiver in the RIOT-OS. Because the CC2520 radio was not initially supported by RIOT-OS, it was necessary to develop and integrate its corresponding driver with the GNRC communication interface.

To provide support for SmartFusion2 on RIOT-OS were created hardware dependent files that are mainly related with the initialization process of the hardware platform, these files are: (i) the periph_conf.h which that discriminates the specifications of the timer; (ii) the board.h and board.c, that contains the specifications of the xtimer implementation to this architecture and does the board initialization; and (iii) the init.h and init.c that is responsible for peripheral initialization before the OS boot stage.

In order to ensure that RIOT-OS was properly running, the software examples provided by RIOT-OS were run and retrieved the expected results.

3.4 Evaluation

The performance of a RTOS is sensitive to several parameters, such as platform, processor, clock-speed, compiler, application design, etc., thus the measuring of performance is not straightforward and can arise several questions around the methods used. Therefore, to avoid ambiguity and give confidence to the obtained results, the Thread-Metric is also used in the heterogeneous solution. Each test was ported to be compliant with RIOT-OS's API and OS services.

3.4.1 Thread-Metric Benchmark Suite on RIOT-OS

Thread-Metric was designed to the ThreadX RTOS but is adaptable to others RTOS through a porting abstraction layer. This benchmark measures the availability of the OS by performing specific OS services at the application layer and measuring the number of times those services are performed during the test. Every test procedure starts with the initialization of the hardware and the kernel, once it reaches the main thread in user space, it makes the needed configurations as well as creates the needed OS threads to perform the desired measurements. Each test is summarized below:

- **Calibration Test**: Establishes a base line for further measurements;
- **Cooperative Context Switch**: Measures the time consumed by the thread switching in a cooperative thread environment. It is composed by five threads running to completion and releasing control in a round-robin fashion. The behaviour of each simple thread comprises the increment of a counter and a thread relinquish to pass execution;
- **Preemptive Context Switch**: Evaluates the time consumed by the switch of threads in a preemptive environment. This test is composed by five threads with different priorities. The test starts will all the threads in suspended mode, except the thread with the lowest priority that is never suspended. The behaviour of each thread includes the increment of a counter, the resume of the next thread with higher priority, and finalizes with the thread suspending itself. Once the highest priority thread runs, it suspends itself and the lowest priority thread automatically resumes;
- **Interrupt Handling**: Measures the time consumed by an interrupt since it is generated until a new thread is scheduled. This test is composed by a thread enforcing the generation of an interrupt, and an ISR incrementing a counter and posting a semaphore. This semaphore is taken by the thread that generates the interrupt, guarantying the synchronization between the generation and the response to the interrupt. The generation of the interrupt is performed through GPIO pins;
- **Interrupt Preemption**: Measures the time consumed when a thread is preempted by an ISR. This test is similar to the previous one, but instead of post a semaphore, a high priority thread is resumed on the ISR. The thread with higher priority, consists of incrementing a counter and next going to

the suspend state, resuming the execution flow to the thread that triggered the interrupt;

- **Message Processing**: Aims to measure the overhead caused by the task of reading and writing a message from a message queue. To do so, is used the IPC message queue provided by RIOT-OS. This test is composed by a single thread that sends a message to the messaging queue and waits until the same message is retrieved. For each send/receive cycle the thread increments a counter;

- **Semaphore Processing**: Measures the time consumed by the usage of semaphores. To perform this test, it was used a lightweight library provided by RIOT-OS that implements semaphores, the `sema` library. This test consists of a thread that gets and release a semaphore, incrementing a counter each time the cycle is completed;

- **Memory allocation**: Aims to measure the time consumed by the dynamic allocation of memory. This test consists of acquiring and releasing memory, incrementing a counter on each successful cycle. To perform this test was used the functions `malloc` and `free` from the C standard library.

To transmit the results of the benchmarks to the user is provided in Thread-Metrics a thread that reports, in defined periods of time, the value of the variable used to count the number of cycles completed. For this thread is associated the high priority possible in RIOT-OS. This thread is let to sleep for a fixed period, in this case 30 seconds, with the usage of xtimer functionalities. Every 30 seconds the thread resumes, saves and resets the counter variable value. After 5 cycles, the test stops, and the results are printed through the UART interface.

3.4.2 Thread-Metric Evaluation

The results obtained from the Thread-Metric Benchmark Suite allow to evaluate the system performance without the interference of networks related tasks, and later evaluate how much the OS performance is affected when it must preform network tasks. Such results can be used as an indicator of OS availability to perform other tasks, rather than network-related ones. Being the Thread-Metric results proportional to the system availability, high scores denote a high system availability while a low score indicates that the system has less availability. In order to evaluate the both solutions under the desired conditions, i.e., at the IoT network edge with packets from the surrounding nodes, it was implemented the topology presented in Figure 3.6.

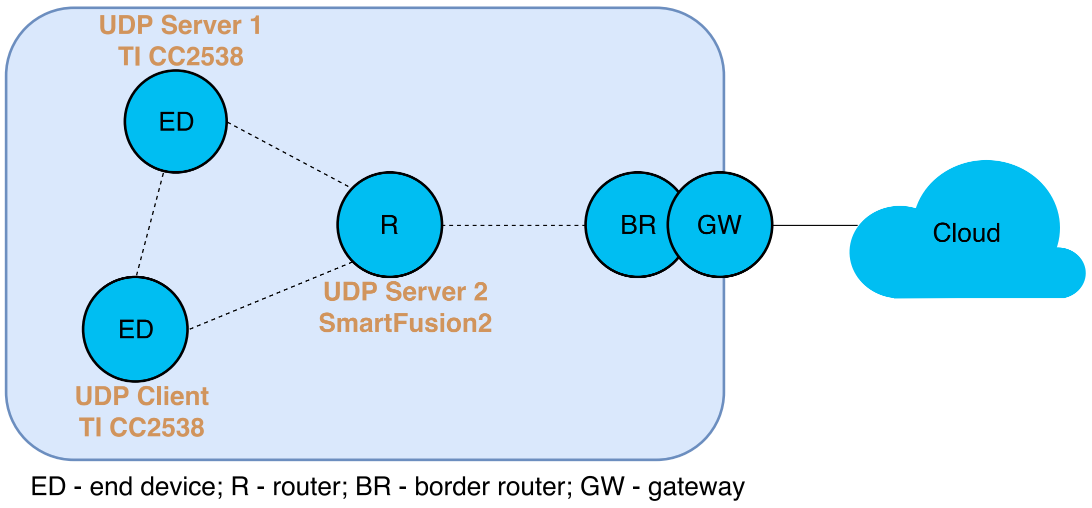

Figure 3.6: Evaluation network topology.

To perform the evaluation was created a network with IPv6 support, provided by 6LoWPAN, with three nodes exchanging UDP packets. In this topology two nodes run a UDP Server and one node hosts the Client. All the nodes are in the range of each other, not happening packet loss due to their distance. Furthermore, the Client exchanges data with booth servers at his maximum packet sending rate (PSR), in this case, it was achieved 250 packets per second. The UDP Server 1 and the Client run on a TI CC2538 SoC while the UDP Server 2 is hosted by the CUTE mote. All nodes run RIOT-OS, with their respective applications, emphasizing that booth servers run the same application. All Thread-Metrics tests were executed in both servers under three conditions: (i) the network is idle and there are no packets being exchanged over the network; (ii) Client 1 exchanges data with Server 1 at its maximum rate; and (iii) Client exchanges data with Server 2 at its maximum rate.

The Server that receives data intended for him, for each received packet, processes and accepts at the MAC layer the packet, forwarding it to upper layers, until delivered in the application where is discarded. By other side, the Server which data is not intended for him, processes and discards all packets at the MAC layer. Thread-Metric results will translate in how much overhead is caused to RIOT-OS when the network is highly saturated, and the received packets intended to be accepted and discarded. Furthermore, with the achieved results it will be possible to evaluate if the heterogeneous solutions can create advantages over the COTS solution. This experiment, despite using a simple topology, represents a real network scenario where all nodes exchange data between them.

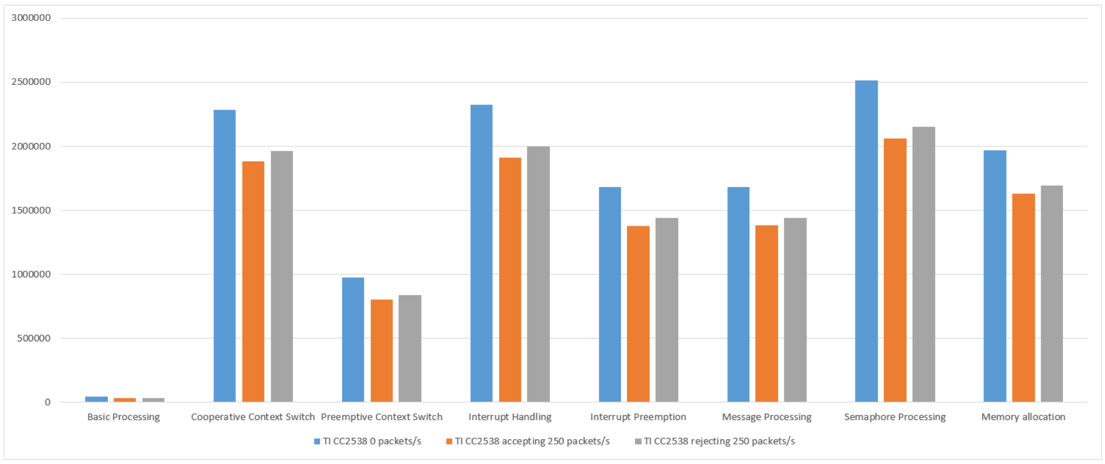

Figure 3.7: TI CC2538 Thread-Metric results.

Figure 3.7 presents the results obtained from the UDP Server 1, that is hosted by the TI CC2538 platform, for the three test case scenarios. The Thread-Metrics results show that the overhead caused by the network tasks decreases the system availability by 18.62%, on average, when all the received packets are meant to be accepted by the node. Furthermore, the system availability decreases 15.08%, when all the packets in the medium are received and discarded at layer 2 of the network stack. The system availability ratio between accept and discard packets is 4.18%, which means that the task of rejecting a packet, which can be considered unnecessary processing still causes deep impact on RIOT-OS execution.

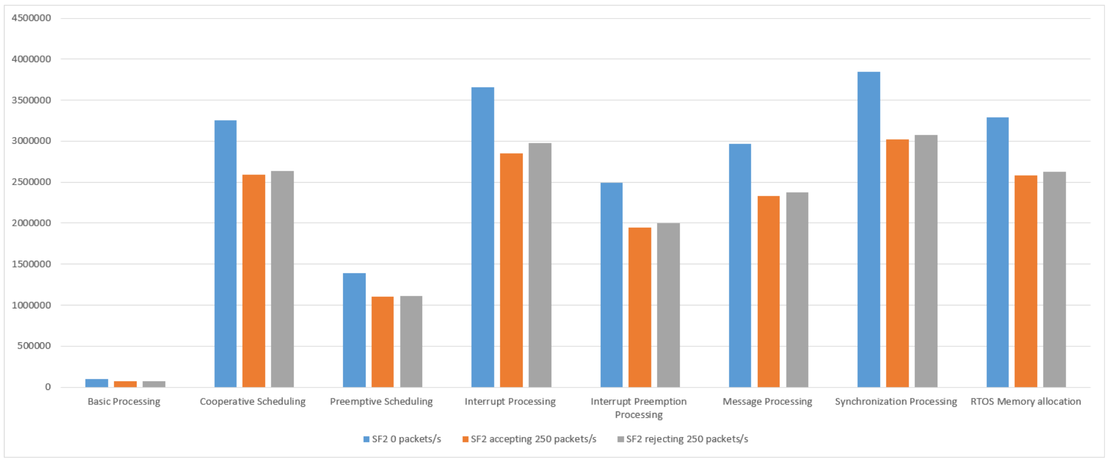

Figure 3.8: SmartFusion2 Thread-Metric results.

The Thread-Metric results obtained on UDP Server 2, which is hosted by RIOT-OS running over the CUTE mote, is presented in Figure 3.8. Similarly, to the previous results, the performance decrease is 21.33%, on average, when the received packets are meant to be accepted and 19.56%, on average, when the packets are processed and discarded at the MAC layer of the network stack. The ratio between accept and discard packets in the system availability is 2.19%, which also detonates that the needless processing of rejecting a packet still impacts largely the normal system execution.

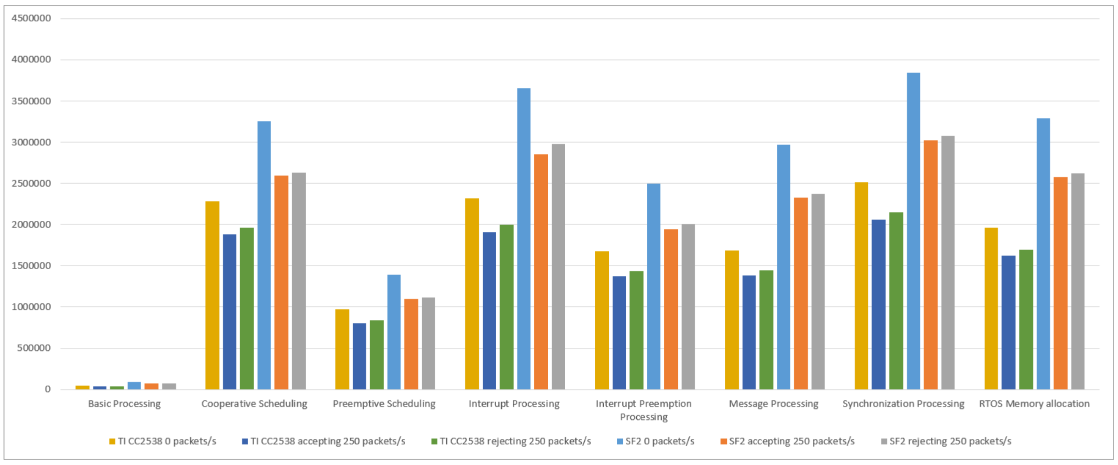

Figure 3.9: SmartFusion2 and TI CC2538 Thread-Metric results.

Figure 3.9 present the results obtained from both solutions side-by-side. Still, the solutions use different hardware platforms, a COTS and an FPSoC, the CUTE mote presents more 37% performance than the TI CC2538 when no packets are present in the medium. Furthermore, has 35% higher performance when the packets are processed and accepted, and 33.56% when the packets are rejected and discarded at the MAC layer. These results reflect the different micro architectures and hardware platforms, such as, memory subsystem, data buses, clock speed, it is 36% higher on the SmartFusion2, access to the radio interface, etc.

3.5 Conclusion

This Chapter analysed the advantages of FPGA-enabled devices in the IoT ecosystem. It was presented the CUTE mote project, which uses a radio transceiver and an FPSoC device to deploy custom accelerators. Then, was justified the choice of RIOT-OS to endow the CUTE mote with extended features and described the changes needed to use the IAR compiler. Furthermore, was exposed the heterogeneous architecture that allow the deployment of RIOT-OS in in the CUTE mote, supported by the Microsemi's SmartFusion2, and the drivers developed to allow the integration of the OS. Finally, it was presented the Thread-Metrics results on two platforms, the TI CC2538, used for validation and as a basis of comparison, and CUTE mote, supported by the Microsemi's SoC. The results point that CUTE mote provide better results, suggesting better performance. However, this happen due to the microarchitectural differences and different hardware of the platforms. Nevertheless, it is important to have a characterization of both systems since the CUTE mote suggests the deployment of hardware accelerators for network-related tasks and allow the further deployment of application-related tasks.

Next Chapter will present the MLA, an accelerator designed to support tasks related to the MAC layer of the standard IoT stack. Such accelerator was already provided in CUTE mote, however, several points of improvement were detected and implemented.

Chapter 4

Accelerators Integration

"Scientists study the world as it is; engineers create the world that has never been."

—Theodore von Kármán

The 6LoWPAN adaptation layer enables the integration of WSN-based devices with an IoT network, allowing their communication trough IPv6 and, consequently, their seamlessly connection to the Internet. Furthermore, allows the WSN-enabled devices to achieve the connectivity requirements stated in Section 1.5 of Chapter 1. However, the connection of such resource-constrained devices, due to the nature of the IEEE 802.15.4 specifications, low bandwidth and low data rates transmissions, in dense or highly active networks the device's performance tends to decrease, as seen in Chapter 3. As previously seen, CUTE mote [113] tackles such issues by allowing the deployment of network-related accelerators for the MAC [122] and the network layers [121] on the RCU. This Chapter aims to integrate RIOT-OS, which support to CUTE mote was explained in Chapter 3, with the accelerators already provided by the solution [121,122], mainly the MAC layer accelerator, which its functionality was deeply extended and improved on the development of this dissertation.

Section 4.1 starts this Chapter with an introduction of the work developed in the in-house project CUTE mote [113], followed in Section 4.2 by the presentation of the refactoring process made over the hardware accelerator. Section 4.3 presents the work developed to integrate the XIoT in RIOT-OS while Section 4.4 presents and discusses the results obtained from the integration of the MAC layer accelerator with RIOT-OS. Finally, Section 4.5 Finally, concludes this Chapter.

4.1 Acceleration on Low-End IoT Devices

The connection of LLN and WSN-based devices to the Internet requires an IoT-enabled communication stack, allowing to fulfil the connectivity requirements presented in Section 1.5 of Chapter 1. To integrate these devices seamlessly on the Internet, it is required to endow the motes with enough processing capabilities to support overheads caused by the IPv6-compliant network stack. However, provide such processing capabilities in low-cost and low-power devices is not straightforward, since the increase of performance consequently increases power-consumption and unit cost.

The wireless low-end devices use mainly IEEE 802.15.4-complaint radios that due to their characteristics, all the communication enabled devices receive and decode all the packages transmitted in the network that use the same communication standard, channel and are inside the communication range, along with receiving interference present on the network. For every frame received by the radio interface, the low-end device must allocate resources that allow to accommodate the packet and process through all networks layers until it reaches its destination layer. Such processing includes, among other tasks, the verification of network addresses in order to accept or discard the packet according to system rules.

Previously, in Chapter 3 were analysed the benefits of heterogeneous solutions in IoT low-end devices and specified that an RCU unit plays a key role in the development of heterogeneous architectures that aim to integrate customized accelerators. The application of hardware accelerators in WSN is not new and has been done before, however, the application of such accelerators traditionally was applied to interface sensors, security related functionalities, data aggregation, compression/decompression and power control mechanisms. The solutions developed targeted a specific application or scenario, which restricts a wide deployment. However, the implementation of hardware-based network accelerators and filters bring portability advantages and allow their widely deployment and integration in new radios because the protocols used in these radios are standard and widely deployed in WSN and IoT networks. The implementation MAC-related features, such as frame filtering and error correction mechanisms, have already been performed by recent RF transceivers. However, features related to the network layer, such as IP header compression/decompression IP address filtering, UDP port verification, etc., are still not supported by any device.

Due to the standardized communication protocols, the hardware accelerators deployed in CUTE mote [113], in the form of soft-peripherals, can integrate any system that follows the same network stack. These systems can include heterogeneous architectures or the deployment on silicon integrating future ASIC radio transceivers with MAC or network filtering capabilities.

4.2 XIoT Hardware Accelerator

The CUTE mote presented in [113] follows a heterogeneous architecture that targets the low-end and resource constrained IoT devices at the network edge, deploying a hardware-based accelerator endowed with network filters and other network-related functionalities in an RCU unit. The usage of such architecture and accelerators as proved to be able to achieve up to 42% of system availability increase in the worst-case scenario and when the packets are not intended to be delivered to the mote performing the evaluation. However, before the integration of the MAC accelerator with RIOT-OS, various modifications were performed to solve problems previously identified, provide extra functionalities, better portability and usability.

The XIoT hardware-based accelerator is an improved version of the MAC layer accelerator developed in CUTE mote and aims to provide:

- An agnostic implementation of the hardware accelerator empowering not only the portability of the network filters, but also the portability of the hardware accelerator;
- Full-duplex communication to the radio interface through the hardware accelerator, allowing the exchange of network frames and commands with the radio peripheral;
- A hardware implementation of the SPI software driver, aiming to decrease the integration complexity in an OS or bare-metal application and reducing the overhead caused by the SPI communication;
- A memory mapped hardware peripheral, accessible through a standard AMBA communication interface, the APB3 protocol;
- A portable and well documented API allowing fast and easy integration of the hardware accelerator in new systems;
- Prepare the accelerator to allow easy integration with upper layer functionalities, e.g., network, transport and application layers.

The development of XIoT required a new finite state machine (FSM) to accommodate all the previously proposed functionalities. In Figure 4.1 is presented the FSM of the hardware accelerator that is composed of seven finite states, with each of them being described below:

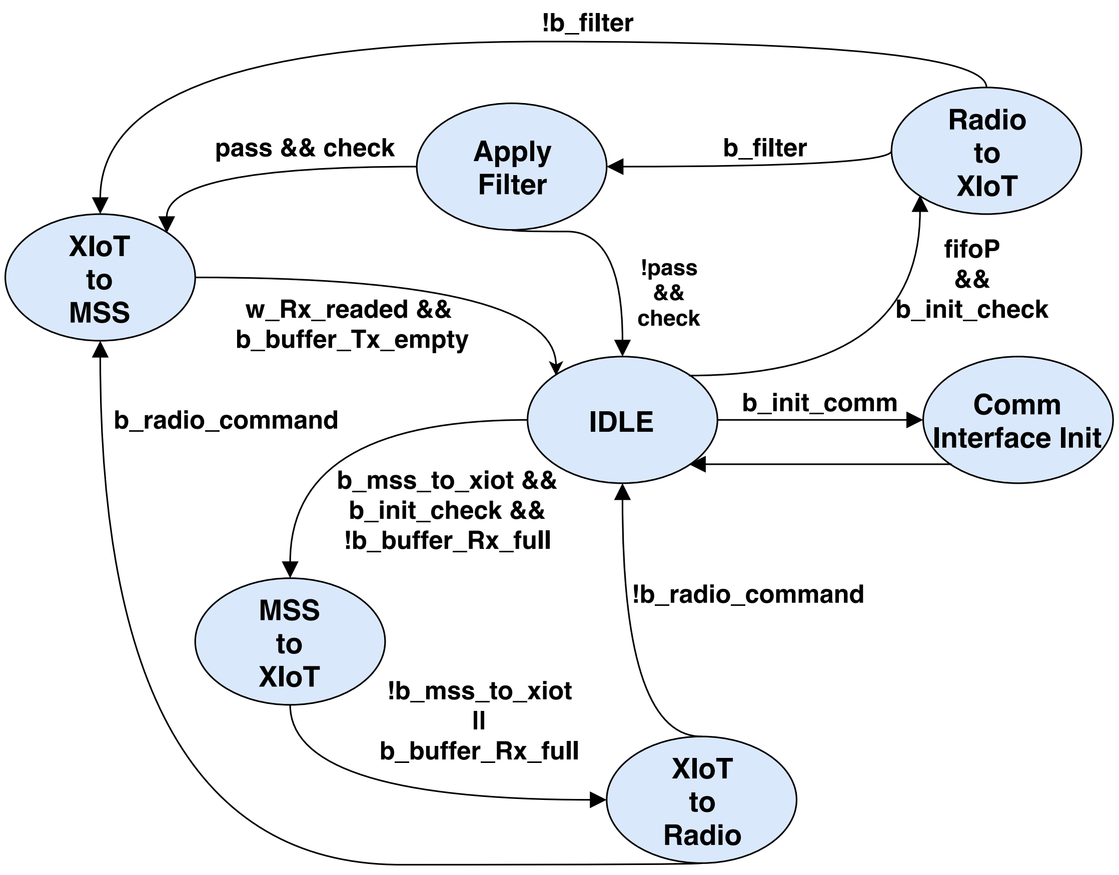

Figure 4.1: XIoT main FSM.

- **Idle**: In this state the hardware will be waiting an event that creates the conditions to switch state. After a reset, this state can only switch to the Comm Interface Init;

- **Comm Interface Init**: Initializes the SPI interface to communicate with the IEEE 802.15.4 radio transceiver. This state is executed when by software the bit `INIT_INTERFACE` from the `CTRL1_REG` register is set. Once the initialization is done, the bit `FSM_INIT` is set by the hardware and the FSM changes to idle state;

- **Radio to XIoT**: This state is triggered by a rising edge of `FIFOP` signal, a physical output from the radio transceiver that signalizes the reception of a frame. Therefore, this state performs the needed actions to read a frame from the radio's FIFO and make it available for further processing by the FSM. If the filtering functionalities are enabled, as defined by the IEEE 802.15.4 standard, this state changes to apply filter state, otherwise it goes to the XIoT to MSS state;

- **Apply Filter**: Analyses a frame received by the radio to apply the filters according the IEEE 802.15.4 standard until the third level of filtering. If the received frame passes the defined criteria, the FSM switches to the XIoT to MSS state to make the received frame available to the MSS. Otherwise, switches immediately to the idle state, discarding the received frame without interrupting the MSS;

- **XIoT to MSS**: This state makes data available in a FIFO to be read by the MSS. When a frame is made available to the MSS an interrupt is generated in the APB3 bus to notify the OS. In case that a command request has been made, only the `RX` bit from the `DATACR_REG` register will be set indicating the availability of data in the `RXDATA_REG` register. The FSM only exits this state when the MSS executes the reading operation, avoiding this way unnecessary calls to the MSS;

- **MSS to XIoT**: Responsible to receive data coming from the MSS. To enter this state the MSS should enable the communication by setting the `MSS_TO_XIOT` bit in the `CTRL2_REG` register and specifying how many bytes will be written in the `TXDATA_REG` through the `TX_SIZE_REG`. The FSM exits the state when the MSS clears the `MSS_TO_XIOT` bit, disabling this way the communication;

- **XIoT to Radio**: In this state, data previously received in a FIFO is transmitted to the radio through the SPI interface. After the transmission, if the bit `RADIO_COMMAND` in the register `CTRL2_REG` is enabled, a radio command was transmitted and a response may be required to the MSS. Consequently, the FSM switches the XIoT to MSS state. However, if the same bit was not enabled, the FSM switches to idle state.

The XIoT soft-peripheral is a memory mapped AMBA compliant peripheral, being provided several registers to perform its access. Figure 4.2 presents the peripheral registers alongside with their addresses and functionality. The registers `RX_SIZE_REG`, `TX_SIZE_REG`, `RXDATA_REG` and `TXDATA_REG` are not bit oriented and are related with the transfer of bytes between the MSS and the XIoT. The data handling registers include the `RX_SIZE_REG` that indicates how many bytes are going to be transfer to the XIoT through the `RXDATA_REG` register, being written by the MSS before start sending data to the XIoT. The `TX_SIZE_REG` is written by the XIoT to indicate to the MSS how many bytes should be read from the `TXDATA_REG`.

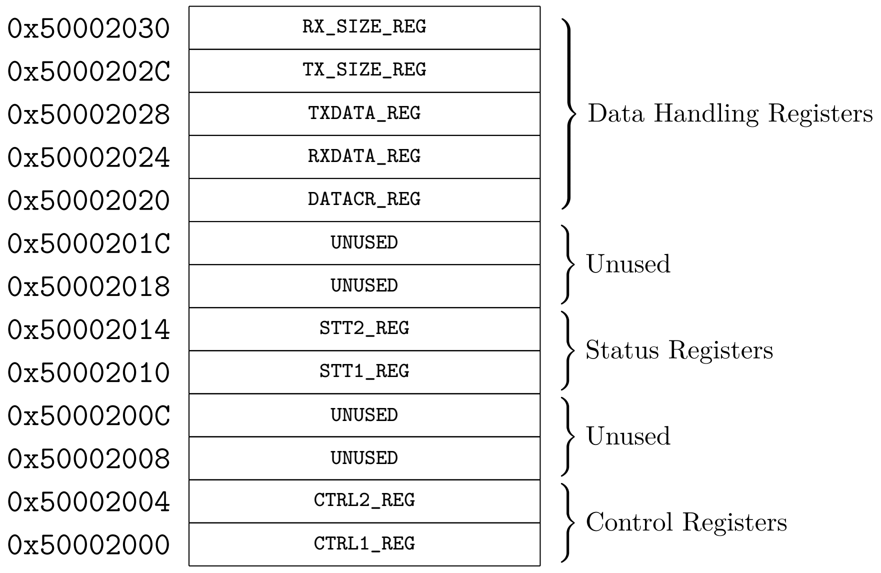

Figure 4.2: XIoT peripheral memory address space.

The bit-fields of `CTRL1_REG` illustrated in Figure 4.3 are used to configure the XIoT. The `RESET` bit-field allows, as the name indicates, to reset the hardware accelerator. By its side, the `ENABLE` bit-field allows enable or disable the XIoT. The `INIT_INTERFACE` bit-field is set to launch the SPI interface initialization, whereas, the `INIT_INTERRUPT` bit-field configures the XIoT interrupt. Furthermore, the `FILTERS_EN` enables the usage of filtering functionalities when it is set and the `CLEAR_INT` bit allows to clear the interrupt generated by the XIoT when a frame is available. For last, the bit `CCA_EN` enables the verification of the Clear Channel Assessment (CCA) pin by hardware.

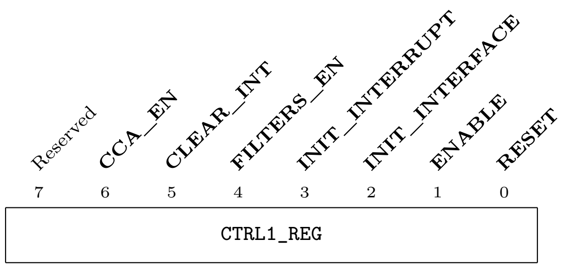

Figure 4.3: `CTRL1_REG` register field.

The `CTRL2_REG` register, presented in Figure 4.4, is composed by five enabled bits that indicate the flow of information in the XIoT. The bit-field `RADIO_COMMAND` is used to indicate when data written in the `RXDATA_REG` by the MSS is to be sent to the radio as a command. The hardware accelerator will set the bit-field `RADIO_TO_XIOT` when performing a read operation from the radio FIFO in response to a positive-edge on the `fifop` GPIO pin. In the opposite direction, the `XIOT_TO_RADIO` bit-field signals a flow of data from the XIoT to the radio through the SPI interface, e.g., making a frame available to the radio to be sent through the wireless interface. The `XIOT_TO_MSS` should be set by the MSS when the last wants to read data made available by the first. Finally, the `MSS_TO_XIOT` is set by the MSS to start transferring data to the XIoT through the `REG_BYTE_RX`.

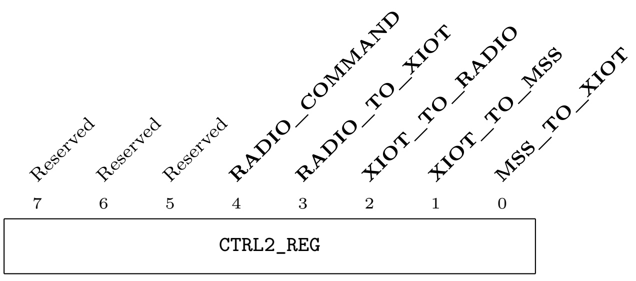

Figure 4.4: `CTRL2_REG` register field.

The bit-fields of `STT1_REG`, as shown in Figure 4.5, provide information about the internal state of XIoT. The `FSM_INIT` bit-field indicates when the initialization of the XIoT is completed. By its side, the `FSM_MAIN` bits allow to acquire the state of the FSM. Finally, the bit `BUSY` is cleared when the FSM is in idle state, indicating to the MSS that the state of the FSM can be changed.

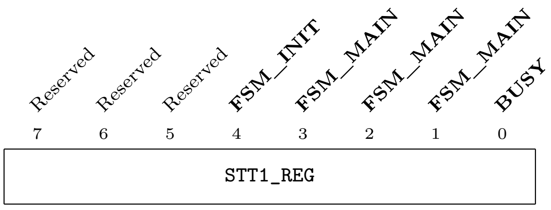

Figure 4.5: `STT1_REG` register field.

The `STT2_REG` register, presented in Figure 4.6, indicates the state of XIoT buffers. It is composed by two bits the `BUFFER_RX_FULL` and the `BUFFER_TX_EMPTY`. The `BUFFER_RX_FULL` is set when the FIFO that receives data from the MSS and is accessible through the `RXDATA_REG` is full. By the other hand, the `BUFFER_TX_EMPTY` is set when no more data is available in the FIFO to be read through the `TXDATA_REG`.

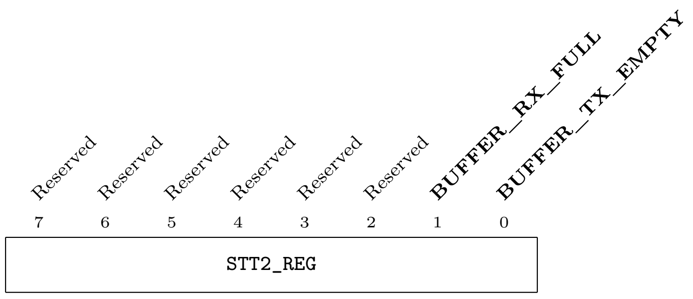

Figure 4.6: `STT2_REG` register field.

From the bit-fields of `DATACR_REG`, presented in Figure 4.7, only one bit is used. The `RX` bit is set when a response from a command requiring a read operation over the wireless radio is complete and the data made available to the MSS in the RX_DATA_REG.

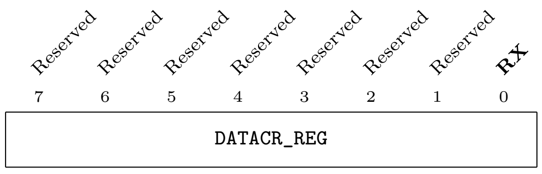

Figure 4.7: `DATACR_REG` register field.

4.3 XIoT deployment in CUTE mote

The integration of XIoT with RIOT-OS requires both the modification of the hardware presented in Chapter 3, and the development of a device driver to integrate XIoT seamlessly with RIOT-OS. Architecturally, XIoT is accommodated in CUTE mote by replacing its previous version as described in [113] and integrating all the processing and filtering features previously supported.

To integrate XIoT in the heterogeneous architecture previously presented in Figure 3.4 from Chapter 3, it was necessary to deploy XIoT between the MSS and the CoreSPI, keeping the SPI bus to interface the radio transceiver and to connect other signals used by this device. Due to the full emulation of the SPI driver in hardware, no direct interaction happens between the MSS and the radio, simplifying this way the firmware development, enabling easy portability and increasing performance through the offloading of software tasks to hardware. Furthermore,

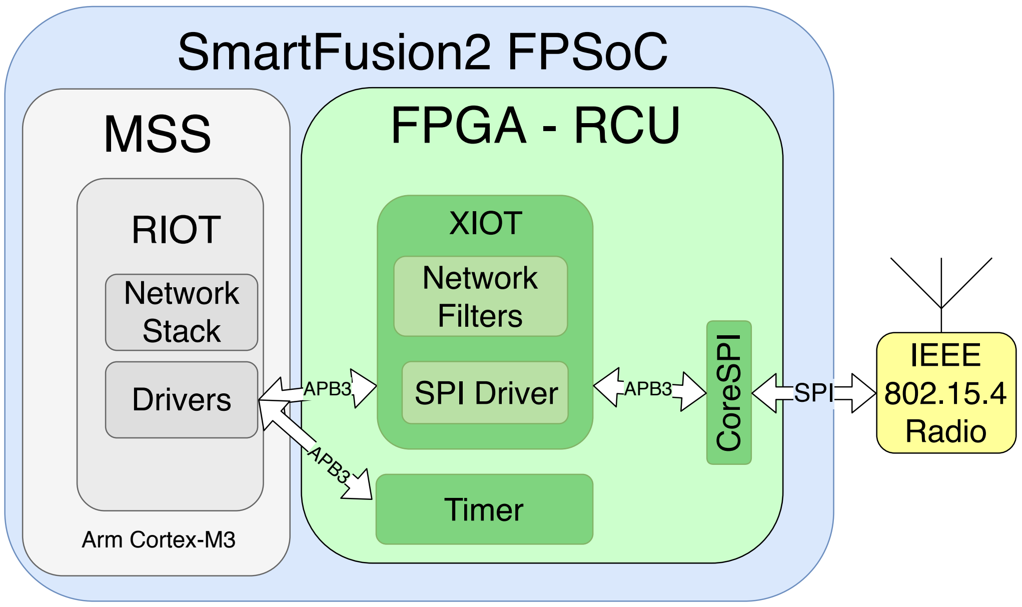

Figure 4.8: Architecture of the heterogeneous solution with XIoT.

to notify the OS about a frame to be forward to upper layers of the network stack, it was configured the APB3 interrupt port available in the MSS subsystem.

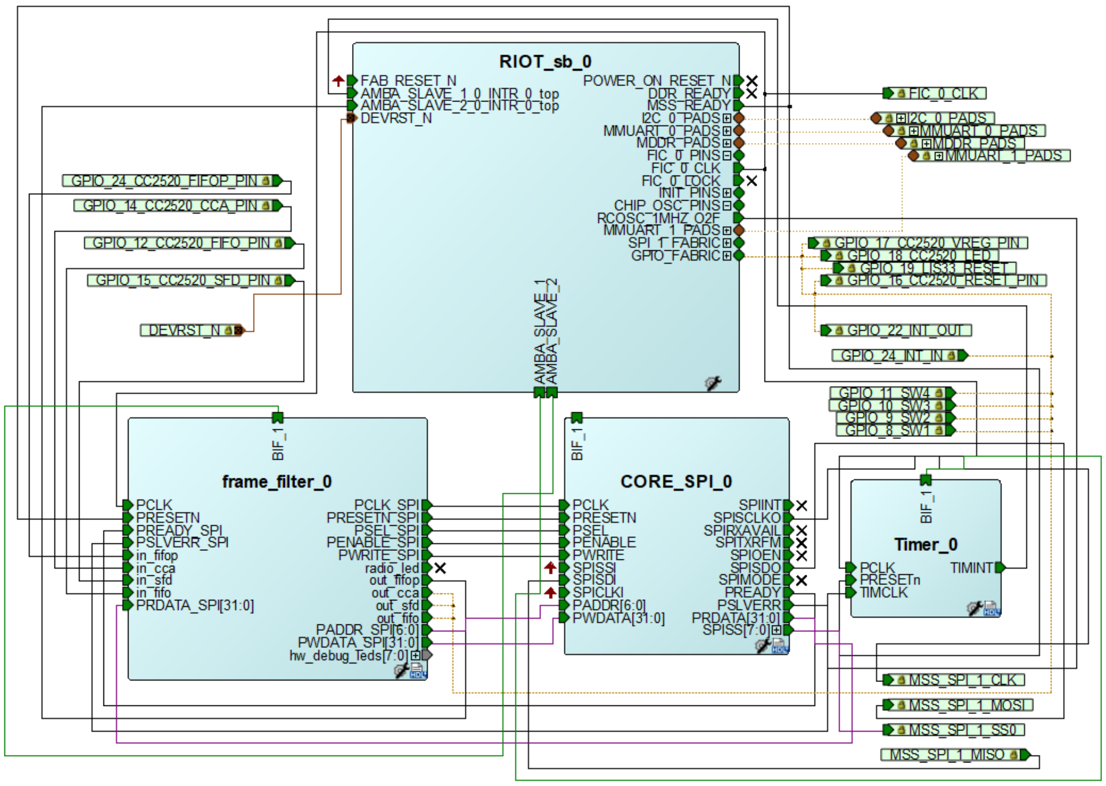

Figure 4.9: Heterogeneous architecture with XIoT integration layout.

The architecture developed, present in Figure 4.8, is built in such a way that the MSS can be let to sleep whenever no actions need to be performed by the OS, being only interrupted by the RCU when a frame is intended to be forward to the network stack.

The filters integrated in the XIoT include the PAN, destination and source address, verification, and a duplicate frame detection mechanism. These filters were previously developed in the work presented in [113] and were integrated in XIoT without modifications, proving the stated by the author, that due to the standard network communication protocols the network filters could be integrated in other systems without changes.

4.3.1 XIoT API

To allow the integration of XIoT accelerator into RIOT-OS, a well-documented and easy portable device driver was developed. The XIoT hardware accelerator is a memory mapped peripheral, consequently, the development of an API required the access to low-level registers.

Listing 4.1: Microsemi's HAL set 8-bit register macro.

```
/***************************************************************************//**
 * The macro HAL_set_8bit_reg() allows writing a 8 bits wide register.
 *
 * BASE_ADDR:   A variable of type addr_t specifying the base address of the
 *              peripheral containing the register.
 * REG_NAME:    A string identifying the register to write. These strings are
 *              specified in a header file associated with the peripheral.
 * VALUE:       A variable of type uint_fast8_t containing the value to write.
 */
#define HAL_set_8bit_reg(BASE_ADDR, REG_NAME, VALUE) \
          (HW_set_8bit_reg( ((BASE_ADDR) + (REG_NAME##_REG_OFFSET)), (VALUE) ))
```

The access to those registers is done using the macro library provided by the Microsemi's hardware abstraction layer (HAL). Listening 4.1 presents an example of the macro that allows to modify all the bit-fields of an 8-bit register. As it is possible to identify, the `REG_NAME##_REG_OFFSET` should be provided with a specific format, compliant with the macro library. All the bit-fields of the XIoT registers are defined in the file `xiot_regs.h`. Listening 4.2 presents a code snippet of that file, which is the file that contains the definitions for the register `CTRL1_REG` according to the Microsemi's low-level HAL format.

The XIoT's API implements several functions to access the peripheral, and two extra functions to guaranty exclusive access to a XIoT device, avoiding concurrency issues. A function for debug proposes is also provided in the XIoT driver for RIOT-OS. All the API functions and the extra ones needed to integrate in RIOT-OS are presented above:

Listing 4.2: Register definition of `CTRL1_REG` required to use Microsemi's HAL.

```
 /*******************************************************************************
 * Control register 1:
 *------------------------------------------------------------------------------
 */
#define CTRL1_REG_OFFSET                    0x00u
                                       /* Reset the state machine */
#define CTRL1_RESET_OFFSET                  0x00u
#define CTRL1_RESET_MASK                    0x01u
#define CTRL1_RESET_SHIFT                   0x00
                                       /* Enable state machine */
#define CTRL1_ENABLE_OFFSET                 0x00u
#define CTRL1_ENABLE_MASK                   0x02u
#define CTRL1_ENABLE_SHIFT                  0x01
                                       /* Init the Communication Interface */
#define CTRL1_INIT_INTERFACE_OFFSET         0x00u
#define CTRL1_INIT_INTERFACE_MASK           0x04u
#define CTRL1_INIT_INTERFACE_SHIFT          0x02
                                       /* Transfer message from XIOT to MSS*/
#define CTRL1_INIT_INTERRUPT_OFFSET         0x00u
#define CTRL1_INIT_INTERRUPT_MASK           0x08u
#define CTRL1_INIT_INTERRUPT_SHIFT          0x03
                                        /* Enable the application of network filters*/
#define CTRL1_FILTERS_EN_OFFSET             0x00u
#define CTRL1_FILTERS_EN_MASK               0x10u
#define CTRL1_FILTERS_EN_RADIO_SHIFT        0x04
                                           /* Clear the fifop interrupt */
#define CTRL1_CLEAR_INT_OFFSET             0x00u
#define CTRL1_CLEAR_INT_MASK               0x20u
#define CTRL1_CLEAR_INT_SHIFT              0x05
                                        /* Enable the usage of CCA */
#define CTRL1_ENABLE_CCA_OFFSET             0x00u
#define CTRL1_ENABLE_CCA_MASK               0x20u
#define CTRL1_ENABLE_CCA_SHIFT              0x05
```

- **`xiot_init`**: Performs the initialization of the XIoT accelerator, and this function should be called once by each XIoT instance before its first utilization. This function receives a variable of type `xiot_t`, that corresponds to a position in an array containing the addresses of the XIoT hardware accelerators. In this case, it will be only used a XIoT accelerator but the driver is prepared to accommodate various instances;

- **`xiot_init_int`**: Configures an interrupt in the vector table and enables the generation of interrupts by the XIoT. The interrupt is generated once a valid frame is ready to be read by the MSS;

- **`xiot_irq_enable and xiot_irq_disable`**: Allows to enable or disable the generation of interrupts from a specific XIoT instance, by passing as argument the `xiot_t` identifier of the instance;

- **`xiot_enable_filtering` and `xiot_disable_filtering`**: These functions allow enabling and disabling the filtering functionalities in the XIoT accelerator, used to filter a packet received in the XIoT before being forwarded to the MSS;

- **`xiot_write`**: Allows transfer a single or multiple bytes to the XIoT that are intended to be transmitted through the network interface. This function receives a variable of type `xiot_t` to identify the XIoT instance, the address of the buffer that holds the bytes to be transmitted and the number of bytes to be sent;

- **`xiot_write_command`**: Enables the transfer of commands from the MSS to the XIoT, allowing the configuration or reading of registers from the radio interface. The parameters received by this function are the `xiot_t` identifier, two addresses of buffers to send and receive the bytes exchanged, and the respective sizes to be transmitted;

- **`xiot_read`**: This function is intended to be used in response to an interrupt generated by the XIoT, informing the availability of network packet to be read by the MSS. This function receives as parameters the identifier of the respective XIoT instance and the address of a buffer to store the received information. The value returned by this function indicates the number of bytes stored in the buffer;

- **`xiot_enable_cca`**: Enables the verification of the CCA pin by the hardware accelerator. This pin is used to verify the availability of the channel to perform transmissions;

- **`xiot_fsm_state`**: Allows to verify the state of the main FSM presented in Figure 4.1, by providing the identifier of the type `xiot_t` to the instance. Can be used to debug proposes, to handle errors or exceptions;

- **`xiot_acquire` and `xiot_release`**: This pair of functions are used to guarantee the exclusive access to the XIoT, and they should be called always as a pair, being called the `XIoT_acquire` before any action and `xiot_release` in the end of the operation. To avoid the exclusive access to the device a software mutex was created to guaranty the exclusive access to the hardware resources, thus these functions allows the access to that mutex.

4.3.2 XIoT Integration in RIOT-OS

Using the API previously presented, a device driver was developed to allow the usage of the XIoT hardware accelerator in RIOT-OS. This driver initializes an instance of the XIoT structure for each device incorporated, allowing this way incorporate several XIoT accelerators with the same driver.

Listing 4.3: XIoT device driver arrays.

```
/*Mutex instantiation for the XIOT Mutex*/
static mutex_t locks[XIOT_NUMOF];

/*Array of XIOT intances*/
static xiot_instance_t * intances[XIOT_NUMOF];

/*Array of hardware addresses*/
static uint32_t xiot_addr[XIOT_NUMOF] = {XIOT_ADDR};
```

Three arrays where include in the driver, comprehending an array of mutexes to guaranty mutual exclusion on access to a XIoT peripheral, an array to accommodate pointers to the instances of the hardware accelerators and finally an array containing the base addresses of the peripherals. These arrays are presented in the Listening 4.3. With the usage of such arrays, each XIoT device can be identified through an order of integration number, for example when only a device is integrated in the system is identified through the number zero.

Due to the granularity and well-defined interfaces of RIOT-OS, the work developed to integrate the XIoT driver was only performed at the hardware dependent level, focusing in the CC2520 peripheral driver previously developed. To establish a new software module that corresponds to the newly created interface, it was defined the `MODULE_XIOT`. This definition allows to initiate the XIoT module trough the `auto_init` functionality provided by RIOT-OS. Furthermore, to allow the activation and deactivation of the filtering capabilities at layer 2 by hardware, it was used the `MODULE_L2FILTER` which enabling the filtering capabilities by hardware when it is not enabled by software. Listening 4.4 illustrates such functionality.

Listing 4.4: Code snippet from the function `_init` from the cc2520 peripheral driver.

```
#ifndef MODULE_L2FILTER
    xiot_enable_filtering(XIOT_BUS);
#endif
```

From the driver developed for CC2520, it was only necessary to modify the `cc2520_internal.c` file in order to make it compliant with the XIoT accelerator. The function's implementation was changed to instead of call the SPI driver functions call the ones from the XIoT driver, maintaining the API of the CC2520 driver.

Listing 4.5: `cc2520_strobe` function from the cc2520 peripheral driver.

```c

uint8_t cc2520_strobe( cc2520_t *dev, const uint8_t command)
{
    uint8_t send_buffer[1];
    send_buffer[0] = command;
    uint8_t receive_buffer[1];

    xiot_acquire(XIOT_BUS);
    xiot_write_command(XIOT_BUS, send_buffer, receive_buffer, 1, 1);
    xiot_release(XIOT_BUS);

    return receive_buffer[0];
}
```

In Listening 4.5 it is represented one of the function's implementation of the `cc2520_internal.c`, where can be seen the implementation of the `cc2520_strobe` using the XIoT API.

Due to the packet send and receive being performed by XIOT, it was necessary to change the packet transfer between RIOT-OS and XIoT. Those changes allow read from the hardware peripheral when a frame is ready to be transferred to the MSS, instead of reading the radio FIFO through the SPI interface. The reception is made through a call to the `xiot_read` after an interruption generated by the `fifop` signal. Once more, was used the definition of the XIoT module to distinguish between the use of the SPI or XIoT interface. This implementation is presented in the Listening 4.6

Listing 4.6: Code snippet from the function `cc2520_rx` from the cc2520 peripheral driver.

```c
    #ifdef MODULE_XIOT
    else {

        /* read fifo contents and length*/
        len = xiot_read(XIOT_BUS, buf);

        /* finally flush the FIFO */
        cc2520_strobe(dev, CC2520_INS_SFLUSHRX);
        cc2520_strobe(dev, CC2520_INS_SFLUSHRX);
    }
    #endif
```

4.4 Evaluation

To evaluate the heterogeneous architecture with the integration of the XIoT hardware accelerator, it was used the Tread-Metric Benchmark to evaluate how RIOT-OS can benefit from the XIoT functionalities, and used the Libero tool to perform a usage resource estimation.

4.4.1 Thread-Metrics Evaluation

The evaluation performed over the CUTE mote architecture empowered by the XIoT hardware accelerator were the same used in Section 3.4 of Chapter 3. The Thread-Metric Benchmark Suite was performed under the same conditions and using the network topology of Figure 3.6, maintaining the same test scenario to allow further comparisons.

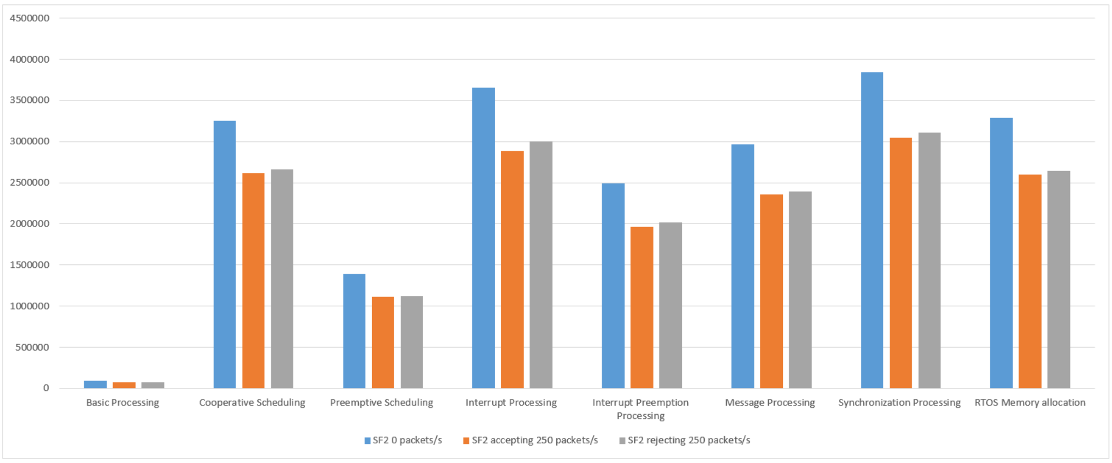

Figure 4.10: SmartFusion2 Thread-Metric results with XIoT accelerator and filter disabled.

Figure 4.10 presents the results from the benchmark performed with the XIoT enabled and all the network related tasks still being performed by software. The goal of this test is to evaluate how the integration of XIoT could affect the RIOT performance. Consequently, the reception of packets from the radio is performed by the XIoT, but the filtering process is performed by software in the MCU. The obtained results are similar to the ones obtained with the CUTE mote architecture without acceleration and that are presented in Figure 3.8. However, is noticed an increase of performance by 0.94% when receiving packets, induced by the implementation of the SPI driver in hardware.

Figure 4.11 presents the results obtained with the XIoT instantiated and the hardware network filters enabled. In this scenario is possible to achieve an increase of performance of 1.28% when receiving packets, in comparison with the architecture based in the SmartFusion2 and presented in Figure 3.5. Furthermore, allows

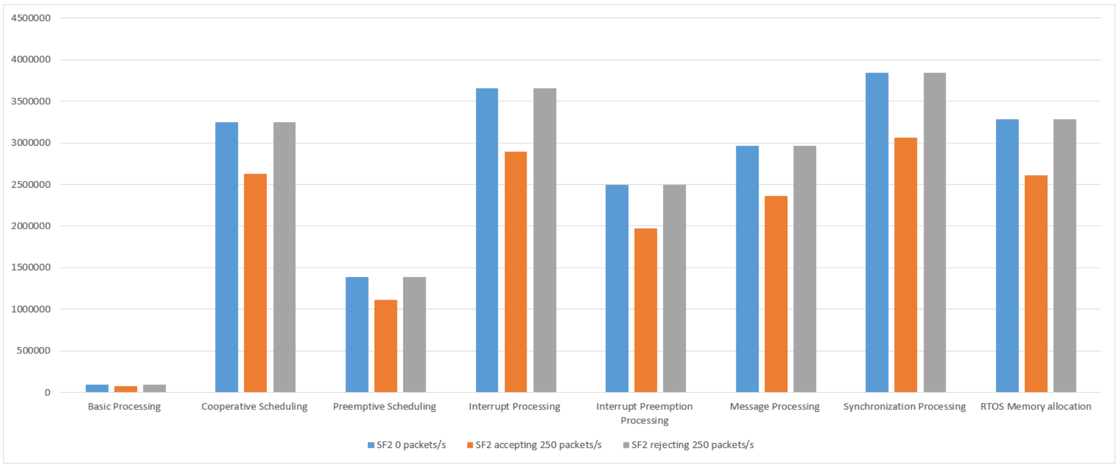

Figure 4.11: SmartFusion2 Thread-Metric results with XIoT accelerator and filter enabled.

to achieve a 19.56% increase of performance when the packets in the medium are not intended to the mote achieving the same performance results as when the medium is free.

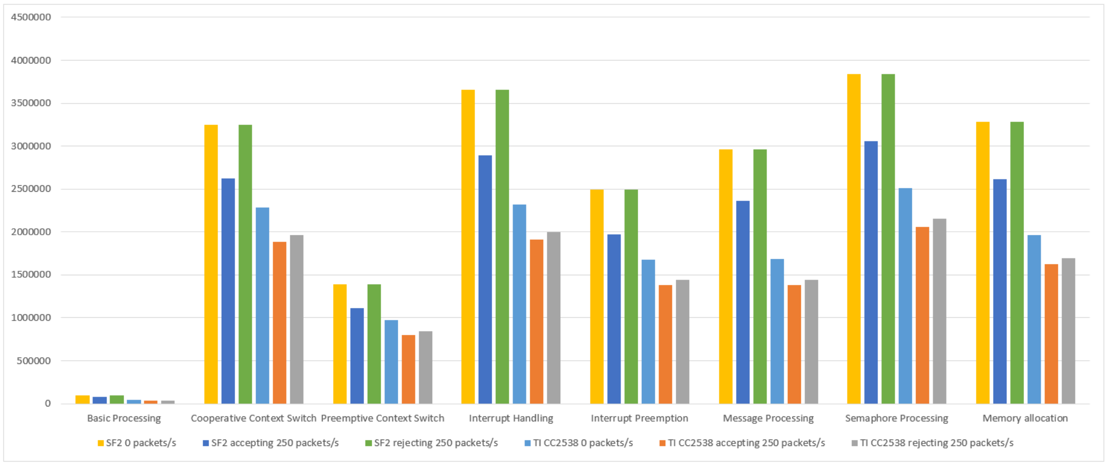

Figure 4.12: SmartFusion2 with XIoT and TI CC2538 Thread-Metric results.

Having now in consideration the COTS platform selected for this dissertation, the results presented in Figure 4.12 shows that the SmartFusion2 endowed with the XIoT and the filtering capabilities enabled achieved a 35,75% performance increase when accepting packets. Furthermore, is achieved the main goal of the XIoT accelerator, by enabling full OS availability when the packets present in the medium are not intended to be delivered to the mote under test, allowing to let the MCU in a low-power mode or performing other activities, while the

XIoT handles the reception and filtering of the packets being transmitted over the wireless network.

4.4.2 RCU Resources Utilization

The estimation of resources utilization was made with resource to the Libero pos-synthesis report, that specifies the resources needed to implement the logic developed in Verilog. As can be seen in Table 4.1 the XIoT is the most resource

Table 4.1: Synthesis results for the M2S090TS SoC.

Module	*4LUT*	*DFF*
`XIoT_0`	2551	3418
`CORE_SPI_0`	401	208
`Timer_0`	108	102
`RIOT_sb_0`	143	132
Total (out of 86184)	3204 (3.80%)	3860 (4.56%)

consuming hardware block requiring nearly 80% of the 4LUT and 89% of the DFF used. However, the implementation of this heterogeneous architecture just consumed 3.80% of the 4LUT and 4.56% of the DFF available in the FPGA fabric of the SmartFusion2 platform, leaving resources for the implementation of other soft-peripherals such as cryptographic accelerators, DPM modules or even to implement more filtering capabilities.

4.5 Conclusion

This Chapter described the combination of the XIoT and RIOT-OS, a hardware peripheral deployed in the RCU component of the CUTE mote that is intended to process and filter incoming network packets on low-end IoT devices, beginning with an introduction of such subject. Then, was exposed the XIoT that resulted from the upgrading of the hardware accelerator deployed in the CUTE mote architecture. Additionally, it was explained the developed API that was used to integrate the hardware accelerator in RIOT-OS. Finally, this Chapter presented the evaluation of the heterogeneous architecture with an RCU resources estimation and the OS evaluation using Thread-Metric Benchmark Suite. From the results obtained is possible to conclude that the usage of such hardware accelerators in low-end IoT devices allow managing more efficiently the network communication and provides greater benefit of the low-power modes through the avoiding of unnecessary calls to the MSS.

Chapter 5

Conclusion and Future Work

"I have no special talent. I am only passionately curious."
—Albert Einstein

The ubiquitous deployment of low-end IoT devices at the network edge brings new challenges in the development of connected devices up to the architectural level of the motes. Thus, this dissertation proposed the deployment of an OS with real time capabilities into a reconfigurable platform endowed with XIoT, an improved version of the previous accelerator supported by the CUTE mote.

This chapter presents the conclusions of this work in Section 5.1 and the future work in Section 5.2 finalizes this dissertation.

5.1 Conclusion

The widely and continuous deployment of embedded devices in the IoT network edge is making real the concept of ubiquitous computing presented by Mark Weiser. The devices used in the network edge are mainly the traditional low-end devices used in WSN, however, this new paradigm presents several challenges at the connectivity and interoperability levels. The connection of these devices to the Internet requires the incorporation of an IPv6 compliant network stack, however, the deployment of a such stack in these devices is not straightforward due to the complexity of the network stack and the scarce resources of the low-end devices. Thus, it raises the necessity of the implementation of a OS suitable for IoT, allowing to support a proper network stack in these devices and manage their inherent complexity.

Solutions like ASIC or COTS devices available in the market are not well suited for the IoT arena where a variety of scenarios impose different requirements and is not possible to develop a one-fit-all solution. However, the newest FPSoC solutions present the possibility to use well-established and supported hardware and software with the opportunity of integrating customized hardware blocks. These platforms provide the necessary flexibility to develop customized solutions for IoT and for this reason, was selected a FPSoC platform to develop this dissertation.

The acceleration on low-end devices is not new, as seen through this dissertation, however, the inclusion of an OS is not transversal to all the known solutions, and at the current state of the art, it was never presented a heterogeneous architecture with the inclusion of an RTOS. Furthermore, only CUTE mote presents a hardware soft-peripheral that permits is portability among platforms and applications, not being designed for a specific target. With the inclusion of network related accelerators, this architecture targets the low-end IoT devices improving their communication capabilities and power consumption.

In order to evaluate the behaviour of RIOT-OS in a COTS platform, it was deployed the OS in an already supported platform, the TI CC2538 with the usage of IAR Workbench. Furthermore, was adapted the Thread-Metrics Benchmark Suite to RIOT-OS and evaluated the OS performance in this platform. To perform this evaluation was created a network topology, with the platform under test running a UDP server application that receives all packets in the medium, this way was possible to establish a base line for further comparisons and understand how the OS availability is affected in the presence of an overloaded medium.

Then, it was used the Microsemi's SmartFusion2 to develop the heterogeneous architecture, composed by the MCU Arm Cortex-M3, the RCU present in the platform and attaching an 802.15.4 radio transceiver that is interfaced with the board through an SPI interface. The porting of RIOT-OS for this platform required the hardware configuration of the board and the development of a new hardware timer. On the software side, it was necessary to develop the peripheral layer for this board, including the development of a new GPIO, timer, CC2520 and SPI drivers. At this stage, the RCU only included the timer and the CoreSPI. Applying the same benchmark suite that was used to evaluate the COTS platform, it was verified a big performance improvement in the OS, mainly by the higher clock source provided by the SmartFusion2.

In the following, was presented the CUTE mote architecture and described the upgrade done over the network accelerator presented in such architecture, giving origin to the XIoT. The integration of XIoT in the heterogeneous architecture was also described and after that presented the API that allows the usage of the hardware peripheral by software. The integration of XIoT in RIOT-OS was also exposed in this dissertation, emphasizing, that the main changes to integrate the accelerator were only performed at the hardware dependent level in the CC2520 peripheral driver.

The evaluation procedure over the heterogeneous architecture was performed as previously using the same benchmark suite and the same network topology, and analysed the results from the SmartFusion2 endowed with the XIoT and the filtering capabilities enabled. This evaluation showed full OS availability when the packets present in the medium do not have as destination address the mote under test. Furthermore, the deployment of the heterogeneous architecture in the SmartFusion2 just required 3.80% of the 4LUT and 4.56% of the DFF resources available in the FPGA fabric.

With this dissertation, it was possible to conclude that the development of heterogeneous architectures with the inclusion of network related hardware accelerators contributes to an increase in performance availability of the OS in IoT low-end devices and allows full availability when the packets in the medium are not intended to be delivered to the mote. This way is possible to achieve longer periods of inactivity by the MCU contributing for a lower energy consumption by the main component of the architecture, or in performance demanding applications leave the MCU free to perform other activities while the XIoT handles the reception and filtering of network traffic. Furthermore, the resource utilization was in total below 5% of the available resources in the SmartFusion2, what allows to implement such capabilities in even constrained platforms, or just by adding a small device with FPGA fabric to legacy hardware. For last, the amount of resource used allows to implement such capabilities in silicon and provide in new radios these features while maintaining the low-end characteristics.

5.2 Future Work

Despite all the work developed during this dissertation some limitations still persist in the heterogeneous architecture. For this reason, and due to the proven potential of such architectures, further upgrades could be done and new functionalities integrated, such as:

- The integration of FIFOs in the XIoT to buffer filtered frames, allowing this way asynchronous communication between the OS and the hardware accelerator, enabling the store of multiple packets to be delivered to the MSS;
- Evaluate the usage of memories instead of DFF to accommodate the received packets, reducing this way the DFF usage that a big part is due to the accommodation the packets that the size can be up to 128 bits;
- The partitioning of the XIoT and the SPI soft-peripheral driver in two different hardware blocks increasing the agnostic implementation of the accelerator by extinguish the dependence on the communication interface used by the radio transceiver;
- Implementation of extra filtering capabilities such as IP, UDP ports, and application protocols. In the same line, implement in hardware the concept of black and/or white lists to allow in hardware reject the packets by its source address;
- Enable in the XIoT the modification by software of the filtering parameters and implement such functionalities in the XIoT API. At the moment of conclusion of this dissertation was only possible to enable and disable the filtering capabilities;
- Despite security being out of the scope of this dissertation, future work could include the usage of Arm's TrustZone technology providing a TEE for the mote, by protecting processor cores, peripherals and communication buses;
- Being the energy consumption out of scope of this dissertation, future work could include the incorporation of WuR in the heterogeneous architecture, allowing the implementation of sleep modes in the RCU;

- Study the application of this heterogeneous architecture in middle or high-end devices.

Bibliography

[1] K. Ashton, “June 22. That ’Internet of Things’ thing,” *RFID Journal*, 1999.

[2] K. Ashton *et al.*, “That ‘Internet of Things’ thing,” *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[3] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, “Internet of Things for Smart Cities,” *IEEE Internet of Things journal*, vol. 1, no. 1, pp. 22–32, 2014.

[4] D. Snoonian, “Smart buildings,” *IEEE spectrum*, vol. 40, no. 8, pp. 18–23, 2003.

[5] X. Krasniqi and E. Hajrizi, “Use of IoT technology to drive the automotive industry from connected to full autonomous vehicles,” *IFAC-PapersOnLine*, vol. 49, no. 29, pp. 269–274, 2016.

[6] N. Vijayakumar and R. Ramya, “The real time monitoring of water quality in IoT environment,” in *Innovations in Information, Embedded and Communication Systems (ICIIECS), 2015 International Conference on.* IEEE, 2015, pp. 1–5.

[7] L. D. Xu, “Information architecture for supply chain quality management,” *International Journal of Production Research*, vol. 49, no. 1, pp. 183–198, 2011.

[8] R. Badia-Melis, P. Mishra, and L. Ruiz-García, “Food traceability: New trends and recent advances. A review,” *Food Control*, vol. 57, pp. 393–401, 2015.

[9] J.-c. Zhao, J.-f. Zhang, Y. Feng, and J.-x. Guo, “The study and application of the IoT technology in agriculture,” in *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, vol. 2. IEEE, 2010, pp. 462–465.

[10] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt, “RIOT OS: Towards an OS for the Internet of Things,” in *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on.* IEEE, 2013, pp. 79–80.

[11] Moore, Gordon E, “Moore’s Law at 40,” *Understanding Moore’s law: four decades of innovation*, pp. 67–84, 2006.

[12] J. Koomey, S. Berard, M. Sanchez, and H. Wong, “Implications of historical trends in the electrical efficiency of computing,” *IEEE Annals of the History of Computing*, vol. 33, no. 3, pp. 46–54, 2011.

[13] J. J. Rodríguez-Andina, M. D. Valdes-Pena, and M. J. Moure, “Advanced features and industrial applications of FPGAs—A review,” *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 853–864, 2015.

[14] J. Bradley, J. Barbier, and D. Handler, “Embracing the Internet of Everything To Capture Your Share of $ 14 . 4 Trillion,” *Cisco Ibsg Group*, p. 2013, 2013. [Online]. Available: http://www.cisco.com/web/about/ac79/docs/innov/IoE_Economy.pdf

[15] Vestberg, Hans, “CEO to shareholders: 50 billion connections 2020,” [Online]. Available: https://www.ericsson.com/en/press-releases/2010/4/ceo-to-shareholders-50-billion-connections-2020, [Accessed on: Jul. 2, 2018].

[16] Cisco, “Cisco Visual Networking Index Predicts Near-Tripling of IP Traffic by 2020,” 2017. [Online]. Available: https://newsroom.cisco.com/press-release-content?type=press-release&articleId=1771211

[17] Ericsson, “IoT connections outlook,” [Online]. Available: https://www.ericsson.com/en/mobility-report/reports/november-2017/internet-of-things-outlook, [Accessed on: Jul. 2, 2018].

[18] Macaulay, James and Buckalew, Lauren, “INTERNET OF THINGS IN LOGISTICS,” [Online]. Available: http://www.dhl.com/en/about_us/logistics_insights/dhl_trend_research/internet_of_things.html, [Accessed on: Jul. 3, 2018].

[19] Intel Corporation, “A Guide to the Internet of Things Infographic,” [Online]. Available: https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html, [Accessed on: Jul. 2, 2018].

[20] IHS Markit, "IoT trend watch 2017," 2017. [Online]. Available: https://cdn.ihs.com/www/pdf/IoT-trend-watch-2017.pdf

[21] Newman, Daniel and Blanchard, Olivier, "2017 IOT BUSINESS," 2017. [Online]. Available: https://futurumresearch.com/product/iot-business-integration-index-2017/

[22] K. L. Lueth, "Iot market – forecasts at a glance," [Online]. Available: https://iot-analytics.com/iot-market-forecasts-overview, [Accessed on: Jul. 2, 2018].

[23] Sparks, Philip, "The route to a trillion devices The outlook for IoT investment to 2035," *ARM Whitepaper*, no. June, pp. 1–14, 2017.

[24] Knud Lasse Lueth, "The 10 most popular Internet of Things applications right now," [Online]. Available: https://iot-analytics.com/10-internet-of-things-applications, Accessed on: Jul. 4, 2018.

[25] M. D. V. Pena, J. J. Rodriguez-Andina, and M. Manic, "The internet of things: The role of reconfigurable platforms," *IEEE Industrial Electronics Magazine*, vol. 11, no. 3, pp. 6–19, 2017.

[26] L. Da Xu, W. He, and S. Li, "Internet of things in industries: A survey," *IEEE Transactions on industrial informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.

[27] P. Fremantle, "A reference architecture for the internet of things," *WSO2 White paper*, 2014.

[28] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: the internet of things architecture, possible applications and key challenges," in *Frontiers of Information Technology (FIT), 2012 10th International Conference on.* IEEE, 2012, pp. 257–260.

[29] Micrium, "How to Think about the Internet of Things (IoT)," [Online]. Available: https://https://micrium.com/rtos/, Accessed on: Jul. 8, 2018.

[30] A. Jain, B. Sharma, and P. Gupta, "Internet of things: Architecture, security goals, and challenges-a survey," *International Journal of Innovative Research in Science and Engineering*, no. 2, 2016.

[31] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares, "IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices," *IEEE Internet Computing*, vol. 21, no. 1, pp. 40–47, 2017.

[32] S. Pinto, D. Oliveira, J. Pereira, J. Cabral, and A. Tavares, "FreeTEE: When real-time and security meet," in *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on.* IEEE, 2015, pp. 1–4.

[33] D. Singh, G. Tripathi, and A. J. Jara, "A survey of Internet-of-Things: Future vision, architecture, challenges and services," in *Internet of things (WF-IoT), 2014 IEEE world forum on.* IEEE, 2014, pp. 287–292.

[34] Olsson, Jonas, "6LoWPAN demystified," *Texas Instruments*, p. 13, 2014.

[35] Gungor, Vehbi C and Hancke, Gerhard P, "Industrial wireless sensor networks: Challenges, design principles, and technical approaches," *IEEE Transactions on industrial electronics*, vol. 56, no. 10, pp. 4258–4265, 2009.

[36] T. M. R. Gomes, "A sensor node soc architecture for extremely autonomous wireless sensor networks," 2017.

[37] K. Pantelaki, S. Panagiotakis, and A. Vlissidis, "Survey of the IEEE 802.15. 4 Standard's developments for wireless sensor networking," *Am. J. Mobile Syst. Appl. Serv*, vol. 2, no. 1, pp. 13–31, 2016.

[38] Z. Shelby, "Neighbor discovery optimization for low power and lossy networks (6lowpan)," *draft-ietf-6lowpan-nd-18*, 2011.

[39] P. Thubert and J. W. Hui, "Compression format for IPv6 datagrams over IEEE 802.15. 4-based networks," 2011.

[40] H. Petersen, M. Lenders, M. Wählisch, O. Hahm, and E. Baccelli, "Old Wine in New Skins?: Revisiting the Software Architecture for IP Network Stacks on Constrained IoT Devices," in *Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems.* ACM, 2015, pp. 31–35.

[41] M. Bauer, M. Boussard, M. Bui, F. Carrez, C. Jardak, J. De Loof, C. Magerkurth, S. Meissner, A. Nettsträter, A. Olivereau *et al.*, "Deliverable D1. 5—Final architectural reference model for the IoT v3. 0," 2013.

[42] A. Bassi, M. Bauer, M. Fiedler, T. Kramp, R. Van Kranenburg, S. Lange, and S. Meissner, *Enabling things to talk.* Springer, 2016.

[43] T. Winter, "RPL: IPv6 routing protocol for low-power and lossy networks," 2012.

[44] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: a survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, 2016.

[45] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks," Tech. Rep., 2014.

[46] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, "Standardized Protocol Stack for the Internet of (important) Things," *IEEE communications surveys & tutorials*, vol. 15, no. 3, pp. 1389–1406, 2013.

[47] J.-H. Hoepman and B. Jacobs, "Increased security through open source," *Communications of the ACM*, vol. 50, no. 1, pp. 79–83, 2007.

[48] J. Eriksson, A. Dunkels, N. Finne, F. Osterlind, and T. Voigt, "Mspsim–an extensible simulator for msp430-equipped sensor boards," in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, vol. 118, 2007.

[49] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[50] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on.* IEEE, 2004, pp. 455–462.

[51] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, *TinyOS: An Operating System for Sensor Networks*, 01 2005, vol. 00.

[52] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, 2018.

[53] FreeRTOS, "The FreeRTOS Kernel ," [Online]. Available: https://freertos.org/, Accessed on: Aug. 3, 2018.

[54] "Amazon FreeRTOS," [Accessed: 22-Aug-2018]. [Online]. Available: https://aws.amazon.com/pt/freertos/

[55] "Mbed OS," [Accessed: 22-Aug-2018]. [Online]. Available: https://www.mbed.com/en/platform/mbed-os/

[56] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 563–579, 2005.

[57] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-rk: an energy-aware resource-centric rtos for sensor networks," in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International.* IEEE, 2005, pp. 10–pp.

[58] Ethernut Project, "Nut/OS Software Manual," [Online]. Available: http://ethernut.de/pdf/enswm28e.pdf, Accessed on: Aug. 3, 2018.

[59] G. Strazdins, A. Elsts, and L. Selavo, "MansOS: easy to use, portable and resource efficient operating system for networked embedded devices," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems.* ACM, 2010, pp. 427–428.

[60] Micrium, "μC/OS RTOS & Stacks," [Online]. Available: https://micrium.com/iot/devices, Accessed on: Jul. 3, 2018.

[61] μClinux, "μClinux," [Online]. Available: http://uclinux.org/, Accessed on: Jul. 3, 2018.

[62] P. Gaur and M. P. Tahiliani, "Operating Systems for IoT Devices: A Critical Survey," in *Region 10 Symposium (TENSYMP), 2015 IEEE.* IEEE, 2015, pp. 33–36.

[63] M. O. Farooq and T. Kunz, "Operating systems for wireless sensor networks: A survey," *Sensors*, vol. 11, no. 6, pp. 5900–5930, 2011.

[64] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems.* Acm, 2006, pp. 29–42.

[65] A. Dunkels, "Full TCP/IP for 8-bit architectures," in *Proceedings of the 1st international conference on Mobile systems, applications and services.* ACM, 2003, pp. 85–98.

[66] Dunkels, Adam, "Rime-a lightweight layered communication stack for sensor networks," in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Delft, The Netherlands*, 2007.

[67] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with cooja," in *Local computer networks, proceedings 2006 31st IEEE conference on.* IEEE, 2006, pp. 641–648.

[68] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-Elefteri, P. Levis, A. Terzis, and R. Govindan, "TOSThreads: thread-safe and non-invasive preemption in TinyOS." in *SenSys*, vol. 9, 2009, pp. 127–140.

[69] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, "Efficient memory safety for TinyOS," in *Proceedings of the 5th international conference on Embedded networked sensor systems.* ACM, 2007, pp. 205–218.

[70] TinyOS, "BLIP Tutorial," [Online]. Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/BLIP_Tutorial, Accessed on: Marc. 30, 2018.

[71] S. Dawson-Haggerty, A. Tavakoli, and D. Culler, "Hydro: A hybrid routing protocol for low-power and lossy networks," in *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on.* IEEE, 2010, pp. 268–273.

[72] Z. Wang, W. Li, and H. Dong, "Review on open source operating systems for internet of things," in *Journal of Physics: Conference Series*, vol. 887, no. 1. IOP Publishing, 2017, p. 012044.

[73] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Transactions on database systems (TODS)*, vol. 30, no. 1, pp. 122–173, 2005.

[74] C. Karlof, N. Sastry, and D. Wagner, "TinySec: a link layer security architecture for wireless sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems.* ACM, 2004, pp. 162–175.

[75] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *Proceedings of the 1st international conference on Embedded networked sensor systems.* ACM, 2003, pp. 126–137.

[76] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *Acm Sigplan Notices*, vol. 49, no. 4, pp. 41–51, 2014.

[77] H. Will, K. Schleiser, and J. Schiller, "A real-time kernel for wireless sensor networks employed in rescue scenarios," in *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on.* IEEE, 2009, pp. 834–841.

[78] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündoğan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things," *arXiv preprint arXiv:1801.02833*, 2018.

[79] "Berkeley's OpenWSN Project," [Accessed: 29-Marc-2018]. [Online]. Available: https://team.inria.fr/eva/berkeleys-openwsn

[80] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele *et al.*, "FIT IoT-LAB: A large scale open experimental IoT testbed," in *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on.* IEEE, 2015, pp. 459–464.

[81] M. Güneş, F. Juraschek, B. Blywis, Q. Mushtaq, and J. Schiller, "A testbed for next generation wireless network research," *PIK-Praxis der Informationsverarbeitung und Kommunikation*, vol. 32, no. 4, pp. 208–212, 2009.

[82] Stratistics, "Field Programmable Gate Array (FPGA) - Global Market Outlook (2017-2026)," [Online]. Available: http://www.strategymrc.com/report/field-programmable-gate-array-fpga-market, Accessed on: May 23, 2018.

[83] Microsemi Corporation, "PolarFire FPGA Family," [Online]. Available: https://www.microsemi.com/product-directory/fpgas/3854-polarfire-fpgas, Accessed on: May 23, 2018.

[84] Microsemi Corporation, "IGLOO2 FPGA Family," [Online]. Available: https://www.microsemi.com/product-directory/fpgas/1688-igloo2, Accessed on: May 23, 2018.

[85] Lattice Semiconductor, “LatticeXP2,” [Online]. Available: http://www.latticesemi.com/Products/FPGAandCPLD/LatticeXP2.aspx, Accessed on: May 23, 2018.

[86] Xilinx (Dec. 6, 2017), “Zynq-7000 All Programmable SoC (v1.12.1),” [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, Accessed on: Dec. 10, 2017.

[87] Intel Corporation, “Cyclone V SoCs,” [Online]. Available: https://www.altera.com/products/soc/portfolio/cyclone-v-soc/overview.html, Accessed on: May 23, 2018.

[88] Microsemi Corporation, “SmartFusion2,” [Online]. Available: https://www.microsemi.com/product-directory/soc-fpgas/1692-smartfusion2, Accessed on: May 24, 2018.

[89] M. Corporation, “SmartFusion2 Security Evaluation Kit,” [Online]. Available: https://www.microsemi.com/existing-parts/parts/143988, Accessed on: May 24, 2018.

[90] W. Lamie and J. Carbone, “Measure your RTOS’s real-time performance,” *Embedded Systems Design*, vol. 20, no. 5, p. 44, 2007.

[91] ANALOG DEVICES, “EVAL-WSN,” [Online]. Available: http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/eval-wsn.html, Accessed on: Jul. 18, 2018.

[92] Micro:bit Educational Foundation, “micro:bit,” [Online]. Available: http://microbit.org, Accessed on: Jul. 18, 2018.

[93] SECO SRL, “UDOO,” [Online]. Available: https://udoo.org, Accessed on: Jul. 18, 2018.

[94] OpenMote Technologies, “OpenMote,” [Online]. Available: http://openmote.com, Accessed on: Jul. 18, 2018.

[95] Advantic Systems, “Tmote Sky,” [Online]. Available: https://telosbsensors.wordpress.com, Accessed on: Jul. 18, 2018.

[96] Eistec, “Mulle,” [Online]. Available: http://eistec.se/mulle, Accessed on: Jul. 18, 2018.

[97] Arago Systems, “WiSMote,” [Online]. Available: http://aragosystems.com/produits/wisnet/wismote, Accessed on: Jul. 18, 2018.

[98] Zolertia, “RE-MOTE,” [Online]. Available: http://zolertia.io/product/re-mote, Accessed on: Jul. 18, 2018.

[99] M. Shibata, Y. Ohtsuka, M. Takahashi, and K. Okamoto, “Advanced FPGA technology trend based on patent analysis with link mining,” in *Electronics Packaging and iMAPS All Asia Conference (ICEP-IAAC), 2018 International Conference on.* IEEE, 2018, pp. 147–151.

[100] A. P. Johnson, R. S. Chakraborty, and D. Mukhopadhyay, “A PUF-enabled secure architecture for FPGA-based IoT applications,” *IEEE Transactions on Multi-Scale Computing Systems*, vol. 1, no. 2, pp. 110–122, 2015.

[101] C. Marchand, L. Bossuet, U. Mureddu, N. Bochard, A. Cherkaoui, and V. Fischer, “Implementation and characterization of a physical unclonable function for IoT: a case study with the TERO-PUF,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 97–109, 2018.

[102] A. K. Nain, J. Bandaru, M. A. Zubair, and R. Pachamuthu, “A secure phase-encrypted IEEE 802.15. 4 transceiver design,” *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1421–1427, 2017.

[103] A. Boutros, S. Hesham, B. Georgey, and M. A. A. El Ghany, “Hardware acceleration of novel chaos-based image encryption for IoT applications,” in *Microelectronics (ICM), 2017 29th International Conference on.* IEEE, 2017, pp. 1–4.

[104] M. B. Chellam and R. Natarajan, “AES Hardware Accelerator on FPGA with Improved Throughput and Resource Efficiency,” *Arabian Journal for Science and Engineering*, pp. 1–18, 2017.

[105] M. Rao, T. Newe, I. Grout, and A. Mathur, “An FPGA-based reconfigurable IPSec AH core with efficient implementation of SHA-3 for high speed IoT applications,” *Security and Communication Networks*, vol. 9, no. 16, pp. 3282–3295, 2016.

[106] Y. Wang, Y. Shi, C. Wang, and Y. Ha, “FPGA-based SHA-3 acceleration on a 32-bit processor via instruction set extension,” in *Electron Devices*

and Solid-State Circuits (EDSSC), 2015 IEEE International Conference on. IEEE, 2015, pp. 305–308.

[107] M.-j. Sung and K.-W. Shin, “An Efficient Hardware Implementation of Lightweight Block Cipher LEA-128/192/256 for IoT Security Applications,” *Journal of the Korea Institute of Information and Communication Engineering*, vol. 19, no. 7, pp. 1608–1616, 2015.

[108] C. P. Antonopoulos and N. S. Voros, “A data compression hardware accelerator enabling long-term biosignal monitoring based on ultra-low power IoT platforms,” *Electronics*, vol. 6, no. 3, p. 54, 2017.

[109] A. De La Piedra, A. Braeken, and A. Touhafi, “Sensor systems based on FPGAs and their applications: a survey,” *Sensors*, vol. 12, no. 9, pp. 12 235–12 264, 2012.

[110] A. Engel and A. Koch, “Heterogeneous wireless sensor nodes that target the internet of things,” *IEEE Micro*, vol. 36, no. 6, pp. 8–15, 2016.

[111] O. Berder and O. Sentieys, “Powwow: Power optimized hardware/software framework for wireless motes,” in *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on.* VDE, 2010, pp. 1–5.

[112] V. Rosello, J. Portilla, and T. Riesgo, “Ultra low power FPGA-based architecture for wake-up radio in wireless sensor networks,” in *IECON 2011-37th Annual Conference on IEEE Industrial Electronics Society.* IEEE, 2011, pp. 3826–3831.

[113] T. Gomes, F. Salgado, A. Tavares, and J. Cabral, “CUTE Mote, A Customizable and Trustable End-Device for the Internet of Things,” *IEEE Sensors Journal*, vol. 17, no. 20, pp. 6816–6824, 2017.

[114] A. Engel and A. Koch, “Hardware-accelerated data compression in low-power wireless sensor networks,” in *International Symposium on Applied Reconfigurable Computing.* Springer, 2014, pp. 167–178.

[115] A. Engel, A. Koch, and T. Siebel, “A heterogeneous system architecture for low-power wireless sensor nodes in compute-intensive distributed applications,” in *Local Computer Networks Conference Workshops (LCN Workshops), 2015 IEEE 40th.* IEEE, 2015, pp. 636–644.

[116] J. Valverde, V. Rosello, G. Mujica, J. Portilla, A. Uriarte, and T. Riesgo, "Wireless sensor network for environmental monitoring: application in a coffee factory," *International Journal of Distributed Sensor Networks*, vol. 8, no. 1, p. 638067, 2011.

[117] J. Valverde, A. Otero, M. Lopez, J. Portilla, E. De La Torre, and T. Riesgo, "Using SRAM based FPGAs for power-aware high performance wireless sensor networks," *Sensors*, vol. 12, no. 3, pp. 2667–2692, 2012.

[118] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," in *Annual International Cryptology Conference*. Springer, 2017, pp. 570–596.

[119] T. Gomes, S. Pinto, A. Tavares, and J. Cabral, "Towards an FPGA-based edge device for the Internet of Things," in *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*. IEEE, 2015, pp. 1–4.

[120] T. Gomes, F. Salgado, S. Pinto, J. Cabral, and A. Tavares, "Towards an FPGA-based network layer filter for the Internet of Things edge devices," in *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE, 2016, pp. 1–4.

[121] Gomes, T and Salgado, F and Pinto, S and Cabral, J and Tavares, A, "A 6LoWPAN Accelerator for Internet of Things Endpoint Devices," *IEEE Internet of Things Journal*, 2017.

[122] T. Gomes, S. Pinto, F. Salgado, A. Tavares, and J. Cabral, "Building IEEE 802.15. 4 accelerators for heterogeneous wireless sensor nodes," *IEEE Sensors Letters*, vol. 1, no. 1, pp. 1–4, 2017.

[123] Gutierrez, Jose A and Naeve, Marco and Callaway, Ed and Bourgeois, Monique and Mitter, Vinay and Heile, Bob, "IEEE 802.15. 4: a developing standard for low-power low-cost wireless personal area networks," *IEEE network*, vol. 15, no. 5, pp. 12–19, 2001.

[124] A. Rupani, D. Pandey, and G. Sujediya, "Review and Study of FPGA Implementation of Internet of Things," *IJSTE-International Journal of Science Technology & Engineering*, vol. 3, no. 2.

[125] L. A. Vera-Salas, S. V. Moreno-Tapia, R. A. Osornio-Rios *et al.*, "Reconfigurable node processing unit for a low-power wireless sensor network,"

in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on.* IEEE, 2010, pp. 173–178.

[126] B. Stelte, “Toward development of high secure sensor network nodes using an FPGA-based architecture,” in *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference.* ACM, 2010, pp. 539–543.

[127] T. Nylanden, J. Boutellier, K. Nikunen, J. Hannuksela, and O. Silvén, “Reconfigurable miniature sensor nodes for condition monitoring,” in *Embedded Computer Systems (SAMOS), 2012 International Conference on.* IEEE, 2012, pp. 113–119.

[128] RIOT, “RIOT in a nutshell,” [Online]. Available: https://riot-os.org/api/index.html, Accessed on: Jul. 23, 2018.