Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science

Integrating Long Range Technology into the Contiki Operating System Framework

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

Jelle Aerts

Promotors:	Prof. Dr. Ir. Kris Steenhaut
	Prof. Dr. Ir. Jacques Tiberghien
Advisors:	Ir. Steffen Thielemans
	Ir. Maite Bezunartea

VRIJE UNIVERSITEIT BRUSSEL · SCIENTIA VINCERE TENEBRAS ·

June 2016

Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Vakgroep Computerwetenschappen

Integrating Long Range Technology into the Contiki Operating System Framework

Proefschrift ingediend met het oog op het behalen van de graad van
Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

Jelle Aerts

Promotors:	Prof. Dr. Ir. Kris Steenhaut
	Prof. Dr. Ir. Jacques Tiberghien
Begeleiders:	Ir. Steffen Thielemans
	Ir. Maite Bezunartea

Juni 2016

Abstract

The Internet of Things in which everyday objects are provided with a microcontroller and a radio transceiver is developing rapidly at the moment. Quite recently, some new wireless technologies like LoRa and SigFox became available and offer energy efficient communication over long distances. Currently, they are mainly employed to build infrastructures with a classic star topology. However, this Master's Thesis took LoRa completely out of its traditional context in order to take the first steps towards a long-distance multi-hop mesh network. As such, the popular operating system for hardware platforms in wireless sensor networks Contiki was ported to the LoRaMote demo platform. The experiences that were gained during this process were bundled in order to provide the online Contiki community for the first time with a detailed and complete porting guide. Finally, different LoRa configurations and their influence on the wireless range between two LoRaMotes were tested.

Het Internet of Things waarin dagelijkse voorwerpen worden voorzien van een elektronische chip en radiozender, is momenteel nog volop in ontwikkeling. Recentelijk kwamen er enkele nieuwe draadloze technologieën zoals LoRa en SigFox op de markt die energiezuinige communicatie over lange afstanden bieden. Op het ogenblik worden deze hoofdzakelijk aangewend om infrastructuren met een klassieke stertopologie te bouwen. In deze masterthesis werd LoRa echter volledig uit de traditionele context gehaald om zo de eerste stappen richting een lange afstands-, multi-hop, mesh netwerk te zetten. Zo werd Contiki, een populair besturingssysteem voor hardware platformen in draadloze sensornetwerken, naar het LoRaMote demoplatform overgezet. De ervaringen die hieruit volgden, werden gebundeld om zo voor de eerste keer een gedetailleerde en volledige "porting" handleiding te verschaffen aan de online Contiki gemeenschap. Tot slot werden verschillende LoRa configuraties en hun effect op het draadloos bereik tussen twee LoRaMotes getest.

Table of Contents

List of Figures

List of Acronyms

6LoWPAN IPv6 over Low power Wireless Personal Area Networks

AES Advanced Encryption Standard

API Application Programming Interface

ARM Acorn RISC Machine

CCA Clear Channel Assessments

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CSMA Carrier Sense Multiple Access

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance

CSS Chirp Spread Spectrum

DFU Device Firmware Upgrade

EEPROM Electrically Erasable Programmable Read-Only Memory

ELF Executable and Linking Format

ETSI European Telecommunications Standards Institute

FEC forward error correction

FIFO First In, First Out

GCC GNU Compiler Collection

GPIO General-Purpose Input/Output

GPS Global Positioning System

GUI graphical user interface

I/O Input/Output

IDE Integrated Development Environments

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IoT Internet of Things

IP Internet Protocol

IPv6 Internet Protocol version 6

ISM Industrial, Scientific and Medical

JTAG Joint Test Action Group

LCD Liquid-Crystal Display

LED Light Emitting Diode

LLC Logical Link Control

LoRaWAN Long Range Wide Area Network

LPM low power mode

LPP Low Power Probing

LPWAN Low-Power Wide-Area Network

LQI Link Quality Indicator

M2M Machine-to-Machine

MAC Media Access Control

MIC message integrity code

MTU Maximum Transmission Unit

OS operating system

OSI Open Systems Interconnection

RAM Random-Access Memory

RDC Radio Duty Cycling

RFC Request For Comments

RPL IPv6 Routing Protocol for Low-Power and Lossy Networks

SAR Specific Absorption Rate

SF spreading factor

SLIP	Serial Line Internet Protocol
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
uIP	micro IP
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
WLAN	Wireless Local Area Network
WSN	Wireless Sensor Networks

1 | Introduction

The SmartNet research group which is part of the ETRO-IRIS department at the Vrije Universiteit Brussel investigates issues related to Wireless Sensor and Actuator Networks. They propose improvements to certain protocols used in such networks and deploy them to explore new applications. ETRO for example is currently developing a feature extraction system solely based on raw camera images obtained from Wireless Sensor Networks (WSN). The high frequency (2.4GHz) and the link budget typical for the commonly used radios in these networks however exclude the connected devices from applications requiring radio waves to cross thick stone walls.

Quite recently, new transceiver technologies which use other license-free bands have emerged. They enable power efficient communication over very long distances and also allow communicating through several thick walls. Examples of such Low-Power Wide-Area Network (LPWAN) technologies are LoRa, SigFox and Weightless. Typically, such novel LPWANs are based upon single-hop star networks built around base stations, similar to the cellular telephone networks. However, these transceivers are potentially very useful to construct more generic Internet of Things (IoT) networks incorporating multi-hop bidirectional communication enabling sensing and actuation. This is the very reason why ETRO is also developing a multi-hop protocol for the LoRa technology at the moment.

Although these new technologies are promising, they are not mature, not standardized and often distributed with a proprietary software framework. This results in incompatibility with existing applications. Contiki on the other hand is a well-known open-source operating system (OS) for severely resource constrained systems and has been around for quite a while now. Integrating those new technologies in the Contiki OS would provide full integration of the devices in the Internet of Things. Practically, this can lead to applications where only a few devices are needed to cover a great surface, or where long range technologies can pass small messages like status updates or broadcasts between several short-range Wireless Sensor Networks or to a distant base station. Zolertia, the manufacturer of the popular WSN device Z1 already adopted the SigFox technology in its new dual-stack platform RE-Mote. A similar project to integrate LoRa technology into Wireless Sensor Networks is however non-existent.

This Master's Thesis consists of taking the first step in this LoRa integration process.

By porting the Contiki operating system to a LoRa platform, it will be possible to empower some advantages of this new technology. Moreover, it will also provide the research team with a full-featured mature operating system for these devices that can be used to perform tests with their experimental LoRa multi-hop protocol. The ETRO department is porting Contiki to a Texas Instruments platform too which will be used by the smart lighting project EDISON and some power-line applications. Detailed guidelines on how to perform such port are however lacking. So, another purpose is to gather all information and document the experiences that will be acquired during this project. These findings then can be used to document a general approach on porting Contiki to new hardware platforms.

In the next chapter a profound background on Wireless Sensor Networks, the Contiki operating system and the LoRa technology will be presented. Chapter 3 then continues on how this project was approached, which conceptual challenges were taken into account and which problems had to be tackled. The general guidelines on porting Contiki that were created based on the gained experience are presented in the 4th chapter. The results that were achieved are discussed in Chapter 5. In the last chapter an overview of the whole project is given, conclusions are drawn and some ideas for future work are suggested. The appendices contain the source code of all different components that are needed to successfully port Contiki to the LoRaMote demo platform.

2 | Background

The Internet has been around for quite some time now, but ever since the rise of wireless network technologies the desire of connectivity has been fueled remarkably. The development of Wireless Local Area Network (WLAN) standards and mobile telephony services has greatly increased the number of interconnected devices. Up till now, this amount of connected devices was more or less restricted to the number of humans. With the current course of events and upcoming breakthrough of the Internet of Things (IoT) in which all kinds of devices will be connected to the Internet, this virtual restriction will no longer hold [1]. The number of connected devices will grow significantly in the near future [2] and the latest version of the Internet Protocol (IPv6) will help connecting them all.

Lots of modern applications rely on data collection to offer new services or service improvements. Industrial control and monitoring, home automation and consumer electronics, environmental and health monitoring, security and military sensing, asset tracking and supply chain management or intelligent agriculture [3, 4] are just some examples of such applications. Collecting information and creating statistics has been common practice for ages, but there were always some restricting issues present. Either people or wired hardware sensors had to perform the measurements. Both have huge disadvantages. Manpower is costly and is not available all the time whereas wired sensors require lots of cable equipment and are not mobile. Then there were also limitations on the employability in certain locations or environments. Some areas can be difficult to access and/or stay at to perform measurements. Whenever the data was collected, it also had to be structured and processed by hand or with some aid of computers.

2.1 Wireless sensor networks

Next to the previously mentioned mainstream wireless technologies, other wireless solutions have been researched and developed as well. Wireless Sensor Networks are self-configuring networks of small sensor nodes scattered in a certain area to monitor environmental conditions [5]. They have been developed especially for data gathering purposes since they have almost no environmental limitations and are relatively inexpensive [4, page 2]. Such inexpensive and easy employable wireless networks will be one of the cornerstones of the Internet of Things and will become omnipresent. As a result, huge amounts of data will be generated. Even more than classical data processing approaches can handle. Therefore the Big Data technology will be an indispensable tool to deduct, visualize and generate useful information from the enormous amount of available data [6].

With the emerging of all these new technologies, the interest in and the market of the Internet of Things is increasing. Quite recently, Belgian telecommunication company Proximus and a subsidiary of electric utility company Engie started to deploy experimental IoT-networks [7, 8]. Regular workshops to keep everyone up to date on the latest wireless technology developments are organized by and for the "wireless community" which consists of experts from the wireless communication industry and involved academia. During their 21st workshop, lots of tips and tricks were given on how to develop sustainable IoT-applications and several demo implementations were showcased. The example applications included solutions for pest control, city environment monitoring and tracking of material, people and even animals. One demo displayed a modular microcontroller with several plug-and-play peripherals [9]. Another interesting concept was presented: the City of Things. It is an open Internet of Things playground, currently in deployment in Antwerp, Belgium. Several IoT-technologies are installed throughout the city, which act as a testbed for performing network and data experiments [10].

Although all these developments may be a great outlook, the current state of affairs does not allow bulk deployments yet. This is because hardware nodes (or motes) in wireless sensor networks have to operate under restricted conditions [3]. These restrictions are a direct result from the nature of sensor node applications. They have to be cheap and only capable of processing relatively easy computations. Therefore usual specifications of motes are a low-end microcontroller, a low-power radio transceiver, some sensors and a battery pack [11, page 457]. The prolonged lifetime of WSN applications leads to strict energy constraints. When a network is required to operate several months or years, there is a huge need for decent energy consumption management. In order to extend the lifespan of the network, nodes will operate in a low-power or sleep mode most of the time. Besides operating in low-power mode, it is also possible to harvest energy from the environment in some cases. To ensure low power consumption and minimal service disruption, special algorithms are needed to put the radio transceivers into sleep mode. Such algorithms are known as duty cycle schemes [12]. Another required characteristic of a WSN network is resilience. For various reasons, a sensor node can fail. Its battery can be depleted, an electronic problem can unfold or it may be even a software bug. Whatever the reason for the failure, it is essential that the network is able to successfully continue relaying data in a timely manner [13]. A last important criterion that has to be taken into account is the impossibility of software upgrades. Since most networks are not designed to transmit huge amounts of traffic and the nodes have strict energy constraints, it is unfeasible to distribute upgrades once the device is deployed. Therefore its software should contain everything the node will ever need [9]. The challenges that result from these restrictions have to be tackled before the Internet of Things will bring advanced applications on a large scale [14].

2.1.1 WSN communication standards

To maintain network connectivity in an environment with the presented challenges, a different communication approach was needed. Several wireless communication standards were developed to address the special requirements.

In 2003 the Institute of Electrical and Electronics Engineers (IEEE) created the IEEE 802.15.4 communication standard. It defines the physical and Media Access Control (MAC) layer in the OSI network model and has been designed specifically for wireless low-data-rate devices with limited energy consumption requirements. On the physical layer, it is responsible for tasks like activation of the radio transceiver, calculating the Link Quality Indicator (LQI), performing Clear Channel Assessments (CCA) and selecting the channel frequency. The MAC layer is responsible for generating and synchronizing network beacons, supporting device security, providing a reliable link between two peer entities and employing the CSMA/CA mechanism [15]. CSMA/CA stands for Carrier Sense Multiple Access with Collision Avoidance and regulates access to the shared medium for multiple devices. Using this method, devices 'listen' to the communication medium before transmitting. If the channel is clear, a device can decide to start the transmission. If another device also started sending, the transmission will be delayed for a random period of time.

Although IEEE 802.15.4 may be a great development, it also has a downside. Natively, it does not support IP packets. Since the Internet Protocol has become the de facto standard for computer networks it is difficult to communicate with external services for devices that use the IEEE 802.15.4 standard. To solve this problem, the Internet Engineering Task Force (IETF) has created the 6LoWPAN protocol which is an acronym for IPv6 over Low power Wireless Personal Area Networks. The main conflict between the two standards consists in the difference in packet size. The IEEE 802.15.4 standard has a maximum frame size of 127 bytes whereas the IPv6 Maximum Transmission Unit (MTU) has to be at least 1280 bytes. Therefore IPv6 packets of 128 bytes or more will simply not fit within an IEEE 802.15.4 frame. At this point 6LoWPAN will operate as an adaptation layer between the Network and Data Link Layer of the OSI network model by fragmenting IP packets upon transmission and reassembling IEEE 802.15.4 frames upon receival. [16]

Several other WSN communication technologies have been developed, building on the IEEE 802.15.4 standard. Some examples are ZigBee, WirelessHART, OCARI, MiWi, ISA100.11a and IEC 62601 (WIA-PA) [17, page 34-2]. These standards will not be discussed here since they are out of the scope of this project.

2.1.2 WSN operating systems

Next to communication standards, several operating systems were developed as well. These are dedicated to operating in low-power circumstances with constrained hardware and processing resources [18]. In this section, three widely used WSN focused operating systems will shortly be introduced. The Contiki operating system will be discussed in the next section.

TinyOS is a lightweight open-source operating system and is written in nesC, a dialect of the C programming language. It is designed to run embedded systems with very low memory requirements and is capable of running with just 400 bytes of memory [19]. This is the result of the fact that no kernel is present, so direct hardware manipulation is required [20]. Two types of processes are distinguished, time-critical events and non-time-critical tasks. Tasks are scheduled by means of adding them to a FIFO queue. When a task is started, it will run until it is completed. In the meantime no other task can be processed. Events on the other hand, are able to stop a running task and get executed. These events are not scheduled but implemented as interrupts. TinyOS provides three multi-hop protocols: dissemination, DIP (a newer dissemination protocol) [21] and TYMO (an adapted DYMO protocol) and also supports 6LoWPAN. On the MAC layer it has support for a TDMA protocol, B-MAC [22], an adaption of Z-MAC [23] and IEEE 802.15.4. [24, page 5908].

MANTIS, the MultimodAl system for NeTworks of In-situ wireless Sensors is another lightweight open-source operating system for embedded systems with multi-threading capabilities. It is written in plain C and runs with only 500 bytes of memory [20]. The process scheduler is based upon the UNIX-style schedulers and employs a priority-based round-robin algorithm [25]. Each process is assigned a priority class and will share runtime on the processor with the processes of the same class once the scheduler starts executing the processes in its priority class. The network stack is implemented in a somewhat special way in MANTIS. The physical and MAC layer are managed by the system, while the other layers are run in user space. This provides more flexibility to user applications since they can be switched with other protocols, but also results into performance loss since the radio driver cannot be accessed directly but only through an interface which results in computational overhead [24, page 5916]. The COMM component which manages the radio driver and MAC layer implements a protocol which resembles the S-MAC protocol [26] and zero-copy sockets [25].

Nano-RK is an open-source real-time operating system written in C. Once loaded it uses 2KB of memory. Its scheduler uses a preemptive priority-based algorithm. By executing at each given time the process with the highest priority, support for real-time applications is empowered [27]. The network stack consists of a TDMA-based protocol and zero-copy buffering [28, 24].

2.2 Contiki

Contiki is a lightweight, open-source operating system designed to run on resource constrained, wireless networked motes [29]. It is developed by Adam Dunkels in 2002 after he successfully created a slimmed down implementation of the TCP/IP stack [30]. Contiki is written in plain C, runs on average with a mere 2KB of memory

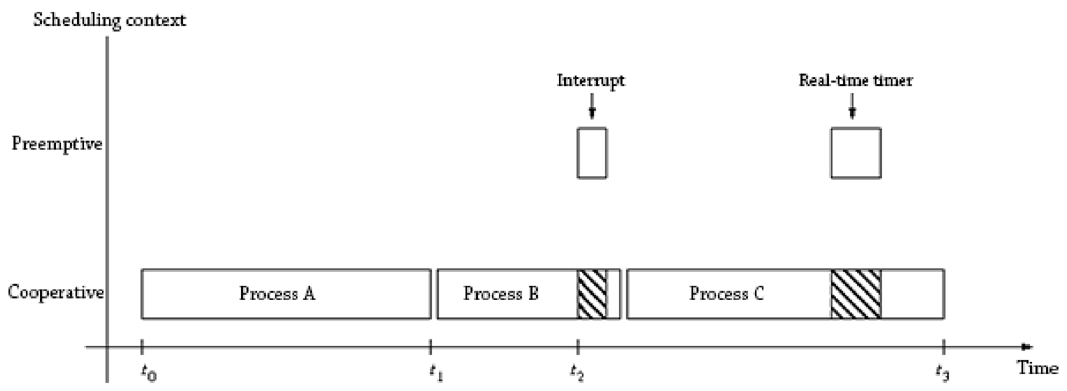


Figure 2.1: Contiki's execution contexts (Source: [32, Section Processes])

[24, page 5909] and almost all of its components are constructed with so called protothreads. Protothreads are another invention of Dunkels' and can be seen as threads specifically designed for memory constrained devices since they have a low memory overhead and do not make use of a stack [31]. They are implemented by means of C macros and are as such completely architecture independent which results in highly portable structures [24, page 5911]. In the following sections Contiki's process scheduler, network capabilities and features will be explained. Later on in Chapter 4, its most important components are discussed in great detail.

2.2.1 Process scheduler

According to the official Contiki Wiki pages [32, Section Processes], code on a Contiki system can be executed in two different contexts: cooperative or preemptive. In a cooperative execution context, processes are scheduled and executed in a sequential way. Each process is carried out until it finishes its job or indicates it is waiting for an event to happen. Only then another process can be started. Preemptive code on the other hand is able to stop cooperative running code and get executed immediately. Only after the preemptive code has completed, the system will continue the stopped process in the cooperative context. The concept of these execution contexts is displayed in Figure 2.1. Processes are always run in cooperative mode whereas hardware interrupts and real-time tasks are executed in a preemptive context.

Contiki's kernel is event-driven which means that the system reacts to specific external events and/or propagates them further to running processes. Processes in Contiki are implemented as a special kind of protothreads which give them the possibility to wait for such external events. Three kinds of events are used: synchronous events, asynchronous events and a polling event. Whenever a synchronous event is posted by a process, it is directly delivered to the destination process. Asynchronous events are posted to an event queue which will deal with the event once the others are processed. Polling is a special kind of event. It establishes a bridge between the preemptive and cooperative contexts, i.e. a poll event can be posted by a hardware interrupt and will make sure the polled process is scheduled as soon as possible. [32, Section Processes]

2.2.2 Network capabilities

When looking at the network stack of Contiki, several technologies are offered to connect a device to the network. The available protocols will be listed following the OSI network model in a bottom-up approach. An overview of Contiki's network stack can be found in Figure 2.5. The physical layer is obviously architecture-dependent. A default Contiki installation includes several radio drivers for popular radio transceivers. If however a specific radio is not supported, a software interface is provided that can be used to include support for the new hardware. This interface will be discussed in Chapter 4.

Radio Duty Cycling layer

According to the OSI network model [33], on top of the physical layer resides the Data Link layer. Contiki's developers chose to introduce an intermediate layer however, called the Radio Duty Cycling (RDC) layer [32, Section Radio Duty Cycling]. The purpose of RDC protocols is to save energy by switching off radio transceivers as much as possible while still be able to send messages across the network. Since on most platforms these mechanisms are implemented at the MAC layer, some of the protocols are named referring to that layer. Three different RDC protocols are available for use: ContikiMAC, X-MAC and Low Power Probing.

In a ContikiMAC environment, the sender will continuously send its data frame, waiting for the receiver to wake up. When the receiver wakes up and detects radio activity, it will not go to sleep again but keep listening and send an acknowledgement (ACK) once it has received the whole frame [34]. To reduce the amount of data frames that are transmitted by the sender, a phase-locking mechanism is included. By observing the timestamp of the ACK, the sender knows when the receiver was awake. Since wake-ups occur periodically, the sender can use this information to start transmitting new frames just before it expects the receiver to be awake [34]. This process is illustrated in Figure 2.2.

The X-MAC protocol is similar to ContikiMAC. Instead of continuously sending data frames, it transmits small request frames (preambles) at random moments. When it receives an ACK, it knows the receiver is listening and will start transmitting the data frame. Also in this protocol a phase-locking technique is used to reduce the amount of transmitted preambles [35]. The X-MAC mechanism is depicted in Figure 2.3.

Lastly, a protocol with a different approach can be used. When using the Low Power Probing (LPP) protocol, sender nodes start listening to the channel whenever they have data to transmit. Receiver nodes on the other hand broadcast a probe each

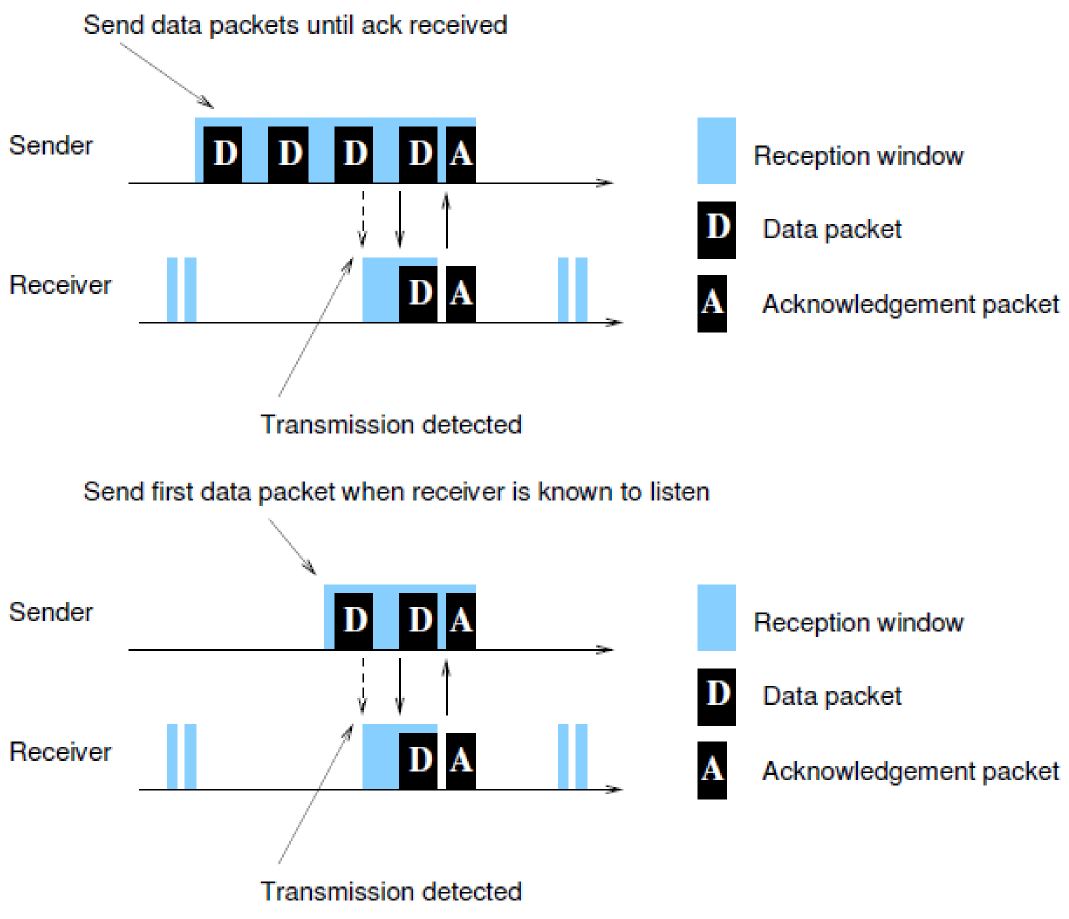


Figure 2.2: The ContikiMAC RDC mechanism, before and after phase locking (Source: [34])

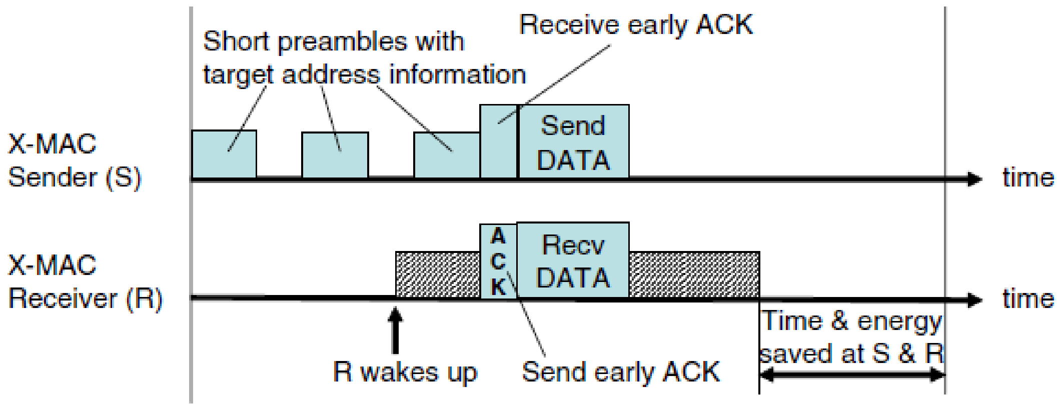


Figure 2.3: The X-MAC RDC mechanism (Source: [35])

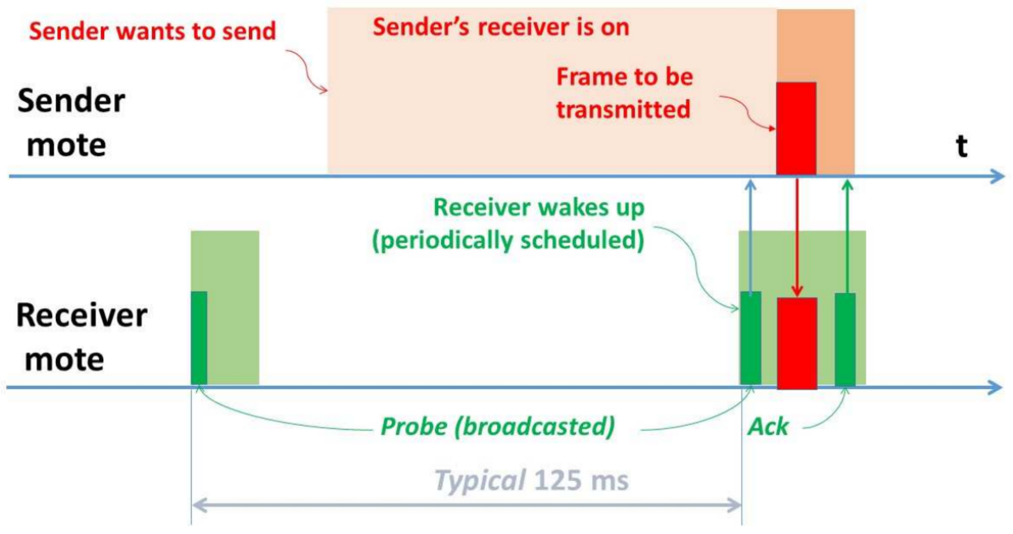


Figure 2.4: The Low Power Probing RDC mechanism (Source: ETRO-VUB Department)

time they wake up. Upon the detection of the correct probe, the sender can start to transmit its data frame [36]. Contiki implements its own version of the LPP protocol but none of the two versions makes use of a phase-locking mechanism, although it is possible to include one. The operation of LPP is shown in Figure 2.4.

Contiki provides another option, namely NullRDC. It can be used for testing and debugging purposes and will disable radio duty cycling by never switching off the radio transceiver.

Data Link (Media Access Control) layer

The previously mentioned Data Link layer consists of two sublayers: the Logical Link Control (LLC) layer and the MAC layer. The LLC layer is responsible for protocol multiplexing, flow control and error detection [33]. This layer is however not implemented by Contiki. The MAC layer is responsible for efficient usage of the shared medium by multiple devices. It has to avoid collisions and to retransmit frames whenever such collision is detected. Two protocols of this category are implemented in Contiki: CSMA and NullMAC.

CSMA short for Carrier Sense Multiple Access is the default used protocol to perform the mentioned tasks [37, Section CSMA/CD operation]. It guarantees that the sent frames successfully arrive at their destinations by keeping track of acknowledgments. If no ACK is received, CSMA will retransmit the lost data frame up to three times. The other protocol NullMAC will just pass on every frame to the RDC layer. Using this protocol, frames are sent across the network but are not guaranteed to arrive correctly [32, Section Change mac or radio duty cycling protocols]. The latter mechanism can be used for testing and debugging purposes.

Network and upper layers

In Contiki, two different network stacks can be run on top of the MAC layer, namely Rime and uIP. Rime is a lightweight layered communication stack that is designed for relaying data in sensor networks. It provides a range of different communication mechanisms, called primitives. Examples are best-effort neighbor broadcasting, reliable network flooding and single-hop or multi-hop unicast communication [38, 39].

The other option is micro IP (uIP) which is a small and simple implementation of the TCP/IP stack. Although small in size, it does not break any of the TCP/IP mechanisms and is compliant with RFC 1222, a specification that describes the requirements for Internet hosts [40]. When combining this size-optimized network stack with the 6LoWPAN and IEEE 802.15.4 protocols, it is possible to run a full TCP/IP stack on memory constrained microcontrollers. For data relaying in a multi-hop wireless sensor network while using uIP, a routing protocol is needed. The Internet Engineering Task Force specifically designed RPL (IPv6 Routing Protocol for Low-Power and Lossy Networks) for this task [41].

2.2.3 Features

Contiki aims to provide its users with features and services like a normal operating system would. Some of them are available on all platforms, others are just available for systems with less constrained hardware resources. Coffee is a file system that has a small and constant RAM footprint file and has a low performance overhead. It uses a first-fit algorithm for allocating flash memory pages to files [43]. When there would be insufficient pages to assign to a file, a garbage collector is activated to reclaim obsolete pages. Coffee also employs a wear-leveling mechanism that spreads the erasure of sectors in order to protect the flash memory from unequal wearing which could lead to damaged sectors and failures [24, page 5912].

Since it is not evident to redeploy wireless sensor networks each time they are updated in a testing stage, tools have been developed to simulate WSN environments. Cooja is such simulator and can be used to simulate the behavior of Contiki nodes at network level. Besides simulating the network stacks, it can also simulate Contiki at operating system level and machine instruction level [44]. Even though simulators are handy tools, it is important to realize that WSN nodes can behave significantly different in a real life setup.

Other available features in a complete Contiki system are a graphical user interface (GUI), web browser, web server, telnet client and screensaver [24, page 5909]. These tools will not be used in this project and will thus not be discussed.

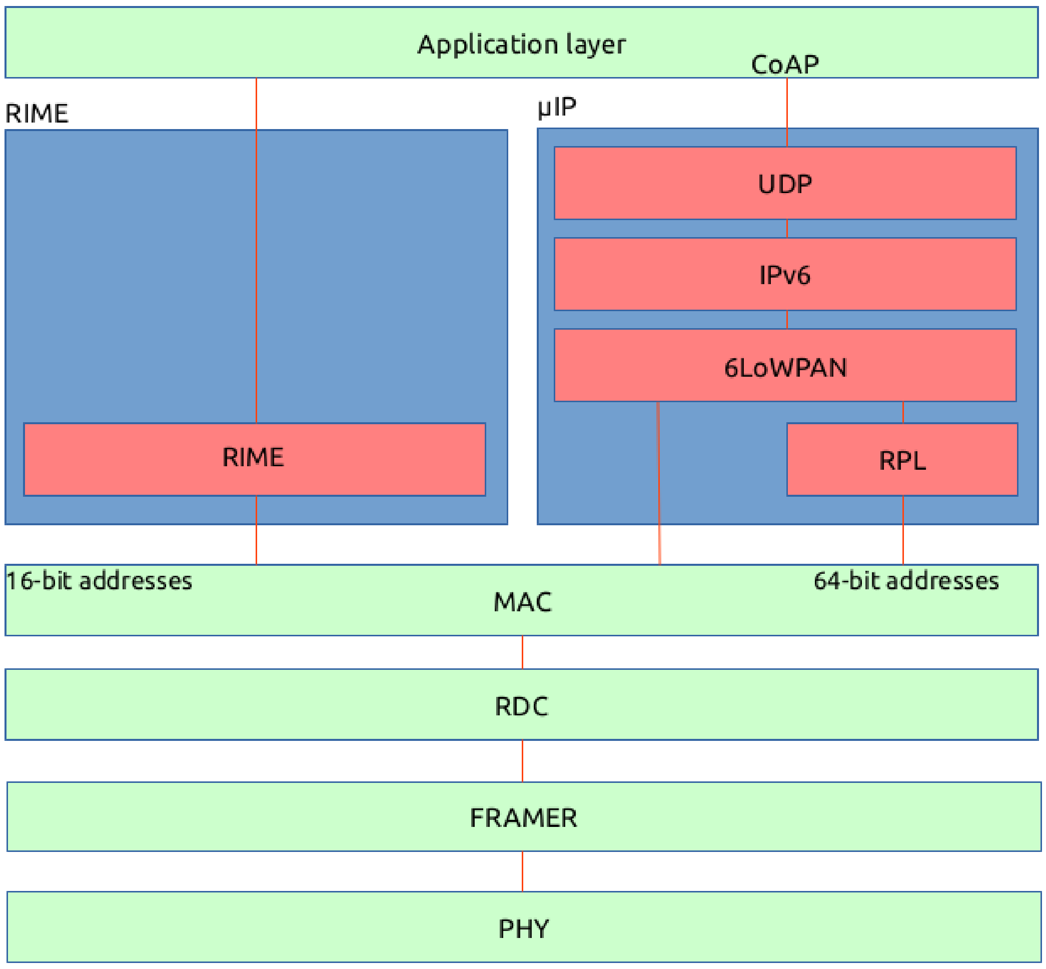


Figure 2.5: Contiki's network stack (Source: [42])

2.3 LoRa

LoRa which is an abbreviation for Long Range, is a spread-spectrum modulation technique developed by Cycleo which was acquired by Semtech in 2012 [45]. It is based on the Chirp Spread Spectrum (CSS) modulation technique, has integrated forward error correction (FEC) and operates in the Industrial, Scientific and Medical (ISM) radio band [46]. More specifically, it operates in the 868MHz band in Europe, the 915Mhz band in the United States and there are plans to implement the Australian 915-928MHz and Chinese 470-510MHz band in the near future [47, wiki page]. It is designed as a solution for LPWAN and aims to facilitate IoT, Machine-to-Machine (M2M) and smart city applications [48, Section Alliance]. On the physical level LoRa boasts characteristics like high robustness, multi-path resistance, Doppler resistance, low power usage and obviously long range communication capabilities [46]. By multiplying a reference bit pattern to the incoming radio signal, it is also possible to communicate in very noisy environments [9].

While Semtech is mainly responsible for the development of the radio transceivers (SX1272), the LoRa Alliance is the driving force behind the LoRaWAN protocol. Its specifications were available in January 2015 for early-adopter partners and were publicly released six months later. Since the technology is fairly new, Semtech and the LoRA Alliance regularly organize workshops and boot camps [49][48, Section Events]. During the (free) online workshops the LoRa modulation technique and LoRaWAN protocol are explained while promising applications are presented alongside. The boot camps are full-day trainings where participants will get acquainted with the platform. The LoRa Alliance also developed a certification program to ensure the interoperability of new hardware platforms in existing LoRa infrastructures [48, Section Certifications]. Although the deployment of LoRa is still in its initial stage, the LoRa Alliance expects a turning point in 2018 and even around three billion LoRa devices in 2022, mostly being operational in agriculture, consumer and smart building applications [45].

2.3.1 LoRaWAN

The LoRaWAN protocol is a MAC layer network protocol designed for LPWAN environments. It describes a star or star-of-stars network topology where all end devices are connected to data concentrators known as gateways. Typically these gateways are powered by and connected to a backbone infrastructure while end devices are usually powered by batteries and connect to the gateway using single-hop LoRa communication [50]. Since LoRa can be used for a broad range of applications with different requirements, three MAC profiles have been determined: Class A, B and C.

Class A devices have a bi-directional link with the gateway where the device opens two downlink transmission windows after each uplink transmission to the gateway [50]. Such devices are thus suited for applications with no downlink latency constraints. Typically they have to report their status a couple of times a day and have (almost) no actuators to power which result in extreme low power consumption [45]. Examples are omnipresent in the smart city concept like thrash containers, wearables, etc.

Class B devices implement the Class A specification but provide extra scheduled downlink reception slots. In order to synchronize the time between all end devices, the gateway broadcasts periodical beacons [50]. Typical Class B applications can be found in agriculture. Actuators report several times a day their status and are able to perform actions with a few minutes latency. Also Class B devices have a very low power consumption profile [45].

Lastly, Class C devices also implement the Class A specification but have a continuously open downlink window which is only closed when the device is transmitting data to the gateway itself [50]. These devices are usually mains powered and are used for applications with low latency requirements. Smart lighting is a perfect example of a Class C implementation since these devices have access to a power source and require immediate actuation [45].

Although LoRaWAN specifies a star topology where all devices are directly connected to a gateway, there is also interest in creating multi-hop network protocols for LoRa enabled devices. An example protocol was already implemented and shows that great surfaces can be covered with very few end devices and just one gateway when empowering multi-hop routing [46].

2.3.2 Features

Combining the physical characteristics of the LoRa enabled radio transceivers with the LoRaWAN specification, allows for network deployments with quite unique features. As said before, the main feature of a LoRa network is its long range communication capability.

Another result of the physical modulation technique is the ability of deep coverage in urban environments with low data rates. In so called 'deep urban' areas like parking lots and metro tubes, one can expect network coverage of 500 meters to two kilometers with a data rate of around 1 kilobit per second. In normal urban areas a range between five and ten kilometers should be able to get covered and should provide a data rate of 5 to 10 Kbps. In a rural environment LoRa can present its best performances by covering ten to twenty kilometers with data rates up to 37.5 Kbps [45].

As a consequence of the energy efficient LoRaWAN specification, it is expected that LoRa end devices can have a lifetime of ten to twenty years. Even a higher lifetime could be achieved when the device is powered by energy harvesting [45].

On a global scale, a LoRa network offers a high capacity infrastructure and multi-tenancy which is a result of the orthogonality of the modulation technique [46]. The whole infrastructure can be a low cost solution since one gateway is able to serve many end devices. If a certain area is well covered by gateways, an additional feature emerges. By means of triangulation, the location of a sensor can be determined without a costly GPS sensor [45, 9].

Security in LoRa communication is empowered at two levels with AES encryption. When a newly deployed end device is activated on the network, it receives two different encryption keys of 128 bits: a network session key and an application session key. The network session key is used between the end device and gateway to encrypt the message integrity code (MIC) of all transmitted frames in order to guarantee data integrity and the authenticity of the end device [50, 45]. The application session key is only known by the application owner and encrypts the payload at the application layer. This results in the privacy of application data since the network carrier will only transmit encrypted data [50, 45].

2.3.3 LoRaMote demo platform

To demonstrate all these features, Semtech developed a demo LoRa platform called LoRaMote (see Figure 2.6). The rest of this section is heavily based on the device's user guide [51] and will present the components and technical details of this platform.

The core of this platform is the WiMOD iM880A module. It is a hardware module developed by IMST and contains an STM32L151C8U6 microcontroller from ST Microelectronics and a LoRa enabled SX1272 radio transceiver from Semtech. The microcontroller provides 64 KB flash memory and 10 KB of RAM. Because the platform is a demo board and is therefore not meant to be deployed in real life. The manufacturer chose to fit it with a WiMOD iM880A module so the development work is not completely lost if the other hardware platform uses the same module. [51]

Since the LoRaMote is a handheld device, it is targeted to be powered by a battery. Hence a holder for standard 9V alkaline batteries is provided. It is also possible to power the platform by the means of a connection with a USB port. Therefore it is also equipped with a switch to be able to choose between the power sources [51]. Other components include three differently colored LEDs (yellow, red and green), 128 kilobit of EEPROM, a JTAG connector to connect hardware programmers and an I/O expander in order to hook up extra peripherals without needing extra microcontroller pins. [51]

In order to support a broad range of applications and to display its IoT capabilities, the platform is equipped with a variety of sensors. The LoRaMote possesses a three-axis accelerometer (MMA8451Q), a three-axis magnetometer (MAG3110), a proximity sensor (SX9500) based on a capacitive SAR (Specific Absorption Rate) controller, a GPS module (UP501) and an MPL3115A2 module wielding an altimeter, thermometer and pressure sensor. [51]

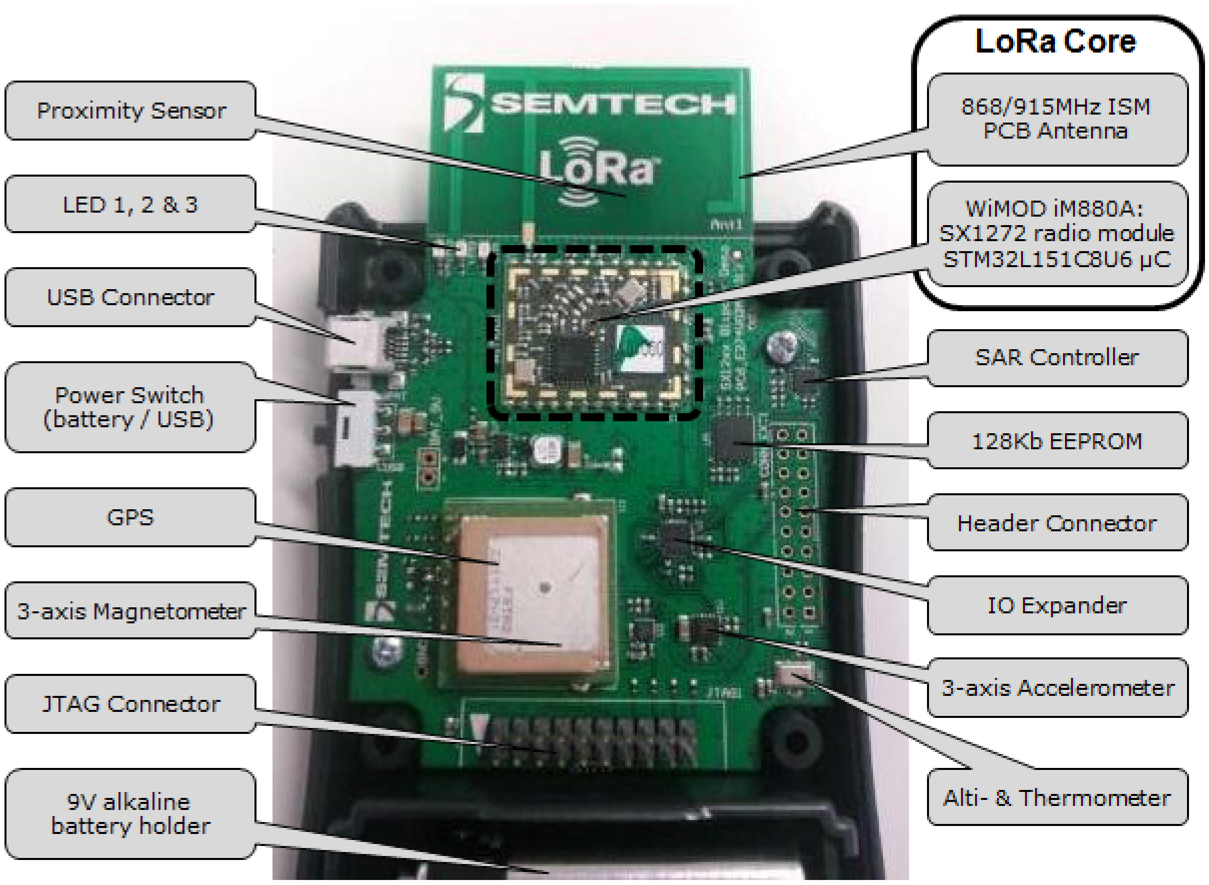


Figure 2.6: LoRaMote demo platform (Source: [51])

3 | Integration approach

The technical aspects and the properties of the technologies used in this project were presented in the previous chapter. Now, an overview will be given of how the whole project was approached, which conceptual challenges were taken into account and which problems had to be tackled.

In order to get familiar with the Contiki operating system framework, the first step was to study its components and their capabilities. This was done by means of examining simple example code. Then a small Contiki application to test the radio connectivity between two sensor nodes was written. It broadcasts dummy frames and receives frames from other broadcasting nodes using the Rime network stack. This application was tested on the Zolertia Z1 sensor node, but the goal of this project would be to run the same Contiki application on a LoRaMote demo platform.

3.1 LoRaMAC on Windows

The next step was to become acquainted with the LoRa technology. This was done through the means of self-study in order to extend the knowledge of LoRa technology that was already present at the ETRO department. One of the online workshops organized by Semtech was attended in order to get a quick introduction.

The LoRaMAC project on GitHub [47] provides a LoRaWAN compliant base system for several hardware platforms and includes preconfigured project files for a few Integrated Development Environments (IDE). A LoRa network node has to be programmed with this base system using a hardware programmer. For the execution of this project, the ETRO department provided a LoRaMote demo platform and an ST-Link/V2 hardware programmer for STM8 and STM32 microcontrollers which is displayed in Figure 3.1. Since the LoRaMAC project comes with project files for the Keil IDE and the ST-Link/V2 includes an installation and configuration guide for the same IDE, the logical decision was made to use this development suite. When the whole environment was set up, a simple LED blinking application was created to test the correct functioning of the tools.

One of the included applications of the LoRaMAC project is a bootloader. It is a small piece of software that is programmed only once onto a platform. When this application is executed, it configures the device to activate a DFU mode (Device

Figure 3.1: ST-Link/V2 programming tool

Firmware Upgrade) which enables the possibility to program new code onto the platform over a USB connection. Such setup greatly improves the ease of use of the platform and therefore its inner workings were further explored. The process of uploading code to the bootloader exists of three steps: building the applications with specific compiler and linker options, converting the executable to a DFU file and the actual programming of the executable. While doing this for the previously developed LED blink application, a problem was encountered with the DfuSe tool of ST Microelectronics. For some reason the executable could not be converted to a DFU file. This turned out to be a bug in the DfuSe software [52] and using the previous version solved the problem.

When the application finally could be uploaded to the platform, no Light Emitting Diodes (LEDs) were blinking. After a long and thorough investigation, it turned out that one of the project settings was wrongly configured. All Keil project files of the LoRaMAC project include a normal and bootloader profile for the applications. The bootloader profile was configured with the default target memory address which means that the DFU application would tried to be written to the same memory location as the bootloader code. The memory slots occupied by the bootloader are however write-protected to prevent mistakes like this. As a consequence, only a part of the application was uploaded which of course resulted in invalid code.

Once the application was built with the correct settings, the LED blinking application could be successfully uploaded to the platform. Yet, another problem arose. In the application, two LEDs were supposed to be lit at the same time but when executed on the hardware, there was a notable delay visible. Until today, this problem has not been solved and its cause is not known. Probably it is also related to the DfuSe tool since the application runs fine when it is loaded with the hardware programmer.

3.2 LoRaMAC on Linux

The LoRaMAC project comes only with project configurations for Windows development suites while the Contiki framework is mainly used on Linux environments at the ETRO department. In order to neatly integrate both technologies, it was decided to start programming the LoRaMote platform from an Ubuntu distribution.

First of all, the appropriate tools had to be found. The GCC ARM Embedded toolchain was chosen to build applications. This toolchain provides the system with a compiler, linker, debugger and other convenient tools. The st-link open-source project allows loading applications from a Linux environment to hardware platforms with the ST-Link/V2 programmer. A replacement for the DfuSe tool was found in the open-source project dfu-util which enables DFU capabilities for Linux distributions. To ease the whole setup, a script was written that would download, compile and install the aforementioned tools, install their prerequisites and set the proper USB permissions.

In order to build and upload applications to the LoRaMote demo platform, a makefile was needed. Such makefile contains all the information on how to build an executable file from source code files and a set of compiler and linker flags. The makefile for LoRaMAC applications was created by investigating the Keil project files and mapping their settings to the equivalent GCC ARM Embedded toolchain parameters. The functionality to flash program and DFU upload the built LoRaMAC applications from this makefile was implemented as well.

Using the created makefile, LoRaMAC applications can be deployed on the LoRaMote demo platform from a Linux environment. It was however impossible to get a working executable from one application, namely the bootloader. Whenever the bootloader was built using the GCC ARM Embedded toolchain, it would halt during initialization of the USB interface each time the code was executed. This problem was thoroughly investigated but no solution was found. During the whole project, the bootloader binary which was built in Keil on a Windows machine was used. It was only near the end of this project that the cause of the issue was discovered. On the official LoRaMAC GitHub someone suggested that the compiler toolchain could cause some trouble [53, 54]. By building the bootloader using an older version of the compiler toolchain, it turned out that the newer versions were indeed the source of the problem. Apparently they break some functionality which caused the bootloader code to halt. The latest working release is GCC ARM Embedded 4.8.

3.3 Deploying Contiki to LoRaMote

From the moment LoRaMAC applications could be loaded onto the LoRaMote from a Linux distribution, the main goal of this project could finally be pursued: porting Contiki to the LoRaMote demo platform so Contiki applications can use the long range communication capabilities provided by the LoRa transceiver.

Both Contiki and LoRa provide specific network protocols to employ their features. Since Wireless Sensor Networks usually have a multi-hop nature, the single-hop LoRaMAC protocol becomes quite useless. Therefore only the physical layer of LoRa's network stack will be used. Practically, this means that only certain parts of the LoRaMAC code are interesting for this project which raises the question how the available code should be facilitated.

During the integration of Contiki and LoRa, two main concepts were kept in mind: reusability and modularity. By carefully reorganizing a system implementation, several components can be reused by other people. This definitely increases the value of a project. By pursuing a modular design of an implementation, it becomes possible to switch or even remove several parts without influencing the overall working of the whole system. Such design relies on high code cohesion in modules and low coupling between the different modules.

Contiki is designed in a quite modular way. The LoRa system code on the other hand is much more intertwined on several levels. By extracting CPU specific code from the LoRaMote code, the support for a new full-featured CPU in Contiki was created. This CPU code can be reused by other platforms.

The most important part that is needed from the LoRaMAC system is of course the physical radio driver code. However, other interesting structures like Virtual COM Port handling and sensor reading functions can be kept as well for use by Contiki. By changing as little as possible to the original code and creating interfaces for the useful parts, a modular and robust integrated system can be created. This also results in a more update-proof construction. New LoRaMAC versions will seamlessly fit in the Contiki code again when they get released. Since it is easier to maintain such implementations, the lifespan of these projects increases as well.

In order to create support for Contiki on new hardware platforms, several components have to be implemented. To realize this for the LoRaMote platform, lots of different sources were used. The implementation of the Zolertia Z1 platform was studied extensively and other porting efforts were consulted. Some of those components could be partly adopted from the LoRaMAC system while others had to be implemented with the help from the STM32L1XX Standard Peripherals Library or product data sheets. The makefile that was previously created for building and uploading LoRaMAC applications served as the base for the new Contiki makefiles. The whole porting process took quite some time because it was not always clear how certain components should be implemented. Since there are no sources that completely cover Contiki's porting process, a guide was created especially for this purpose. This guide is presented in the next chapter and discusses all needed components and their implementations so it can be used as a reference work to create new ports. The results of porting the LoRaMote demo platform are shown in Chapter 5.

4 | Porting Contiki to new hardware platforms

The Contiki operating system is designed to run on resource constrained, low-power Wireless Sensor Network nodes and is constructed in such a way that it should be easy to deploy to new hardware platforms. Hence, the majority of its components is completely platform-independent. When deploying Contiki to a new platform, only a handful of platform-dependent components have to be implemented to get the whole system up and running. One of the goals of this project is to provide the Contiki community with a detailed guide on how to port the Contiki operating system to new hardware platforms by implementing these platform-dependent components. The developers state that the operating system is indeed easy to port [55, 56], but a quick search in the official mailing lists [57] teaches us that a lot of users still struggle with this matter. Existing information on porting Contiki often neglects important details or does not treat certain aspects.

Yair Hershkovitz and Demitry Lev for example wrote a porting guide [58] but merely described the basic porting components of Contiki. The same applies to Wincent Balin's documentation [59] for his entry to the MSP430 Design Contest which was held in 2006. The University of Bern held a seminar in 2009 where Cyrill Schluep presented the steps of how to port Contiki to new platforms [60], but once again it only included an enumeration of the basic steps. In 2011 George Oikonomou shared his own porting experiences, but started from an existing discontinued porting effort [61] and thus does not contain any details about how to start a new port. Zhun Shen and his team even published a paper in 2013 which describes how to port a platform from Contiki to another operating system [62]. Adam Dunkels' original presentation remains one of the best sources available although it dates from 2007 and is nowadays only available from archived copies [63]. In the same year Alexandru Stan wrote his Bachelor thesis which covers the same topic a bit more extensively [64]. Both of these works are excellent sources with examples, but are a bit outdated and lack information about low power mode implementations for microcontrollers.

All of these papers thus either lack information, are not detailed enough or are simply outdated. They also often refer to the 'native' platform which is provided within the Contiki repository and is supposed to be the simplest port for a new platform. In reality, it contains a lot of code which is not essential to get Contiki running which is confusing for users who have no experience with Contiki's inner workings. Next to the mentioned papers, more guidelines can be found on the official Contiki wiki page and within the source files [32]. As a result of all these scattered and incomplete sources of information on how to port Contiki, one can have a hard time to actually get it working while it should be in fact easy to do. This is the main reason why there was a need for a unified document with a detailed guide.

In the rest of this chapter an enumeration of 'to be ported' components will be presented as well as the steps to be taken on how to do it. All will be accompanied by examples from our own Contiki port to the LoRaMote demo platform which wields an STM32L151C8 microcontroller [51, 47], using the STM32L1 Standard Peripherals Library [65].

4.1 Contiki's directory structure

The main Contiki directory [66] contains multiple files and folders. Those with any importance will be discussed although they are not necessarily of interest for porting purposes. Contiki's base system resides in the `/core` directory and has no need to be ported [63]. Its code should only be consulted for more information on the inner workings, but should not be changed in order to 'fix' a process. To perform a port there are two relevant directories, namely `/platform` and `/cpu`. Only these two can contain architecture dependent code. Typically, the cpu folder will be used by all platforms with the same microcontroller and contains specific code like startup files, libraries, timers, watchdogs, multi-threading, low power mode controllers and an ELF loader. Generally the platform folder contains all other code which is not related to the CPU. This commonly includes the main function, LED control, sensor reading and all other functions the board is capable of. A last directory that will be used is `/tools`. This is the storage place for all tools that will be used during the build and upload process (e.g. hardware programmer, debugger, mote listing tools, terminal, etc.). One file of interest is Makefile.include. It contains the make instructions for the Contiki base system and will build the whole project once a destination platform has been chosen.

The first step of porting Contiki should be creating the appropriate subfolders in the `/cpu`, `/platform` and `/tools` directories. The two first directories should be provided with their own makefile, which are respectively `/cpu/{CPU_NAME}/Makefile.{CPU_NAME}` and `/platform/{PLATFORM_NAME}/Makefile.{PLATFORM_NAME}`. The contents of these files include instructions to compile, link and upload code. Their implementation will be discussed later. Then two new folders should be made in the personal platform directory, named 'dev' and 'net'. The dev folder will contain the files with the low level code for sensors while the net folder will contain the low level code for the radio driver.

Using this information, the following directories should be created for the LoRaMote platform example:

- `{CONTIKI_PATH}/cpu/arm/stm32l1`

- `{CONTIKI_PATH}/platform/loramote`
- `{CONTIKI_PATH}/platform/loramote/dev`
- `{CONTIKI_PATH}/platform/loramote/net`
- `{CONTIKI_PATH}/tools/loramote`

Next, the STM32L1 Standard Peripherals Library should be placed in the `/cpu/arm/stm32l1` directory, together with linker scripts and system startup files for the STM32L1 that can be found in the LoRaMac GitHub repository [47]. These files will be crucial to build the whole project for this platform.

4.2 Components to port

The native platform which is included in Contiki's source code is often referred to as a porting template. Although it indeed contains all basic necessary implementations, it is not always clear what is happening on each line of code when one has no or limited foreknowledge as mentioned before. Needless to say, when there should be a platform already available which is similar to the one that is being ported, it is more interesting to use that code as a starting point. Some caution is advised here since not all ports are completely finished and thus can result in unexpected results. Either way, it is always easier to start from existing code in order to have the basic structures.

In the following section the different porting components will be discussed, accompanied with examples from the LoRaMote demo board. Since the multi-threading library is not used often and the ELF loader requires great knowledge of the ELF format and the CPU architecture [63], these components are out of the scope of this paper and are omitted.

4.2.1 Configuration files

Each platform should contain two configuration files: contiki-conf.h and platform-conf.h. The Contiki configuration file contains several configuration options related to the operating system like C compiler instructions, C types, uIP settings, duty cycling preferences, apps and widgets preferences, etc. [63]. The platform configuration file can be used for platform specific settings like clock configuration and constants used by the platform's clocks, sensors or other components. These configuration files can be copied from the native platform or a platform that is similar to the one that is being ported. Later on during this guide, several configuration options will be added to these files.

Practically, the `/platform/native/contiki-conf.h` configuration file is copied to `/platform/loramote/contiki-conf.h`. The clock_time_t and CLOCK_CONF_SECOND definitions should be removed from this new Contiki configuration file. Then a new file `/platform/loramote/platform-conf.h` is created, containing the default clock configuration.

```
#ifndef __PLATFORM_CONF_H__
#define __PLATFORM_CONF_H__

typedef unsigned long clock_time_t;
typedef unsigned long long rtimer_clock_t;

#endif /* __PLATFORM_CONF_H__ */
```

4.2.2 Clock module

The clock module (clock.c) is one of Contiki's most essentials components as it handles most of the timed events used by the system and its applications. The objective of the clock module is to generate a certain amount of ticks each second and to continue processes if their etimer or ctimer have expired. Such pending processes are started by calling the etimer_request_poll() function. The amount of ticks per second can be chosen freely, but is usually a power of two. The reason for this is that their divisions can be optimized greatly [67, page 92]. A number between 64 and 512 ticks is common, depending on which granularity is requested.

According to Contiki's official documentation on timers, there are six components of the clock module API that need porting [32, Section Timers].

- **clock_time_t clock_time();**
  Returns the amount of ticks that have passed since the initialization of the clock.
- **unsigned long clock_seconds();**
  Returns the amount of seconds that have passed since the initialization of the clock.
- **void clock_delay(unsigned int delay);**
  Shortly delays the processor by executing some no-operations (NOPs).
- **void clock_wait(int delay);**
  Delays the system for a certain amount of clock ticks.
- **void clock_init(void);**
  Initializes the clock module.
- **CLOCK_CONF_SECOND;**
  Defines the number of ticks per second.

For the LoRaMote example, the STM32 Systick structure has been used. During clock initialization the CPU is told to fire an interrupt several times per second in order to increase the tick counter and check for processes which are ready to continue their tasks. The full implementation can be found in Appendix A.1. In the implementation of the Systick interrupt, two ENERGEST calls can be found. These are used by Contiki to create an estimation of the power consumption. More of these calls will be found in the implementation of other components. The RELOAD_VALUE constant uses two other constants, namely F_CPU and CLOCK_SECOND. F_CPU refers to the frequency of the main processor and should be defined in the platform configuration file. CLOCK_SECOND is a constant defined by Contiki and will take the same value as CLOCK_CONF_SECOND. CLOCK_CONF_SECOND should be defined in the platform configuration file as well and is never supposed to be called directly.

The following two instructions are thus added to the platform-conf.h file.

```
/* The frequency of the main processor */
#define F_CPU                   32000000
/* The desired amount of ticks per second */
#define CLOCK_CONF_SECOND       256
```

4.2.3 Rtimer

The rtimer (rtimer-arch.c) is an independent clock. It differs from the clock module by giving extensive control to its users and not being used by Contiki itself. Its sole purpose is to precisely schedule one event at a time and it is used by configuring a wake up time and a callback function. This callback function profits from being preemptive, meaning that when the rtimer expires the current process will be stopped in favor of executing this rtimer callback function [32, Section Processes & Section Timers]. The user callback function is triggered by executing the rtimer_run_next(). The amount of ticks that this clock produces is much higher than the main clock for great precision, typically a power of two ranging between 8192 and 32768.

In order to port the rtimer, the following items have to be implemented [32, Section Timers]:

- **RTIMER_CLOCK_LT(a, b);**
  Should return TRUE if ’a’ is less than ’b’, otherwise FALSE.
- **RTIMER_ARCH_SECOND;**
  Defines the number of ticks per second.
- **void rtimer_arch_init(void);**
  Initializes the rtimer module.
- **rtimer_clock_t rtimer_arch_now();**
  Returns the amount of ticks that have passed since the initialization of the clock.
- **int rtimer_arch_schedule(rtimer_clock_t wakeup_time);**
  Schedules a callback to a user function.

In the LoRaMote example the STM32 TIM structure is used to implement the rtimer. The full implementation can be found in Appendix A.2. As in the clock module example, the initialization of this timer configures an interrupt that will handle the rtimer tick counter and optionally call the users’ callback function. This implementation is not necessary. It can also be implemented without a tick counter by configuring an interrupt that will immediately call the callback function. This will be shown in Section 4.2.11 discussing low power mode. The used RTIMER_SECOND constant is defined by Contiki which refers to RTIMER_ARCH_SECOND. Just like the CLOCK_CONF_SECOND constant, RTIMER_ARCH_SECOND should never be called directly.

Again two constants should be added to the platform-conf.h file.

```
/* The desired amount of ticks per second */
#define RTIMER_ARCH_SECOND              32768
#define RTIMER_CLOCK_LT(a,b)            ((signed short)((a)-(b)) < 0)
```

4.2.4 Watchdog

A watchdog is a hardware device that is able to recover a microcontroller from system errors [68]. It makes use of a counter that will count down to zero. The system should reset this counter to its original value on a regular basis. If this would fail and the counter reaches zero, the watchdog will judge that the application is not executing in a normal or expected way and will reset the microcontroller. A variant of the normal watchdog is a windowed watchdog. The principle of the countdown counter stays the same, but an extra condition is imposed. If the application resets the counter too soon, the windowed watchdog will suspect a malfunction too and will reset the microcontroller.

Contiki's watchdog interface (watchdog.c) requires five functions to be implemented:

- **void watchdog_init(void);**
  Initializes the watchdog module.
- **void watchdog_start(void);**
  Starts the watchdog countdown process.
- **void watchdog_stop(void);**
  Stops the watchdog countdown process.
- **void watchdog_periodic(void);**
  Resets the watchdog's countdown counter.
- **void watchdog_reboot(void);**
  Restarts the watchdog device.

Both watchdog variants are implemented for the LoRaMote example in Appendix A.3. The windowing feature of the windowed watchdog is not used here. The STM32 watchdogs cannot be disabled once started [69, page 135 & 553], therefore this function is implemented as a stub. Some new constants are to be added to the platform configuration file. These give the possibility to select the preferred variant and to control the watchdog activation and its timeout value.

```
/* LSI clock frequency */
#define F_LSI                           37000
/* Start watchdog */
#define ENABLE_WATCHDOG                 1
/* Use independant watchdog */
#define WATCHDOG_USE_IWDG               1
/* Independant watchdog timeout in milliseconds */
#define WATCHDOG_IWDG_TIMEOUT           250
```

4.2.5 LEDs

One of the most basic ways to get some feedback from the platform is by using LEDs. Usually it is also one of the easiest components to implement. By default, Contiki recognizes up to three differently colored LEDs: red, green and yellow/blue. The control interface (leds-arch.c) makes use of bitmasks to get and set the current LED configuration. The form of this bitmask can be chosen by the platform itself by defining the LEDS_RED, LEDS_GREEN, LEDS_YELLOW and LEDS_BLUE constants as unique power of two values. The LEDS_ALL constant then should be defined as the maximum value than can be achieved by adding all the separate LED values. The default bitmask definition can be found in the `/core/dev/leds.h` file. The Contiki LED interface can be used once the following three functions are implemented:

- **void leds_arch_init(void);**
  Initializes the LED hardware structures.
- **unsigned char leds_arch_get(void);**
  Returns a bitmask of the currently active LED configuration.
- **void leds_arch_set(unsigned char leds);**
  Lights the LEDs according the given bitmask.

The LoRaMote example makes use of Contiki's default bitmask and uses the LoRaMac GPIO interface to handle the LED configuration. Its full implementation can be found in Appendix A.4. In order to indicate that the platform features LEDs, an extra constant can be added to the platform configuration file.

```
/* Indicate that LEDs are available to use */
#define PLATFORM_HAS_LEDS               1
```

4.2.6 Serial line driver

Solely using the LEDs as feedback is a bit restrictive. It is much more interesting to be able to get some output by the means of text messages. This can be achieved by using a hardware debugger or by configuring a serial line connection to the board. The usage of a hardware debugger will be discussed in the Code upload mechanisms section (4.4), while this section will focus on the establishment of a serial line connection.

Whenever a printf command is executed in the C language, its parameters are passed on through several other commands until the moment it reaches a specific system call. On a machine with an operating system, the system will make sure the output is redirected correctly to an output console. On embedded systems however, such underlying I/O dispatching service is not available. Therefore the main principle of getting text output from bare metal platforms is to implement a so called 'sink'. A sink is a framework that will capture all standard I/O calls and redirect them elsewhere. The destination of this redirection or a combination of destinations can be chosen from a range of different peripheral possibilities. The message can be relayed over a network for example or it can be shown on an LCD display. The goal of this module is however to redirect the message through a UART/USART hardware device so it can be read by a connected terminal application. Such UART/USART device is responsible for translating a sequence of bits to a serial series of 'logical voltage levels' [70]. These virtual voltages can then be used by a serial communication driver to transmit them physically. Usually, a terminal application physically connects to the board using an RS-232 serial port. On newer boards however, it is more common to use USB Virtual COM Ports which emulate RS-232 connections. So in order be capable of using printf commands in Contiki applications, this I/O intercepting sink framework needs to be implemented. The functions that are required by this framework depend on the used compiler toolchain.

- IAR C Compiler:
  size_t __write(int handle, const unsigned char *buf, size_t bufsize);
- ARM C Compiler:
  int fputc(int ch, FILE *f);
- GNU C Compiler:
  int __io_putchar(int ch);
  size_t _write(int handle, const unsigned char *buffer, size_t size);

Whereas Contiki uses the default sink framework for output redirection, it does provide an interface for user input handling [32, Section Input and Output]. The first step should be to enable an interrupt on the UART/USART device whenever it receives data. The interrupt callback function then should pass on the received character to Contiki's serial_line_input_byte() function. This serial line interface will buffer all received characters until it detects a newline character. Whenever the newline character is entered, Contiki will send the received text string to all active processes for further usage.

In the LoRaMote example project the GNU C Compiler was used. The _write() function should call the __io_putchar() function for each character in its buffer. Then each character can be redirected to the desired end device, in this case the Virtual COM Port structure of the USB interface. All implementations can be found in Appendix A.5: the _write() function in `syscalls.c` and __io_putchar() in `serial-line-arch.c`. The latter file also includes the code for serial user input handling.

4.2.7 Sensors

Contiki provides its users with a unified sensor interface which can be used to retrieve the current measured value or to get notified on sensor changes. This way all sensors can be accessed and controlled in a similar way regardless of the underlying peripheral technology and microcontroller. As a result of this abstraction, applications written for Contiki can become completely platform independent. The Contiki sensor structure consists of the seven functions listed below and interrupt handlers if needed. Note that only the first four functions have to be implemented since Contiki provides the implementation of the last three.

- **static void init(void);**
  Initializes the sensor's hardware
- **static void activate(void);**
  Activates the sensor.
- **static void deactivate(void);**
  Deactivates the sensor.
- **static int value(int type);**
  Returns the sensor's current value.
- **static int active(void);**
  Returns the sensor's current activation status.
- **static int configure(int type, int value);**
  Contiki's interface to control the sensor.
- **static int status(int type);**
  Contiki's interface to get the sensor's status.

In the sensor showcase found in Appendix A.6, the LoRaMote's SX9500 proximity sensor is implemented as a button. It is a device that is connected to the SX1509 I/O Expander, therefore a cascading interrupts implementation is needed. When the SX9500 detects a value higher than the configured threshold, an interrupt will be generated. The SX1509 will detect that the SX9500 fired an interrupt and will generate another interrupt itself. This interrupt will be captured by the STM32 EXTI structure which will call Contiki's sensors_changed() function in order to let the system know that a sensor event occurred. The implementation of this cascaded interrupts construction is pretty low level and required the help of the datasheets of the SX1509 I/O Expander [71], the SX9500 proximity sensor [72] and the iM880A microcontroller module [73]. Lastly, to indicate that this platform wields a button, a new entry is made in the platform configuration file.

```c
/* Indicate that a button is present for use */
#define PLATFORM_HAS_BUTTON             1
```

4.2.8 Node MAC address & node ID

In order to be able to connect to a device on the data link layer, it needs a unique MAC address. This MAC address will be used by the network stack to send encapsulated network packets, called frames, to other devices. Most network interfaces already come with a unique MAC address, but sometimes it can happen that they have to be generated. If a MAC address has to be generated, other unique properties of the device are to be used. Contiki's devices also bear a node ID which is usually derived from the MAC address. It can also automatically derived from another source or even manually set. The way in which this ID is assigned can be freely chosen by the developer. Contiki requires two functions (node-id.c) related to the node's MAC address and ID:

- **void node_id_restore(void);**
  Globally sets the node's MAC address and ID.
- **void node_id_burn(unsigned short id);**
  Programs a chosen node ID to special memory location on the platform.

The STM32L1 microcontroller features three unique ID registers for a total of 96 bits [69, page 882-883]. In the LoRaMote's implementation in Appendix A.7, these registers are used for the generation of its MAC address and node ID.

4.2.9 SLIP driver

In Section 4.2.6 the process of redirecting the device's input and output was described. Of course, this way of communication is a bit basic and thus restrictive. There are other technologies available which allow a more complicated way of communicating. One of them is the Serial Line Internet Protocol or SLIP in short. The protocol encapsulates IP packets in order to send TCP/IP traffic over serial line connections [74].

Contiki provides an interface (slip-arch.c) for easy SLIP implementing on new platforms which is similar to the one for serial line handling. This time the configured serial line input interrupt should call the slip_input_byte() function. On the other hand, the output redirection is handled a little differently. There are no more toolchain dependent sink functions involved anymore, just one function that is required by Contiki, namely slip_arch_writeb().

- **void slip_arch_init(unsigned long ubr);**
  Initializes the serial connection for SLIP handling. Enable the serial input interrupt here.
- **void slip_arch_writeb(unsigned char c);**
  Forwards all outgoing SLIP packets to the serial line connection.

The LoRaMote's SLIP driver implementation can be found in Appendix A.8. Once again, the USB's capability of emulating a serial COM port has been used.

4.2.10 Radio driver

When consulting Contiki's Wiki pages on how to implement a port for a microcontroller's radio capabilities, not much useful information will be found. All radio related pages treat higher level topics like RPL, radio duty cycling and MAC protocols. Adam Dunkels' porting Contiki workshop only reveals a few hints [63] and in the meanwhile Contiki's radio interface has changed. Even though radio communication is a key feature in the existence of Contiki, the steps that need to be taken to implement a new radio remains highly undocumented. By examining the source of existing radio drivers however, a radio_driver struct can be found. Each new radio implementation should use this C struct to define its own radio driver functions. The following list contains the most important functions of the data structure that are required to get a working radio device.

- **int (* init)(void);**
  Initializes the radio hardware.
- **int (* prepare)(const void *payload, unsigned short payload_len);**
  Prepares a packet to be sent by the radio.
- **int (* transmit)(unsigned short transmit_len);**
  Sends the previously prepared packet.
- **int (* send)(const void *payload, unsigned short payload_len);**
  Prepares & transmits a packet.
- **int (* read)(void *buf, unsigned short buf_len);**
  Copies a received packet to Contiki's input buffer.
- **int (* channel_clear)(void);**
  Performs a Clear Channel Assessment (CCA) to find out if another device is currently transmitting.
- **int (* receiving_packet)(void);**
  Checks if the radio driver is currently receiving a packet.
- **int (* pending_packet)(void);**
  Checks if the radio driver has just received a packet.
- **int (* on)(void);**
  Turns the radio on.
- **int (* off)(void);**
  Turns the radio off (or go to into Low Power Mode).

Next to these interface functions provided by Contiki, some other interrupt functions should be implemented before a working radio driver can be achieved. These interrupt functions are completely platform dependent, but will usually include callbacks for radio events like received packet, transmitted packet and maybe error handling functions.

Since this porting project reuses parts of the LoRaMac system, the implementation of the LoRaMote's radio driver for Contiki mainly consists of forwarding all data between Contiki and the SX1272 radio driver functions. The precise implementation is presented in Appendix A.9.

4.2.11 Low power mode

Since lots of Internet of Things and wireless sensor network applications wait for specific events to occur, the processor keeps itself busy most of the time by polling all processes to know if there are new instructions to execute. A significant amount of power can be saved by putting the microcontroller in a low power mode (LPM). In such a low power mode several peripherals of the board are deactivated, for example the main processor. By using the low power mode capabilities of the board, its lifetime can be extended greatly since it is generally powered by a sole battery. Usually a board has several low power modes. The datasheets should then be checked to select the most suited one. When there are no more processes to run, the microcontroller should switch to the chosen low power mode. The microcontroller exits the low power mode by powering up all needed peripherals once the clock module or the rtimer detects that a process is pending. Depending on the selected low power mode it is possible that the hardware clocks on which Contiki's clock module and rtimer are based, are deactivated as well. In this case both timers just stop ticking which will break the application's normal execution. This is the main reason why the clock module and rtimer should be using an external clock that is not deactivated by the selected low power mode.

The STM32L1, LoRaMote's microcontroller, features several low power modes. After analyzing the differences between them in the STM32L1 datasheet [75], the STOP mode was chosen. It offers a fast wake up time while still greatly reducing the power consumption. The STOP mode also deactivates the main processor. Both the clock module and the rtimer use it as their clock source which means that their functionality will render useless once the low power mode is entered. Therefore these modules need to be implemented with an external clock. The STM32L1 real-time clock (RTC) is used for this purpose since it can be sourced from the low speed external oscillator (LSE). The RTC structure initialization is shown in Appendix A.10. The functions to control the low power mode can be found in Appendix A.11.

Contiki's new clock module implementation makes use of the RTC Wake Up interrupt. This interrupt is fired a couple of times per second to fulfill the clock module's purpose. The amount of ticks per second stays configurable by changing the CLOCK_CONF_SECOND constant in the platform configuration file. The new code for the clock module with LPM support is set out in Appendix A.12.

In the previous implementation of the rtimer, event scheduling was done in a similar way as in the clock module: by generating interrupts many times per second. By using the RTC Alarm structure, it is possible to schedule a callback time on the hardware itself. The interrupt will only be fired once the process' rtimer expires. As a result, the amount of interrupts is greatly reduced. The full implementation of the rtimer with LPM support can be found in Appendix A.13.

Lastly, again two extra constants are added to the platform configuration file.

```
/* The frequency of the LSE clock */
#define F_LSE                           32768
/* Time in microseconds to wake up from low power mode */
#define MCU_WAKE_UP_TIME                3400
```

4.3 Main function

Contiki's main function implementation (contiki-{PLATFORM_NAME}-main.c) is rarely discussed in porting guides, although when not configured correctly it can be an unsuspected source of weird bugs. It loads the required Contiki components, initializes the platform's hardware and configures its network capabilities. Then it will start the user processes and schedule them accordingly. Some caution is required in the order of initialization. Both the etimer and sensor modules are implemented as Contiki processes, therefore they need to be initialized after the Contiki process initialization has been completed. Since ctimers are implemented as etimers, its initialization should happen after the etimer process has started. The core of this main function is the process scheduler loop. It is an infinite loop that will schedule the started processes according their cooperative context [32, Section Processes].

```
// Initialize here the hardware, Contiki components & network stack

/* Start the process scheduler loop */
while(1) {
  int r;
  do {
    r = process_run();
  } while(r > 0);

  // Enter low power mode here
}
```

The implementation of the LoRaMote main file for Contiki is demonstrated in Appendix A.14. Its process scheduler loop is a bit more extensive in order to integrate the low power mode and the proper execution of the standard I/O redirection.

4.4 Code upload mechanisms

Now that all components are covered, they have to be programmed on the platform somehow. The idea consists of compiling the whole project and generating a single binary file. This binary file then needs to be programmed on the microcontroller. There are two main techniques that can be used. Either the application gets flash programmed to the microcontroller using a hardware programmer or a bootloader that will put the device in a firmware update mode (DFU) is installed so applications can be uploaded through a USB connection.

Such hardware programmer often comes with debug and I/O redirection capabilities and is thus indeed a great tool to use during the early stages of the porting project. A downside of this method is that applications which use the I/O redirection sink framework will not work without the hardware debugger. Since the system will not know how to handle these I/O calls as a consequence of the missing sink framework, the whole system will halt once the hardware debugger is no longer used. In order the get these applications running without such debugger, the serial line driver discussed in Section 4.2.6 must be implemented.

When the hassle of hardware programmers is not desired, a bootloader can be used. This bootloader initially must be installed by the hardware programmer though. Once installed correctly, it is possible to load new applications to the platform by the means of a USB connection. Note that applications launched from bootloaders can require different build options than those which are not launched from a bootloader. As long as the serial line driver is not set up, the microcontroller will still require the hardware debugger in order to handle any I/O calls. Once both the bootloader and serial line driver are correctly set up, the platform can be programmed and interacted with by just a USB connection.

To program Contiki onto the LoRaMote board, the ST-Link in-circuit debugger/programmer [76] is used. Its debug and I/O redirection capabilities are empowered by the usage of OpenOCD [77]. This I/O redirection mechanism is called semihosting on ARM targets [78, page 8-2]. The LoRaMac project [47] includes a bootloader to be used for this platform and is placed in the `/tools/loramote` directory. In order to open DFU capable devices and upload new firmware over USB, the dfu-util tool [79] is used. Since these three tools are usable by all STM32 devices, a new folder is created namely `/tools/stm32`. Each tool was downloaded and built to this directory. Further usage of these tools will be discussed in the next section when the required makefiles are explained. The custom OpenOCD configuration file for the LoRaMote platform can be found in Appendix A.15.

4.5 Makefiles

The last step in creating a successful port of Contiki is being able to set up the makefiles. These files contain the instructions to compile all source files, link them into a binary and upload it to the microcontroller. As mentioned before, several makefiles are needed to create a working port. Contiki's top-level makefile takes care of its core components. Hence a makefile for the microcontroller drivers is required alongside a separate makefile for the platform dependent code. When building an application for Contiki, the desired target platform is indicated by defining the TARGET environment variable. Contiki will automatically include the chosen platform's makefile. In this makefile there should be a reference to the makefile of the platform's microcontroller which makes the makefile chain complete.

The objective of the platform makefile is to specify the target's source files and the directories where they reside. These must respectively be defined in the CONTIKI_TARGET_SOURCEFILES and CONTIKI_TARGET_DIRS variables [63]. As said before, there must be an inclusion of the CPU's makefile defined. Furthermore, this file should contain the instructions to detect connected devices, to program binaries using the hardware programmer or bootloader and to connect to the device's serial output. If the platform needs specific Contiki components or compiler flags (CFLAGS), these also should be defined here.

The microcontroller's architecture source files, like startup files, C library and CPU drivers must be listed in the CPU makefile. It is also here that the support for one or more compiler toolchains and their linker scripts should be implemented. For a successful build, each toolchain requires its own set of compiler and linker flags (LDFLAGS). Contiki features a default set of makefile rules to compile and link all source files. If for some reason these makefiles rules do not suffice, they can be redefined here.

The example makefiles for the LoRaMote platform can be found in Appendix A.16 and A.17.

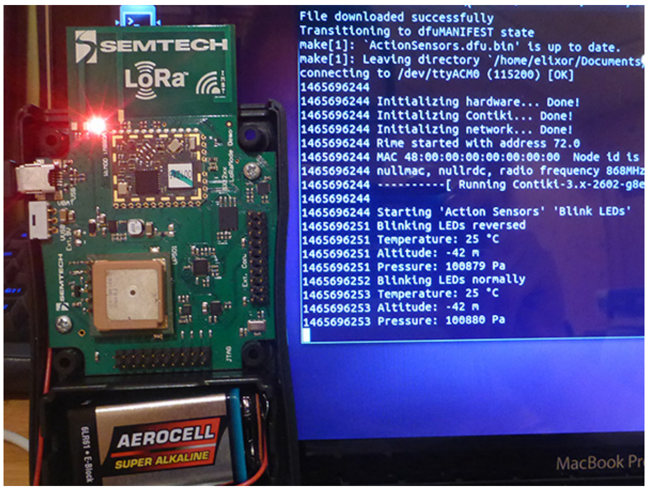
5 | Testing the Contiki enabled LoRaMotes

Once all components presented in the previous chapter were successfully implemented for the Contiki enabled LoRaMote demo platform, several tests were performed. First of all, some of the peripherals and sensors present on the board were tested. A small application was created and prints out the current temperature, altitude and atmospheric pressure when the node's button is pressed and sequentially blinks the LEDs. This application is shown in Figure 5.1 and confirms that the clock module, serial output, and LED and sensor interfaces work as expected. Note that the altitude sensor returns a weird value. It is however the official LoRaMac implementation of the sensor. Other test applications were created to check the correct functioning of other components such as the rtimer and serial input.

To verify that LoRa's radio transceiver is operational when running Contiki, the radio test application which was written in the beginning of this project was deployed to the LoRaMote platform. The correct functioning of the radio transceiver is indicated by the LED's activity. A blinking red LED shows that frames are being transmitted, a green LED that a frame is received and the yellow LED glows when multiple frames are received consecutively. Initially, only dummy frames with arbitrary data were sent to check if a connection between two nodes could be set up. Later on, useful data like sensor readings were included for transmission as well. Lastly, some actuators that react on incoming sensor readings were implemented. In this manner this simple interactive application can act as a proof for the successful port of Contiki to the LoRaMote platform.

5.1 LoRa's range performance

In order to evaluate the performance of LoRa, a comparison was made between the popular Zolertia Z1 and the LoRaMote platform. The same radio testing application was deployed to both platforms and the maximal distance that could be covered between two nodes without losing connectivity has been measured. The evaluation was executed in a residential area with a significant amount of bushes. The maximal distance that the Zolertia Z1s could cover in these surrounding without external antenna was between 7 and 10 meters. This result did not come as a surprise since



```
File downloaded successfully
Transitioning to dfuMANIFEST state
make[1]: `ActionSensors.dfu.bin' is up to date.
make[1]: Leaving directory `/home/elixor/Documents
connecting to /dev/ttyACM0 (115200) [OK]
1465696244
1465696244 Initializing hardware... Done!
1465696244 Initializing Contiki... Done!
1465696244 Initializing network... Done!
1465696244 Rime started with address 72.0
1465696244 MAC 48:00:00:00:00:00:00:00  Node id is
1465696244 nullmac, nullrdc, radio frequency 868MHz
1465696244 ----------[ Running Contiki-3.x-2602-g8e
1465696244
1465696244 Starting 'Action Sensors' 'Blink LEDs'
1465696251 Blinking LEDs reversed
1465696251 Temperature: 25 °C
1465696251 Altitude: -42 m
1465696251 Pressure: 100879 Pa
1465696252 Blinking LEDs normally
1465696253 Temperature: 25 °C
1465696253 Altitude: -42 m
1465696253 Pressure: 100880 Pa
```

Figure 5.1: Testing the LoRaMote's peripherals & sensors

the internal antenna of the Zolertia Z1 is sensitive to orientation, environment and weather conditions [80]. Generally, they performed better in surroundings with few obstacles or in case both communicating sensor nodes were in line-of-sight.

When using the LoRa modulation technique for wireless communication, several settings can be changed to set up the desired configuration. Two of these settings are directly related to the range of communication, namely transmit power and spreading factor. Obviously, higher transmit power can result in a larger communication range. The default setting of +14 dBM was used. This is also the maximal output power allowed by the European Telecommunications Standards Institute (ETSI) [81]. The spreading factor (SF) in a CSS modulation technique is a setting that controls how many chips will be used to represent a symbol. A higher spreading factor results in lower data rates and a longer air-time, but is more resistant to interference which leads to larger communication ranges.

Three different spreading factors were tested on the LoRaMote: SF7, SF9 and SF12. The default radio test application was used to evaluate the communication range with SF7 and SF9. The application had to be adapted when testing SF12 however. The air-time of the transmitted frames was so long that they were constantly colliding. Therefore the bidirectional testing function was disabled when using SF12.

During the experiment, one LoRaMote was placed in the center of a garden (see Figure 5.2) while the other LoRaMote was moved around (see Figure 5.3). To get a plausible view on LoRa's communication range performance, two different scenarios were assessed. In the first case the range was tested while moving away from the residential area (south) whereas in the second case the test was conducted while moving through the residential area (north). The results can be found in Table 5.1 and are depicted in Figures 5.4, 5.5 and 5.6.

It is clear that the LoRa signal level weakens significantly because of bushes, trees and houses on its path since much larger distances can be crossed in less densely built areas. Another notable result is the performance of spreading factor 12. It has

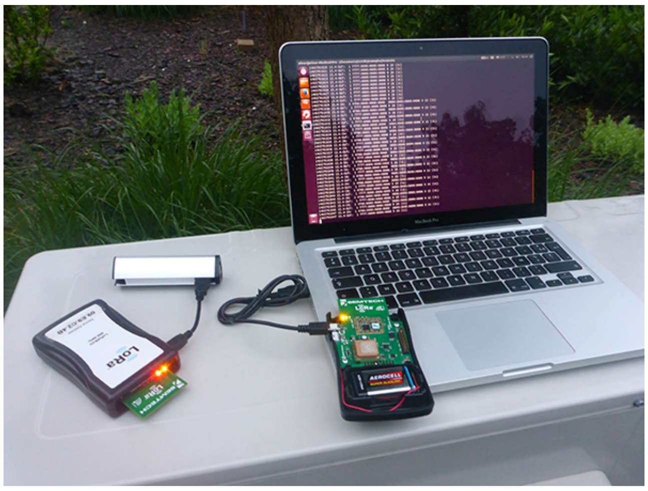


Figure 5.2: LoRaMote 1 connected to the computer, LoRaMote 2 attached to a portable power bank

Figure 5.3: LoRaMote 2 being moved around in the neighborhood

	SF7	SF9	SF12
Moving through RA	318m	362m	608m
Moving away from RA	637m	952m	1040m

Table 5.1: Comparison of LoRa's maximal communication ranges in a residential area (RA) using different spreading factors (SF)

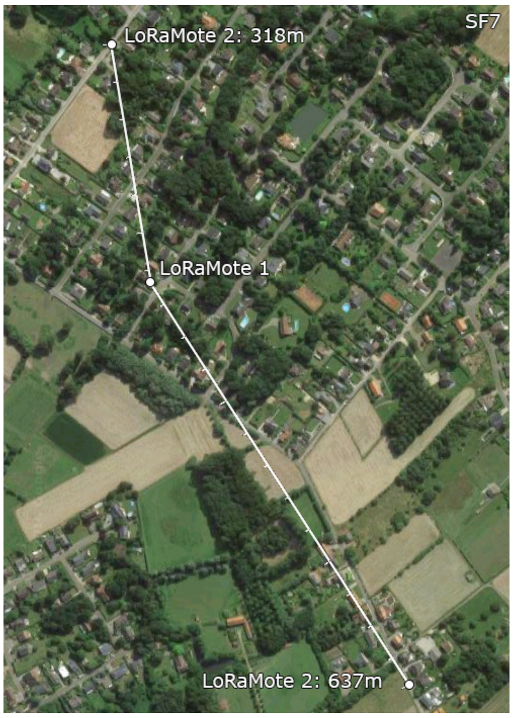


Figure 5.4: LoRa's maximal communication range using spreading factor 7 (Source: Google Maps imagery)

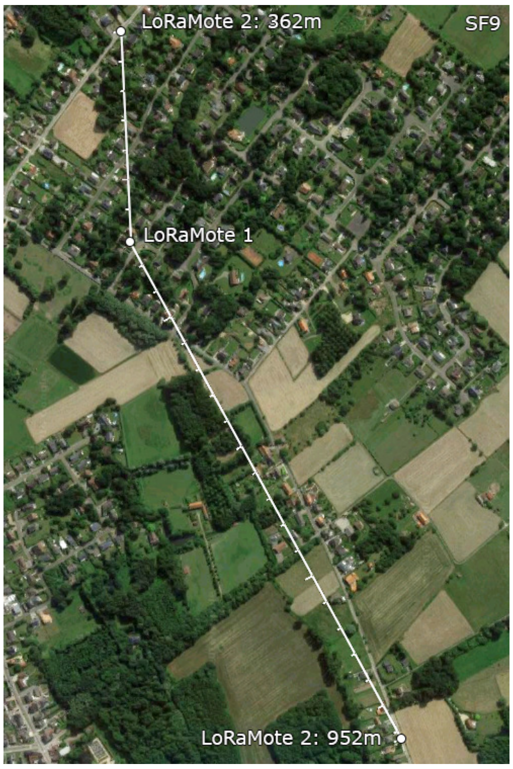


Figure 5.5: LoRa's maximal communication range using spreading factor 9 (Source: Google Maps imagery)

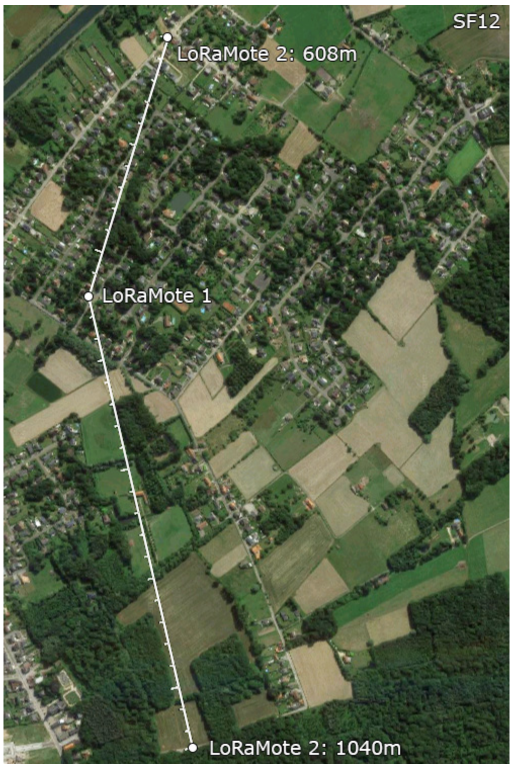


Figure 5.6: LoRa's maximal communication range using spreading factor 12 (Source: Google Maps imagery)

a greater impact when used in densely built areas then in more open areas. The maximal communication distance increased with around 68% between SF9 and SF12 in dense areas, whereas it 'only' increased with 9% in open areas. Note that these distances are measured between two LoRa end devices and not between an end device and a base station like in a classical LoRa infrastructure where base stations are located at strategical places.

5.2 IPv6 & RPL support

The Rime communication stack is a good tool for communication in wireless sensor networks. It is however not possible to communicate with other devices outside the network using Rime. Therefore it is a great added value when a platform is able to run the TCP/IP stack. This way it can communicate with external services which will extend the platform's possibilities enormously. Since Contiki is modularly built, it suffices to swap the Rime network stack with the 6LoWPAN stack.

The RPL-UDP example project that comes with Contiki was chosen to perform a network connectivity test. The project consists of two different applications: one for data concentrators (server) and one for the other nodes which send data to the data concentrator (client). Normally, a newly activated client waits for a data concentrator to announce itself by the means of a multi-cast beacon message. If after a while no beacon has been received, the client itself tries to contact a data concentrator. When a data concentrator is activated or detects a client request, it immediately sends a beacon. This beacon contains RPL routing information. So if a client receives the beacon, it knows where the data concentrator is located in the network topology and how to reach it. Once the routing table is updated, the client is able to send data to the data concentrator.

Since the LoRaMote devices were able to communicate using the Rime stack, the 6LoWPAN stack should work as well since all underlying layers remain the same. However, the RPL-UDP example which was deployed to the LoRaMotes did not work. After thoroughly investigating this problem, it was discovered that the server does receive the requests from the client but the client never receives a beacon from the server. Debugging the network revealed that the beacon is not discarded somewhere along the parsing mechanisms as thought as first, it just never arrives at the device physically. The sender beacon process was analyzed, but it turned out that everything went well and that the frames was transmitted correctly. A beacon that is being transmitted can even be noticed since it causes static noise in audio headphones.

If the LoRa radio settings are changed, it is however possible to receive data. More specifically, the CRC (Cyclic Redundancy Check) has to be disabled and frames need to get a fixed length. When these settings are enabled, the radio transceiver no longer discards these frames at hardware level. As a result, the client does receive the server's beacons but they are discarded in upper layers, which is a logical consequence of disabling the CRC mechanism since these frames were faulty. The issue thus lies somewhere between the framing and transmitting of server beacon frames. Since all these layers do function perfectly when serving the Rime stack, it is not immediately clear what the cause of this issue could be nor how it should be solved.

In this chapter, the results of the porting efforts of the two previous chapters were shown. Applications to test the board's peripherals, sensors and radio capabilities were deployed and proved that the Contiki operating system now successfully runs on the LoRaMote platform. One component does not work however. Despite the careful analysis of this issue with the 6LoWPAN network stack, no solution has been found and will require more investigation.

6 | Conclusion

In this Master's Thesis a new wireless sensor network technology called LoRa and its capabilities were studied. It is a radio modulation technique that trades off data rate in for communication range. At the moment, this technology is researched and deployed by both field experts and academia. LoRa technology provides a great new tool to create Low Power Wide Area Networks which are considered by the LoRa Alliance as single-hop star topologies only. The technology however allows employing it for other interesting concepts like long range multi-hop networks. Another disadvantage of the technology is its immaturity. It uses a proprietary system and thus not many applications are available whereas other embedded operating systems like Contiki have been around for quite some time and provide a great modular framework to develop applications for Internet of Things purposes.

Since both LoRa and Contiki have interesting traits, integrating both technologies can result in several advantages. By using Contiki as operating system, a great amount of existing Internet of Things applications will become available on LoRa devices. It will also increase Contiki's employability since only one operating system would be needed for short and long range applications. Moreover, it would also be a first step towards the cooperation of these short and long range networks. The fact that Zolertia already built a sensor node with a dual network stack consisting of 6LoWPAN and SigFox proofs that there is definitely interest in this concept.

As a result of these circumstances, the need to take this first step was tangibly present. Therefore the purpose of this Master's Thesis became creating a port of the Contiki operating system for the LoRaMote demo platform. Since no detailed or complete guide exists on how to port the Contiki operating system to new hardware platforms, the ambition of this project also included to create a unified document for the online Contiki community that covers all aspects, provides detailed explanations and gives clear examples on how to implement the necessary components.

The outcome of this last objective is presented in Chapter 4. In this chapter, Contiki's main structure is explained and the components that need porting are listed. For these components it is also discussed how to approach the implementation for new hardware platforms. Experiences from porting Contiki to the LoRaMote are included when generally applicable. Lastly, an explanation is given on how to build the whole project and upload it to the newly supported hardware platform. This way, any user with knowledge of the C language should be able to successfully create a Contiki port.

In order to demonstrate the efforts that went into taking the first step into integrating two technologies, a couple of tests were conducted in Chapter 5. First, the correct working of the LoRaMote's peripherals and sensors was verified by the means of a Contiki application. Using a different network-enabled Rime-driven Contiki application, the wireless communication capabilities of the LoRaMote were demonstrated and the communication range performance was examined. Another network experiment based on 6LoWPAN was attempted but did not succeed. There is still an unsolved issue regarding the framing or transmitting of frames. Since the whole system works perfectly when the modular network layer stack is swapped with Rime, the underlying cause is not clear and is as such still not found.

Although Contiki may be ported to the LoRaMote platform, there are still a lot of interesting projects that can be done. First of all, the previously mentioned issue with the 6LoWPAN network stack should be solved. From what was observed, the problem probably lies within the lower network layers. It is possible that the whole LoRa-6LoWPAN network stack will have to be investigated though. Another project could be the creation of a dual network stack sensor node like the Zolertia RE-Mote but with LoRa technology instead. Such development would greatly improve the integration of short range and long range wireless communication technologies. Lastly, since the LoRa technology is quite new, lots of its properties and possibilities still have to be discovered. The support for multi-hop networks with LoRa end devices is already in development, but other traits can be studied like its long-term energy profile, node-to-node performance in urban environments or integration possibilities with other long range technologies.

Bibliography

[1] J. Westö and D. Björklund, “An overview of enabling technologies for the Internet of Things”, 2014.

[2] Juniper Research, “’Internet of Things’ Connected Devices to Almost Triple to Over 38 Billion Units by 2020”, *Juniper Press Releases*, July 2015.

[3] D. Estrin, L. Girod, G. Pottie, and M. Srivastava, “Instrumenting the world with wireless sensor networks”, in *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP ’01). 2001 IEEE International Conference on*, vol. 4, 2001, pp. 2033–2036.

[4] E. H. Callaway Jr, *Wireless sensor networks: architectures and protocols*. CRC press, 2003.

[5] A. L. Colina, A. Vives, A. Bagula, M. Zennaro, and E. Pietrosemoli, *IoT in 5 Days*. E-Book, March 2015, rev 1.0.

[6] M. Chen, S. Mao, and Y. Liu, “Big data: A survey”, *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.

[7] Proximus, “Proximus first to launch commercial offer for "Internet of Things" via new LoRa-network”, *Proximus Press Releases*, October 2015.

[8] ENGIE, “ENGIE launches Belgium’s first Internet of Things network”, *ENGIE Press Releases*, June 2015.

[9] imec, “21st Wireless Community Workshop”, http://www.wirelesscommunity.be/work-meetings/21st-work-meeting-connecting-everything-to-the-internet-of-things/, 2016, [Wireless Community Workshop; attended April 27, 2016].

[10] iMinds, “Antwerp becomes Europe’s place-to-be for Internet of Things (IoT) experiments”, *iMinds News*, December 2015.

[11] H. M. A. Fahmy, *Wireless Sensor Networks: Concepts, Applications, Experimentation and Analysis*. Springer, 2016.

[12] G. Anastasi, M. Conti, M. Di Francesco, and A. Passarella, “Energy conservation in wireless sensor networks: A survey”, *Ad hoc networks*, vol. 7, no. 3, pp. 537–568, 2009.

[13] O. M. Al-Kofahi and A. E. Kamal, *Resilient Wireless Sensor Networks: The Case of Network Coding*. Springer, 2016.

[14] iMinds, "Investigating fundamental IoT breakthroughs", https://www.iminds.be/en/gain-insights/iot/contribution/strategic-research, 2016, [Online; accessed May 26, 2016].

[15] Institute of Electrical and Electronics Engineers (IEEE), "IEEE Standard for Low-Rate Wireless Personal Area Networks (WPANs)", *IEEE Std 802.15.4-2015*, pp. 1–709, April 2016.

[16] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", Internet Requests for Comments, RFC Editor, RFC 4944, September 2007.

[17] R. Zurawski, *Industrial communication technology handbook*. CRC Press, 2014.

[18] A. Phani, D. J. Kumar, and G. A. Kumar, "Operating systems for wireless sensor networks: A survey technical report", 2007.

[19] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*, "TinyOS: An operating system for sensor networks", in *Ambient intelligence*. Springer, 2005, pp. 115–148.

[20] A. Dwivedi, M. Tiwari, and O. Vyas, "Operating systems for tiny networked sensors: a survey", *International Journal of Recent Trends in Engineering*, vol. 1, no. 2, pp. 152–157, 2009.

[21] N. Riga, I. Matta, and A. Bestavros, "DIP: Density Inference Protocol for wireless sensor networks and its application to density-unbiased statistics", in *Proceedings of the Second International Workshop on Sensor and Actor Network Protocols and Applications (SANPA)*. Citeseer, 2004.

[22] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks", in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, 2004, pp. 95–107.

[23] I. Rhee, A. Warrier, M. Aia, J. Min, and M. L. Sichitiu, "Z-MAC: a hybrid MAC for wireless sensor networks", *IEEE/ACM Transactions on Networking (TON)*, vol. 16, no. 3, pp. 511–524, 2008.

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

[25] 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.

[26] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks", in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 2002, pp. 1567–1576.

[27] 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.

[28] H.-k. J. Chu, "Zero-copy TCP in Solaris", in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. Usenix Association, 1996, pp. 21–21.

[29] A. Dunkels, B. Grönvall, 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.

[30] A. Dunkels, "Contiki: Bringing IP to Sensor Networks", *ERCIM News*, no. 76, January 2009.

[31] A. Dunkels and O. Schmidt, "Protothreads - Lightweight Stackless Threads in C", SICS – Swedish Institute of Computer Science, Tech. Rep. T2005:05, March 2005.

[32] Contiki, "Contiki OS Wiki", https://github.com/contiki-os/contiki/wiki, 2016, [Online; accessed May 17, 2016].

[33] ISO and IEC, "ISO/IEC 7498-1: Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model", *International Standard ISO/IEC*, vol. 7498-1, p. 59, 1996.

[34] A. Dunkels, "The ContikiMAC Radio Duty Cycling Protocol", Swedish Institute of Computer Science, Tech. Rep. T2011:13, December 2011.

[35] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks", in *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 2006, pp. 307–320.

[36] R. Musăloiu-E, C.-J. M. Liang, and A. Terzis, "Koala: Ultra-low power data retrieval in wireless sensor networks", in *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on*. IEEE, 2008, pp. 421–432.

[37] Institute of Electrical and Electronics Engineers (IEEE), "IEEE Standard for Ethernet", *IEEE Std 802.3-2012*, December 2012.

[38] A. Dunkels, "Rime – A Lightweight Layered Communication Stack for Sensor Networks", in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, January 2007.

[39] Contiki, "The Rime communication stack", http://contiki.sourceforge.net/docs/2.6/a01798.html#_details, 2012, [Contiki Documentation; accessed May 31, 2016].

[40] A. Dunkels, "Full TCP/IP for 8 Bit Architectures", in *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*. usenix, May 2003.

[41] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks", Internet Requests for Comments, RFC Editor, RFC 6550, March 2012.

[42] M. Gamallo Gascón, "Analysis of Performance of RPL Routing Protocol for Different Radio Duty Cycling Protocols in Wireless Sensor Network Testbed", Bachelor's Thesis, Vrije Universiteit Brussel, 2016.

[43] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt, "Enabling Large-Scale Storage in Sensor Networks with the Coffee File System", in *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*, April 2009.

[44] 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.

[45] Semtech, "Introduction to LoRa", https://www.regonline.com/lora_webex_workshop, 2015, [Online Workshop; attended September 16, 2015].

[46] M. Bor, J. Vidler, and U. Roedig, "LoRa for the Internet of Things", in *EWSN '16 Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*. Junction Publishing, February 2016, pp. 361–366.

[47] LoRa Alliance, "LoRa Network Node Code", https://github.com/Lora-net/LoRaMac-node, 2016, [LoRa Device System; accessed May 17, 2016].

[48] LoRa Alliance, "Wide Area Networks for IoT", https://www.lora-alliance.org/, 2016, [Online; accessed June 3, 2016].

[49] Semtech, "Upcoming Events", http://www.semtech.com/events/, 2016, [Online; accessed June 3, 2016].

[50] LoRa Alliance, *LoRaWAN Specification*, January 2015.

[51] Semtech, *LoRaMote User Guide*, July 2014.

[52] cvskill.gary, "DFU File Manager: Unable to create image from this file", https://my.st.com/public/STe2ecommunities/mcu/Lists/STM32Discovery/DFUFileManagerUnabletocreateimagefromthisfile, 2014, [Online; accessed May 29, 2016].

[53] malroe, "CoIDE Toolchain", https://github.com/Lora-net/LoRaMac-node/issues/67, 2016, [Online; accessed May 17, 2016].

[54] Smarco10, "Bootloader Issue", https://github.com/Lora-net/LoRaMac-node/issues/53, 2016, [Online; accessed May 17, 2016].

[55] Contiki, "Contiki Operating System", http://www.contiki-os.org/, 2016, [Online; accessed May 16, 2016].

[56] Contiki, "About Contiki", https://web.archive.org/web/20110623121408/http://www.sics.se/contiki/about-contiki.html, 2011, [Online; accessed May 16, 2016].

[57] Contiki, "Contiki Mailing List", https://sourceforge.net/p/contiki/mailman/, 2016, [Online; accessed May 16, 2016].

[58] Y. Hershkovitz and D. Lev, "Porting Contiki to the LPC2148 Education Board", http://courses.cs.tau.ac.il/embedded/projects/fall2009/contiki_lpc2148/, 2009, [Course Final Project; accessed April 14, 2016].

[59] W. Balin, "Contiki OS port for ez430-RF2500", http://wincent.balin.at/ofdigitalmusic/wp-content/uploads/2010/04/Contiki-OS-port-to-ez430-RF2500.pdf, 2010, [MSP430 Design Contest Entry; accessed April 14, 2016].

[60] C. Schluep, "Porting Contiki to the BTnode System", http://www.cds.unibe.ch/teaching/hs09_seminar/seminar_hs09_schluep.pdf, 2009, [Seminar; accessed November 16, 2015].

[61] G. Oikonomou and I. Phillips, "Experiences from Porting the Contiki Operating System to a Popular Hardware Platform", in *Proc. 2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*, 2011, pp. 54–59.

[62] Z. Shen, D. O. Afolabi, H.-N. Liang, N. Zhang, D. Liu, K. L. Man, E. G. Lim, C.-U. Lei, Y. Yang, and L. Cheng, "Porting LooCI from the Contiki Platform to the Zigduino Platform: An Working Approach", *IAENG International Journal of Computer Science*, vol. 40, no. 2, pp. 104–109, 2013.

[63] A. Dunkels, "Porting Contiki - Crash Course", https://web.archive.org/web/20110623121533/http://www.sics.se/%7Eadam/contiki-workshop-2007/workshop07porting.ppt, 2007, [Workshop Presentation; accessed April 14, 2016].

[64] A. Stan, "Porting the Core of the Contiki operating system to the TelosB and MicaZ platforms", Bachelor's Thesis, University of Bremen, 2007.

[65] ST Microelectronics, "STM32L1XX Standard Peripherals Library", http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/LN1939/PF257913, 2015, [Driver Library; accessed April 14, 2016].

[66] Contiki, "Contiki OS Code", https://github.com/contiki-os/contiki, 2016, [Contiki Device System; accessed May 17, 2016].

[67] I. L. Crawford and K. R. Wadleigh, *Software Optimization for High Performance Computing*. Prentice Hall PTR, 2000.

[68] N. Murphy and M. Barr, "Watchdog Timers", *Embedded Systems Programming*, vol. 13, no. 12, pp. 112–124, 2000.

[69] ST Microelectronics, *STM32L1XX Reference Manual*, July 2015.

[70] P. Priyadarshan and B. R. Mundari, "Serial Communication using UART", Bachelor's Thesis, National Institute of Technology, Rourkela, 2008.

[71] Semtech, *SX1509 Data Sheet*, October 2009.

[72] Semtech, *SX9500 Data Sheet*, February 2014.

[73] IMST, *iM880A Data Sheet*, July 2013.

[74] J. Romkey, "Nonstandard for transmission of IP datagrams over serial lines: SLIP", Internet Requests for Comments, RFC Editor, STD 47, June 1988.

[75] ST Microelectronics, *STM32L1XX Data Sheet*, January 2015.

[76] ST Microelectronics, *ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32*, October 2012.

[77] D. Rath, *Open On-Chip Debugger*, May 2015.

[78] ARM, *RealView Compilation Tools Developer Guide*, December 2010.

[79] S. Schmidt and T. Volden, "dfu-util", http://dfu-util.sourceforge.net/, 2016, [Online; accessed May 17, 2016].

[80] M.-P. Uwase, N. T. Long, J. Tiberghien, K. Steenhaut, and J.-M. Dricot, *Poster Abstract: Outdoors Range Measurements with Zolertia Z1 Motes and Contiki*. Springer International Publishing, 2014, pp. 79–83.

[81] LoRa Alliance, "A technical overview of LoRa and LoRaWAN", White Paper, November 2015.

Appendix A
Porting Contiki to LoRaMote

A.1 Clock module

This clock module should be located at `/cpu/arm/stm32l1/clock.c`.

```c
#include <stdio.h>
#include "contiki.h"
#include "stm32l1xx_conf.h"
#include "stm32l1xx_systick.h"
/*---------------------------------------------------------------------------*/
/* After how many clock cycles should the systick interrupt be fired */
#define RELOAD_VALUE                    ((F_CPU/CLOCK_CONF_SECOND) - 1)
/*---------------------------------------------------------------------------*/
static volatile unsigned long seconds;
static volatile clock_time_t ticks;
/*---------------------------------------------------------------------------*/
/* This interrupt function will increase the tick counter */
void SysTick_Handler(void)
{
  ENERGEST_ON(ENERGEST_TYPE_IRQ);

  ticks++;
  if((ticks % CLOCK_SECOND) == 0) {
    seconds++;
    energest_flush();
  }

  /* If an etimer expired, continue its process */
  if(etimer_pending()) {
    etimer_request_poll();
  }

  ENERGEST_OFF(ENERGEST_TYPE_IRQ);
}
/*---------------------------------------------------------------------------*/
void clock_init(void)
{
  ticks = 0;
  seconds = 0;

  /* Select the main processor as systick clock source */
  SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);

  /* Set the reload value */
  SysTick_SetReload(RELOAD_VALUE);

  /* Enable the systick interrupt */
  SysTick_ITConfig(ENABLE);

  /* Enable the systick timer */
  SysTick_CounterCmd(SysTick_Counter_Enable);
}
/*---------------------------------------------------------------------------*/
unsigned long clock_seconds(void)
{
  return seconds;
}
/*---------------------------------------------------------------------------*/
void clock_set_seconds(unsigned long sec)
{
  seconds = sec;
}
/*---------------------------------------------------------------------------*/
clock_time_t clock_time(void)
{
  return ticks;
}
/*---------------------------------------------------------------------------*/
/* Busy-wait the CPU for a duration depending on CPU speed */
void clock_delay(unsigned int i)
{
  for(; i > 0; i--) {
    unsigned int j;
    for(j = 50; j > 0; j--) {
      __NOP();
    }
  }
}
/*---------------------------------------------------------------------------*/
/* Wait for a multiple of clock ticks (3.9ms per tick at 256Hz) */
void clock_wait(clock_time_t i)
{
  clock_time_t start;
  start = clock_time();
  while(clock_time() - start < i);
}
/*---------------------------------------------------------------------------*/
```

A.2 Rtimer

This rtimer module should be located at `/cpu/arm/stm32l1/rtimer-arch.c`.

```
#include "rtimer-arch.h"
/*---------------------------------------------------------------------------*/
/* The prescaler assumes the timer is sourced with same clock speed as the CPU */
#define RTIMER_PRESCALER                (F_CPU / (RTIMER_SECOND*2))
/*---------------------------------------------------------------------------*/
static volatile rtimer_clock_t rtimer_clock;
static volatile rtimer_clock_t timeout_value;
/*---------------------------------------------------------------------------*/
void TIM2_IRQHandler(void)
{
  ENERGEST_ON(ENERGEST_TYPE_IRQ);

  rtimer_clock++;

  /* Clear interrupt pending flag */
  TIM_ClearITPendingBit(TIM2, TIM_IT_Update);

  /* Check for, and run, any expired rtimers */
  if(rtimer_clock==timeout_value){
    rtimer_run_next();
  }

  ENERGEST_OFF(ENERGEST_TYPE_IRQ);
}
/*---------------------------------------------------------------------------*/
void rtimer_arch_init(void)
{
  rtimer_clock = 0;
  timeout_value = 0;

  /* To init, we use structures that are filled in and passed to the HAL */
  NVIC_InitTypeDef              NVIC_InitStructure;
  TIM_TimeBaseInitTypeDef       TIM_TimeBaseStructure;

  /* Enable TIM2 peripheral clock */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

  /* Enable the TIM2 Interrupt in the NVIC */
  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  /* Configure TIM2 */
  TIM_TimeBaseStructure.TIM_Period = 1;
  TIM_TimeBaseStructure.TIM_Prescaler = RTIMER_PRESCALER;
  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

  /* Enable TIM2 & TIM2 Update interrupt */
  TIM_ClearFlag(TIM2, TIM_FLAG_Update);
  TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
  TIM_Cmd(TIM2, ENABLE);
}
/*---------------------------------------------------------------------------*/
rtimer_clock_t rtimer_arch_now(void)
{
  return rtimer_clock;
}
/*---------------------------------------------------------------------------*/
void rtimer_arch_schedule(rtimer_clock_t t)
{
  timeout_value = rtimer_clock + t;
}
/*---------------------------------------------------------------------------*/
```

A.3 Watchdog

The watchdog implementation should be located at `/cpu/arm/stm32l1/watchdog.c`.

```
#include "dev/watchdog.h"
#include "stm32l1xx_conf.h"
#include "contiki-conf.h"
/*---------------------------------------------------------------------------*/
static uint8_t counterValue;
/*---------------------------------------------------------------------------*/
void watchdog_init(void)
{
#if WATCHDOG_USE_IWDG
  /* Get the LSI frequency: 37kHz according to stm32l1x datasheet,
     but can be measured with a high speed oscillator for greater precision */
  uint32_t LsiFreq = F_LSI;

  /* Enable write access to IWDG_PR and IWDG_RLR registers */
  IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);

  /* IWDG counter clock: LSI/32 */
  IWDG_SetPrescaler(IWDG_Prescaler_32);

  /* Set counter reload value to obtain 250ms IWDG TimeOut.
     (the timeout may varies due to LSI frequency dispersion)
     Counter Reload Value = IWDG counter clock period/250ms
              = (LSI/32) / 250ms
              = (LsiFreq/32) / 0.25s
              = LsiFreq / (32 * 4)
              = LsiFreq / 128
  */
  IWDG_SetReload(LsiFreq/(32 * (1000/WATCHDOG_IWDG_TIMEOUT)));
#else
  /* Enable WWDG clock */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);

  /* WWDG clock counter = (PCLK1 (32MHz)/4096)/8 = 977 Hz (~1024 us) */
  WWDG_SetPrescaler(WWDG_Prescaler_8);

  /* Set Window value to 127; WWDG counter should be refreshed only when the
    counter
     is below 127 (and greater than 64) otherwise a reset will be generated */
  WWDG_SetWindowValue(127);

  /* Set counter value to 127; WWDG timeout = ~1024 us * 64 = 65.53 ms
     In this case the refresh window is:
     ~1024us * (127-127) = 0 ms < refresh window < ~1024us * 64 = 65.53ms */
  counterValue = 127;
#endif
}
/*---------------------------------------------------------------------------*/
void watchdog_start(void)
{
#if ENABLE_WATCHDOG
  /* We setup the watchdog to reset the device after a specific time,
     unless watchdog_periodic() is called */
  #if WATCHDOG_USE_IWDG
    IWDG_ReloadCounter();
    IWDG_Enable();
  #else
    WWDG_Enable(counterValue);
  #endif
#endif
}
/*---------------------------------------------------------------------------*/
void watchdog_periodic(void)
{
  /* This function is called periodically to restart the watchdog timer */
#if WATCHDOG_USE_IWDG
  IWDG_ReloadCounter();
#else
  WWDG_SetCounter(counterValue);
#endif
}
/*---------------------------------------------------------------------------*/
void watchdog_stop(void)
{
  /* Impossible to stop watchdogs once started */
}
/*---------------------------------------------------------------------------*/
void watchdog_reboot(void)
{
  watchdog_stop();
  watchdog_init();
  watchdog_start();
}
/*---------------------------------------------------------------------------*/
```

A.4 LEDs

This LED module should be located at `/platform/loramote/dev/leds-arch.c`.

```
#include "dev/leds.h"
#include "lora-contiki-interface.h"
/*---------------------------------------------------------------------------*/
void leds_arch_init(void)
{
  /* Initialize LED structures */
  GpioInit(&Led1, LED_1, PIN_OUTPUT, PIN_PUSH_PULL, PIN_NO_PULL, 0);
  GpioInit(&Led2, LED_2, PIN_OUTPUT, PIN_PUSH_PULL, PIN_NO_PULL, 0);
  GpioInit(&Led3, LED_3, PIN_OUTPUT, PIN_PUSH_PULL, PIN_NO_PULL, 0);
}
/*---------------------------------------------------------------------------*/
unsigned char leds_arch_get(void)
{
  /* Create LED bitmask using bitwise OR'ing the seperate values */
  return (GpioRead(&Led1) ? LEDS_RED : 0)
        | (GpioRead(&Led2) ? LEDS_GREEN : 0)
        | (GpioRead(&Led3) ? LEDS_YELLOW : 0);
}
/*---------------------------------------------------------------------------*/
void leds_arch_set(unsigned char leds)
{
  /* Test the LED bitmask and set the LEDs accordingly */
  GpioWrite(&Led1, (leds & LEDS_RED) ? 0 : 1);
  GpioWrite(&Led2, (leds & LEDS_GREEN) ? 0 : 1);
  GpioWrite(&Led3, (leds & LEDS_YELLOW) ? 0 : 1);
}
/*---------------------------------------------------------------------------*/
```

A.5 Serial line driver

This I/O redirection module should be located at `/cpu/arm/stm32l1/syscalls.c`.

```
#include <stdio.h>
#include <errno.h>
#include "contiki.h"
/*---------------------------------------------------------------------------*/
/* Register name faking - works in collusion with the linker */
register char *stack_ptr asm ("sp");
extern int errno;
/*---------------------------------------------------------------------------*/
extern int __io_putstring(const unsigned char *buffer, int size);
extern int __io_putchar(int ch);
/*---------------------------------------------------------------------------*/
#if defined(REDIRECT_STDIO)
/*---------------------------------------------------------------------------*/
/* Write a character to a file. 'libc' subroutines will use this system routine
   for output to all files, including stdout. Returns number of bytes sent */
```

```
size_t _write(int handle, const unsigned char *buffer, size_t size)
{
  int data_idx;

  #ifdef REDIRECT_STDIO_STRINGMODE
    __io_putstring(buffer, size);
  #else
    for(data_idx = 0; data_idx < size; data_idx++) {
      __io_putchar(*buffer++);
    }
  #endif
  return size;
}
/*---------------------------------------------------------------------------*/
caddr_t _sbrk(int incr)
{
  extern char end;  // Defined by the linker
  static char *heap_end;
  char *prev_heap_end;

  if(heap_end == 0) {
    heap_end = &end;
  }
  prev_heap_end = heap_end;
  if(heap_end + incr > stack_ptr) {
    _write(1, "Heap and stack collision\n", 25);
    //abort();
    errno = ENOMEM;
    return (caddr_t)-1;
  }

  heap_end += incr;
  return (caddr_t)prev_heap_end;
}
/*---------------------------------------------------------------------------*/
#endif /* REDIRECT_STDIO */
/*---------------------------------------------------------------------------*/
```

This I/O redirection module should be located at /platform/loramote/dev/serial-line-arch.c.

```
#include "serial-line-arch.h"
/*---------------------------------------------------------------------------*/
/* Retarget stdio */
#ifdef __GNUC__
/* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
   set to 'Yes') calls __io_putchar() */
  #define PUTSTRING_PROTOTYPE int __io_putstring(const unsigned char *buffer, int
    size)
  #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
  #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE * f)
#endif /* __GNUC__ */
/*---------------------------------------------------------------------------*/
extern bool Virtual_ComPort_IsOpen(void);
/*---------------------------------------------------------------------------*/
void serial_line_arch_input_callback(UartNotifyId_t id){
  watchdog_periodic();
  if(id == UART_NOTIFY_RX){
    uint8_t ch = 0;
    while(UartGetChar(&UartUsb, &ch));
    serial_line_input_byte(ch);
  }
}
/*---------------------------------------------------------------------------*/
void serial_line_arch_init(void)
{
  UartUsb.IrqNotify = serial_line_arch_input_callback;
}
/*---------------------------------------------------------------------------*/
PUTCHAR_PROTOTYPE
{
  watchdog_periodic();
  /* Does not work during hardware interrupts */
  if(Virtual_ComPort_IsOpen()){
    while(UartPutChar(&UartUsb, ch));
  }
  return ch;
}
/*---------------------------------------------------------------------------*/
PUTSTRING_PROTOTYPE
{
  watchdog_periodic();
  if(Virtual_ComPort_IsOpen()){
    while(UartPutBuffer(&UartUsb, (uint8_t*)buffer, size));
  }
  return size;
}
/*---------------------------------------------------------------------------*/
void uart1_set_input(int (*input)(unsigned char c)){}
/*---------------------------------------------------------------------------*/
```

A.6 Button sensor

This button sensor implementation should be located at `/platform/loramote/dev/button-sensor.c`.

```
#include "lib/sensors.h"
#include "dev/button-sensor.h"
#include "lora-contiki-interface.h"
/*---------------------------------------------------------------------------*/
static int _initialized = 0;
static int _active = 0;
extern Gpio_t NIrqSx9500;
extern Gpio_t TxEnSx9500;
/*---------------------------------------------------------------------------*/
void cascaded_button_interrupt(void)
{
  ENERGEST_ON(ENERGEST_TYPE_IRQ);

  uint8_t statusSX1509 = 0;
  uint8_t statusSX9500 = 0;

  SX1509Read(RegInterruptSourceB, &statusSX1509);
  SX9500Read(SX9500_REG_IRQSRC, &statusSX9500);

  /* Check if interrupt is generated by SX1509 IOE8 pin & SX9500 close proximity
    sensor */
  if(((statusSX1509 & 0x01) == 0x01) && ((statusSX9500 & 0x40) == 0x40)){
    sensors_changed(&button_sensor);
  }

  /* Clear NINT interrupt */
  /* (NIRQ interrupt is automatically cleared by reading interrupt source register
    ) */
  SX1509Read(RegInterruptSourceB, &statusSX1509);
  SX1509Write(RegInterruptSourceB, (statusSX1509 & 0xFE));

  ENERGEST_OFF(ENERGEST_TYPE_IRQ);
}
/*---------------------------------------------------------------------------*/
static void init(void)
```

```
{
  uint8_t status = 0;

  /* Initialize the SX9500 proximity sensor */
  GpioInit(&NIrqSx9500, N_IRQ_SX9500, PIN_INPUT, PIN_PUSH_PULL, PIN_NO_PULL, 1);
  GpioInit(&TxEnSx9500, TX_EN_SX9500, PIN_OUTPUT, PIN_PUSH_PULL, PIN_NO_PULL, 1);

  SX9500Init();
  SX9500Write(SX9500_REG_IRQMSK, 0x10);
  SX9500Write(SX9500_REG_IRQSRC, 0x10);

  do {
    SX9500Read(SX9500_REG_IRQSRC, &status);
  }while((status & 0x10) == 0x00); /* While compensation for CS0 is pending */


  /* Enable SX9500 close proximity interrupt (NIRQ) */
  SX9500Write(SX9500_REG_IRQMSK, 0x40);
  SX9500Write(SX9500_REG_IRQSRC, 0x00);
  /* Increase proximity detection threshold */
  SX9500Write(SX9500_REG_PROXCTRL6, 0x1F);

  /* Enable the SX1509 IO Expander interrupt (NINT) */
  SX1509Write(RegInterruptMaskB, 0xFE);
  /* Set edge detection to falling */
  SX1509Write(RegSenseLowB, 0x02);
  SX1509Read(RegPullUpB, &status);
  SX1509Write(RegPullUpB, (status | 0x01));

  /* Initialize WKUP1/EXTI0 interrupt */
  Gpio_t IrqSX1509;
  GpioInit(&IrqSX1509, WKUP1, PIN_INPUT, PIN_PUSH_PULL, PIN_NO_PULL, 0);
  GpioSetInterrupt(&IrqSX1509, IRQ_FALLING_EDGE, IRQ_VERY_LOW_PRIORITY, &
    cascaded_button_interrupt);

  _initialized = 1;
  _active = 1;
}
/*---------------------------------------------------------------------------*/
static void activate(void)
{
  if(!_initialized){
    init();
  }
  _active = 1;
}
/*---------------------------------------------------------------------------*/
static void deactivate(void)
{
  _active = 0;
}
/*---------------------------------------------------------------------------*/
static int active(void)
{
  return _active;
}
/*---------------------------------------------------------------------------*/
static int value(int type)
{
  uint8_t regValue = 0;
  uint16_t offset = 0;

  /* Read 1st sensor offset */
  SX9500Read(SX9500_REG_OFFSETMSB, (uint8_t*)&regValue);
  offset = regValue << 8;
  SX9500Read(SX9500_REG_OFFSETLSB, (uint8_t*)&regValue);
  offset |= regValue;

  return (offset > 2000);
```

```c
}
/*---------------------------------------------------------------------------*/
static int configure(int type, int value)
{
  switch(type) {
  case SENSORS_HW_INIT:
    init();
    return 1;
  case SENSORS_ACTIVE:
    if(value) {
      activate();
    } else {
      deactivate();
    }
    return 1;
  }

  return 0;
}
/*---------------------------------------------------------------------------*/
static int status(int type)
{
  switch(type) {
  case SENSORS_READY:
    return active();
  }

  return 0;
}
/*---------------------------------------------------------------------------*/
SENSORS_SENSOR(button_sensor, BUTTON_SENSOR, value, configure, status);
/*---------------------------------------------------------------------------*/
```

A.7 Node MAC address & node ID

This node MAC and ID restorer functions should be located at `/cpu/arm/stm32l1/node-id.c`.

```c
#include <string.h>
#include "sys/node-id.h"
#include "contiki-conf.h"
/*---------------------------------------------------------------------------*/
#define DEVICE_ID_REG0          (*((volatile uint32_t *)0x1FF80050))
#define DEVICE_ID_REG1          (*((volatile uint32_t *)0x1FF80054))
#define DEVICE_ID_REG2          (*((volatile uint32_t *)0x1FF80064))
/*---------------------------------------------------------------------------*/
unsigned short node_id = 0;
unsigned char node_mac[8];
volatile uint32_t device_id[3];
/*---------------------------------------------------------------------------*/
void node_id_restore(void)
{
  device_id[0] = DEVICE_ID_REG0;
  device_id[1] = DEVICE_ID_REG1;
  device_id[2] = DEVICE_ID_REG2;

  (*(uint32_t *)node_mac) = DEVICE_ID_REG1;
  (*(((uint32_t *)node_mac) + 1)) = DEVICE_ID_REG2 + DEVICE_ID_REG0;
  node_id = (unsigned short)DEVICE_ID_REG2;
}
/*---------------------------------------------------------------------------*/
void node_id_burn(unsigned short id){}
/*---------------------------------------------------------------------------*/
```

A.8 SLIP driver

The SLIP driver should be placed at `/platform/loramote/dev/slip-arch.c`.

```c
#include "dev/slip.h"
#include "lora-contiki-interface.h"
/*---------------------------------------------------------------------------*/
extern bool Virtual_ComPort_IsOpen(void);
/*---------------------------------------------------------------------------*/
void slip_arch_input_callback(UartNotifyId_t id){
  watchdog_periodic();
  if(id == UART_NOTIFY_RX){
    uint8_t ch = 0;
    while(UartGetChar(&UartUsb, &ch));
    slip_input_byte(ch);
  }
}
/*---------------------------------------------------------------------------*/
void slip_arch_init(unsigned long ubr)
{
  UartUsb.IrqNotify = slip_arch_input_callback;
}
/*---------------------------------------------------------------------------*/
void slip_arch_writeb(unsigned char ch)
{
  watchdog_periodic();
  if(Virtual_ComPort_IsOpen()){
    while(UartPutChar(&UartUsb, ch));
  }
}
/*---------------------------------------------------------------------------*/
```

A.9 Radio driver

The radio driver should be placed at `/platform/loramote/net/lora-radio-arch.c`. Related files like radio configurations also should reside in this directory.

```c
#include "lora-radio-arch.h"
/*---------------------------------------------------------------------------*/
#define DEBUG 0
#if DEBUG
  #define PRINTF(...)   printf(__VA_ARGS__)
#else
  #define PRINTF(...)
#endif
/*---------------------------------------------------------------------------*/
#define CLEAR_RXBUF()           (lora_radio_rxbuf[0] = 0)
#define IS_RXBUF_EMPTY()        (lora_radio_rxbuf[0] == 0)
/*---------------------------------------------------------------------------*/
/* Incoming data buffer, the first byte will contain the length of the packet */
static uint8_t lora_radio_rxbuf[LORA_MAX_PAYLOAD_SIZE + 1];
static RadioEvents_t RadioEvents;
static int packet_is_prepared = 0;
static const void *packet_payload;
static unsigned short packet_payload_len = 0;
static packetbuf_attr_t last_rssi = 0;
/*---------------------------------------------------------------------------*/
static int lora_radio_init(void);
static int lora_radio_prepare(const void *payload, unsigned short payload_len);
static int lora_radio_transmit(unsigned short payload_len);
static int lora_radio_send(const void *data, unsigned short len);
static int lora_radio_read(void *buf, unsigned short bufsize);
static int lora_radio_channel_clear(void);
static int lora_radio_receiving_packet(void);
static int lora_radio_pending_packet(void);
static int lora_radio_on(void);
static int lora_radio_off(void);
void OnTxDone(void);
void OnRxDone(uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr);
void OnTxTimeout(void);
void OnRxTimeout(void);
void OnRxError(void);
/*---------------------------------------------------------------------------*/
PROCESS(lora_radio_process, "LoRa radio driver process");
/*---------------------------------------------------------------------------*/
const struct radio_driver lora_radio_driver =
{
  lora_radio_init,
  lora_radio_prepare,
  lora_radio_transmit,
  lora_radio_send,
  lora_radio_read,
  lora_radio_channel_clear,
  lora_radio_receiving_packet,
  lora_radio_pending_packet,
  lora_radio_on,
  lora_radio_off,
};
/*---------------------------------------------------------------------------*/
static int lora_radio_init(void)
{
  PRINTF("\nRADIO INIT IN\n");

  SpiInit(&SX1272.Spi, RADIO_MOSI, RADIO_MISO, RADIO_SCLK, NC);
  SX1272IoInit();

  /* Radio initialization */
  RadioEvents.TxDone = OnTxDone;
  RadioEvents.RxDone = OnRxDone;
  RadioEvents.TxTimeout = OnTxTimeout;
  RadioEvents.RxTimeout = OnRxTimeout;
  RadioEvents.RxError = OnRxError;

  Radio.Init(&RadioEvents);
  Radio.SetChannel(RF_FREQUENCY);

#if defined(USE_MODEM_LORA)
  Radio.SetTxConfig(MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
                    LORA_SPREADING_FACTOR, LORA_CODINGRATE,
                    LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON, LORA_CRC_ON,
                    LORA_FREQUENCY_HOPPING_ON, LORA_HOPPING_PERIOD,
                    LORA_IQ_INVERSION_ON, TX_TIMEOUT_VALUE);
  Radio.SetRxConfig(MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR,
                    LORA_CODINGRATE, 0, LORA_PREAMBLE_LENGTH,
                    LORA_SYMBOL_TIMEOUT, LORA_FIX_LENGTH_PAYLOAD_ON,
                    LORA_FIXED_PAYLOAD_LENGTH, LORA_CRC_ON,
                    LORA_FREQUENCY_HOPPING_ON, LORA_HOPPING_PERIOD,
                    LORA_IQ_INVERSION_ON, RX_CONTINUOUS_MODE);
#elif defined(USE_MODEM_FSK)
  Radio.SetTxConfig(MODEM_FSK, TX_OUTPUT_POWER, FSK_FDEV, 0,
                    FSK_DATARATE, 0,
                    FSK_PREAMBLE_LENGTH, FSK_FIX_LENGTH_PAYLOAD_ON,
                    FSK_CRC_ON, 0, 0, 0, TX_TIMEOUT_VALUE);
  Radio.SetRxConfig(MODEM_FSK, FSK_BANDWIDTH, FSK_DATARATE,
                    0, FSK_AFC_BANDWIDTH, FSK_PREAMBLE_LENGTH,
                    0, FSK_FIX_LENGTH_PAYLOAD_ON, FSK_FIXED_PAYLOAD_LENGTH,
                    FSK_CRC_ON, 0, 0, 0, RX_CONTINUOUS_MODE);
#else
```

```
  #error "Please define a modem in the compiler options."
#endif

  process_start(&lora_radio_process, NULL);

  PRINTF("RADIO INIT OUT\n");
  return 0;
}
/*---------------------------------------------------------------------------*/
static int lora_radio_prepare(const void *payload, unsigned short payload_len)
{
  PRINTF("PREPARE IN: %u bytes\n", payload_len);
  packet_is_prepared = 0;

  /* Checks if the payload length is supported */
  if(payload_len > LORA_MAX_PAYLOAD_SIZE) {
    return RADIO_TX_ERR;
  }

  packet_payload = payload;
  packet_payload_len = payload_len;
  packet_is_prepared = 1;

  PRINTF("PREPARE OUT\n");
  return RADIO_TX_OK;
}
/*---------------------------------------------------------------------------*/
static int lora_radio_transmit(unsigned short payload_len)
{
  PRINTF("TRANSMIT IN\n");

  if(!packet_is_prepared) {
    return RADIO_TX_ERR;
  }

  Radio.Send((uint8_t *)packet_payload, packet_payload_len);
  packet_is_prepared = 0;

  PRINTF("TRANSMIT OUT\n");
  return RADIO_TX_OK;
}
/*---------------------------------------------------------------------------*/
static int lora_radio_send(const void *payload, unsigned short payload_len)
{
  if(lora_radio_prepare(payload, payload_len) == RADIO_TX_ERR) {
    return RADIO_TX_ERR;
  }
  return lora_radio_transmit(payload_len);
}
/*---------------------------------------------------------------------------*/
static int lora_radio_read(void *buf, unsigned short bufsize)
{
  PRINTF("READ IN\n");

  /* Checks if the RX buffer is empty */
  if(IS_RXBUF_EMPTY()) {
    PRINTF("READ OUT: RX BUFFER EMPTY\n");
    return 0;
  }

  /* Checks if buffer has the correct size */
  if(bufsize < lora_radio_rxbuf[0]) {
    PRINTF("READ OUT: TOO SMALL BUFFER\n");
    return 0;
  }

  /* Copies the packet received */
  memcpy(buf, lora_radio_rxbuf+1, lora_radio_rxbuf[0]);
  packetbuf_set_attr(PACKETBUF_ATTR_RSSI, last_rssi);
  bufsize = lora_radio_rxbuf[0];
  CLEAR_RXBUF();

  PRINTF("READ OUT\n");
  return bufsize;
}
/*---------------------------------------------------------------------------*/
static int lora_radio_channel_clear(void)
{
  PRINTF("CHANNEL CLEAR IN\n");
  bool channel_clear;

#if defined(USE_MODEM_LORA)
  channel_clear = Radio.IsChannelFree(MODEM_LORA, RF_FREQUENCY, CCA_THRESHOLD);
#elif defined(USE_MODEM_FSK)
  channel_clear = Radio.IsChannelFree(MODEM_FSK, RF_FREQUENCY, CCA_THRESHOLD);
#else
  #error "Please define a modem in the compiler options."
#endif

  PRINTF("CHANNEL CLEAR OUT\n");
  return channel_clear;
}
/*---------------------------------------------------------------------------*/
static int lora_radio_receiving_packet(void)
{
  return 0;
}
/*---------------------------------------------------------------------------*/
static int lora_radio_pending_packet(void)
{
  PRINTF("PENDING PACKET\n");
  return !IS_RXBUF_EMPTY();
}
/*---------------------------------------------------------------------------*/
static int lora_radio_off(void)
{
  Radio.Sleep();
  PRINTF("RADIO OFF\n");
  return 0;
}
/*---------------------------------------------------------------------------*/
static int lora_radio_on(void)
{
  Radio.Rx(RX_TIMEOUT_VALUE);
  PRINTF("RADIO ON\n");
  return 0;
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(lora_radio_process, ev, data)
{
  PROCESS_BEGIN();

  PRINTF("LoRa radio: process started\n");
  int len;

  while(1) {
    PROCESS_YIELD_UNTIL(ev == PROCESS_EVENT_POLL);

    PRINTF("LoRa radio: polled\n");
    packetbuf_clear();
    len = lora_radio_read(packetbuf_dataptr(), PACKETBUF_SIZE);

    if(len > 0) {
      packetbuf_set_datalen(len);
      NETSTACK_RDC.input();
    }

    if(!IS_RXBUF_EMPTY()) {
      process_poll(&lora_radio_process);
    }
  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
void OnTxDone(void)
{
  PRINTF("PACKET SENT\n");
  Radio.Rx(RX_TIMEOUT_VALUE);
}
/*---------------------------------------------------------------------------*/
void OnRxDone(uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr)
{
  memcpy(lora_radio_rxbuf+1, payload, size);
  lora_radio_rxbuf[0] = size;
  PRINTF("PACKET RECEIVED\n");
  process_poll(&lora_radio_process);
  last_rssi = (packetbuf_attr_t)rssi;
}
/*---------------------------------------------------------------------------*/
void OnTxTimeout(void)
{
  PRINTF("TX TIMEOUT\n");
  Radio.Rx(RX_TIMEOUT_VALUE);
}
/*---------------------------------------------------------------------------*/
void OnRxTimeout(void)
{
  PRINTF("RX TIMEOUT\n");
  Radio.Rx(RX_TIMEOUT_VALUE);
}
/*---------------------------------------------------------------------------*/
void OnRxError(void)
{
  PRINTF("RX ERROR\n");
  Radio.Rx(RX_TIMEOUT_VALUE);
}
/*---------------------------------------------------------------------------*/
```

A.10 External clock

This external clock implementation should be located at `/cpu/arm/stm32l1/rtc-arch.c`.

```
#include "rtc-arch.h"
/*---------------------------------------------------------------------------*/
/* The prescaler assumes the RTC timer is sourced with the LSE
   and uses the minimum asynchronous division factor (2) */
#define RTC_PRESCALER                   ((F_LSE/2) / RTIMER_SECOND)
/*---------------------------------------------------------------------------*/
static int rtcInitialized = 0;
/*---------------------------------------------------------------------------*/
void init_rtc(void)
{
  if(rtcInitialized){
    return;
  }

  /* Initialize the RTC */
  RTC_InitTypeDef RTC_InitStructure;
  RTC_TimeTypeDef RTC_TimeStruct;
  RTC_DateTypeDef RTC_DateStruct;

  /* Enable the PWR clock */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);

  /* Allow access to RTC */
  PWR_RTCAccessCmd(ENABLE);

  /* Reset RTC Domain */
  RCC_RTCResetCmd(ENABLE);
  RCC_RTCResetCmd(DISABLE);

  /* Enable the LSE OSC */
  RCC_LSEConfig(RCC_LSE_ON);

  /* Wait till LSE is ready */
  while(RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET){}

  /* Select the RTC Clock Source */
  RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);

  /* Enable the RTC Clock */
  RCC_RTCCLKCmd(ENABLE);
  RTC_TimeStructInit(&RTC_TimeStruct);
  RTC_DateStructInit(&RTC_DateStruct);
  RTC_SetTime(RTC_Format_BIN, &RTC_TimeStruct);
  RTC_SetDate(RTC_Format_BIN, &RTC_DateStruct);

  /* Wait for RTC APB registers synchronisation */
  RTC_WaitForSynchro();

  /* Configure the RTC data register and RTC prescaler */
  RTC_InitStructure.RTC_AsynchPrediv = 0x01;
  RTC_InitStructure.RTC_SynchPrediv  = RTC_PRESCALER-1;
  RTC_InitStructure.RTC_HourFormat   = RTC_HourFormat_24;
  RTC_Init(&RTC_InitStructure);

  /* Wait for RTC APB registers synchronisation */
  RTC_WaitForSynchro();

  rtcInitialized = 1;
}
/*---------------------------------------------------------------------------*/
```

A.11 Low power controller

The low power control functions should be located at `/cpu/arm/stm32l1/lpm-arch.c`.

```
#include "lpm-arch.h"
/*---------------------------------------------------------------------------*/
static int stopModeActivated = 0;
/*---------------------------------------------------------------------------*/
void lpm_enter_stopmode(void)
{
  stopModeActivated = 1;

  /* Disable the Power Voltage Detector */
  PWR_PVDCmd(DISABLE);

  /* Set MCU in ULP (Ultra Low Power) */
  PWR_UltraLowPowerCmd(ENABLE);

  /* Disable fast wakeUp */
  PWR_FastWakeUpCmd(DISABLE);

  /* Enter Stop Mode */
  PWR_EnterSTOPMode(PWR_Regulator_LowPower, PWR_STOPEntry_WFI);
}
/*---------------------------------------------------------------------------*/
void lpm_exit_stopmode(void)
{
  if(!stopModeActivated){
    return;
  }

  /* Disable IRQ while the MCU is not running on HSE */
  __disable_irq();

  /* Enable HSE */
  RCC_HSEConfig(RCC_HSE_ON);

  /* Wait till HSE is ready */
  while (RCC_GetFlagStatus(RCC_FLAG_HSERDY) == RESET){}

  /* Enable PLL */
  RCC_PLLCmd(ENABLE);

  /* Wait till PLL is ready */
  while (RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET){}

  /* Select PLL as system clock source */
  RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);

  /* Wait till PLL is used as system clock source */
  while (RCC_GetSYSCLKSource() != 0x0C){}

  /* Set MCU in ULP (Ultra Low Power) */
  PWR_UltraLowPowerCmd(DISABLE); // add up to 3ms wakeup time

  /* Enable the Power Voltage Detector */
  PWR_PVDCmd(ENABLE);

  __enable_irq();

  stopModeActivated = 0;
}
/*---------------------------------------------------------------------------*/
```

A.12 Low power clock module

The low power clock module replaces the `/cpu/arm/stm32l1/clock.c` file.

```
#include "contiki.h"
#include "stm32l1xx_conf.h"
#include "rtc-arch.h"
#include "lpm-arch.h"
/*---------------------------------------------------------------------------*/
#define DEBUG 0
#if DEBUG
  #define PRINTF(...)   printf(__VA_ARGS__)
#else
  #define PRINTF(...)
#endif
/*---------------------------------------------------------------------------*/
/* The counter assumes the timer is sourced with the LSE and uses Div16 */
#define RTC_WKUPCOUNTER                 ((F_LSE/16) / CLOCK_SECOND)
/*---------------------------------------------------------------------------*/
static volatile unsigned long seconds;
```

```c
static volatile clock_time_t ticks;
/*---------------------------------------------------------------------------*/
void RTC_WKUP_IRQHandler(void)
{
  ENERGEST_ON(ENERGEST_TYPE_IRQ);

  /* Check on the WakeUp flag */
  if(RTC_GetITStatus(RTC_IT_WUT) != RESET)
  {
    ticks++;
    if((ticks % CLOCK_SECOND) == 0){
      seconds++;
      energest_flush();
      PRINTF("second %i (%i ticks)\n", seconds, ticks);
    }

    /* If an etimer expired, continue its process */
    if(etimer_pending()){
      lpm_exit_stopmode();
      etimer_request_poll();
    }

    /* Clear RTC WakeUp flags */
    RTC_ClearITPendingBit(RTC_IT_WUT);
  }

  /* Clear the EXTI line 20 */
  EXTI_ClearITPendingBit(EXTI_Line20);

  ENERGEST_OFF(ENERGEST_TYPE_IRQ);
}
/*---------------------------------------------------------------------------*/
void clock_init(void)
{
  seconds = 0;
  ticks = 0;

  /* Initialize the RTC clock */
  init_rtc();

  /* Initialize the RTC WakeUp interrupt */
  EXTI_InitTypeDef EXTI_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;

  /* EXTI configuration */
  EXTI_ClearITPendingBit(EXTI_Line20);
  EXTI_InitStructure.EXTI_Line = EXTI_Line20;
  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStructure);

  /* Enable the RTC Wakeup Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = RTC_WKUP_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  /* RTC Wakeup Interrupt Generation: Clock Source: RTCDiv_16, Wakeup Time Base:
     7.8ms at 128Hz */
  RTC_WakeUpClockConfig(RTC_WakeUpClock_RTCCLK_Div16);
  RTC_SetWakeUpCounter(RTC_WKUPCOUNTER-1);

  /* Enable the Wakeup Interrupt */
  RTC_ITConfig(RTC_IT_WUT, ENABLE);

  /* Enable Wakeup Counter */
  RTC_WakeUpCmd(ENABLE);
}
/*---------------------------------------------------------------------------*/
unsigned long clock_seconds(void)
{
  return seconds;
}
/*---------------------------------------------------------------------------*/
void clock_set_seconds(unsigned long sec)
{
  seconds = sec;
}
/*---------------------------------------------------------------------------*/
clock_time_t clock_time(void)
{
  return ticks;
}
/*---------------------------------------------------------------------------*/
/* Busy-wait the CPU for a duration depending on CPU speed */
void clock_delay(unsigned int i)
{
  for(; i > 0; i--) {
    unsigned int j;
    for(j = 50; j > 0; j--) {
      __NOP();
    }
  }
}
/*---------------------------------------------------------------------------*/
/* Wait for a multiple of clock ticks (7.8ms per tick at 128Hz) */
void clock_wait(clock_time_t i)
{
  clock_time_t start;
  start = clock_time();
  while(clock_time() - start < i);
}
/*---------------------------------------------------------------------------*/
```

A.13 Low power rtimer

The low power rtimer replaces the /cpu/arm/stm32l1/rtimer-arch.c file.

```
#include "rtimer-arch.h"
/*---------------------------------------------------------------------------*/
#define MCU_WAKE_UP_TIME_TICKS          (MCU_WAKE_UP_TIME / (1000000/RTIMER_SECOND
    ))
/*---------------------------------------------------------------------------*/
static const uint8_t SecondsInMinute = 60;
static const uint16_t SecondsInHour = 3600;
static const uint32_t SecondsInDay = 86400;
static const uint8_t HoursInDay = 24;
static const uint16_t DaysInYear = 365;
static const uint16_t DaysInLeapYear = 366;
static const double DaysInCentury = 36524.219;
static const uint8_t DaysInMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
    31};
static const uint8_t DaysInMonthLeapYear[] = {31, 29, 31, 30, 31, 30, 31, 31, 30,
    31, 30, 31};
static uint8_t PreviousYear = 0;
static uint8_t Century = 0;
/*---------------------------------------------------------------------------*/
void RTC_Alarm_IRQHandler(void)
{
  ENERGEST_ON(ENERGEST_TYPE_IRQ);

  /* Check on the AlarmA flag */
  if(RTC_GetITStatus(RTC_IT_ALRA) != RESET)
  {
    lpm_exit_stopmode();
    rtimer_run_next();

    /* Clear RTC AlarmA Flags */
    RTC_ClearITPendingBit(RTC_IT_ALRA);
  }

  /* Clear the EXTI line 17 */
  EXTI_ClearITPendingBit(EXTI_Line17);

  ENERGEST_OFF(ENERGEST_TYPE_IRQ);
}
/*---------------------------------------------------------------------------*/
void rtimer_arch_init(void)
{
  /* Initialize the RTC clock */
  init_rtc();

  /* Initialize the RTC Alarm interrupt */
  EXTI_InitTypeDef EXTI_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;

  /* EXTI configuration */
  EXTI_ClearITPendingBit(EXTI_Line17);
  EXTI_InitStructure.EXTI_Line = EXTI_Line17;
  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStructure);

  /* Enable the RTC Alarm Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = RTC_Alarm_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  /* Disable AlarmA interrupt */
  RTC_ITConfig(RTC_IT_ALRA, ENABLE);

  /* Disable the AlarmA */
  RTC_AlarmCmd(RTC_Alarm_A, DISABLE);
}
/*---------------------------------------------------------------------------*/
rtimer_clock_t rtimer_arch_now(void)
{
  rtimer_clock_t calendarValue = 0;
  uint8_t i = 0;

  RTC_TimeTypeDef RTC_TimeStruct;
  RTC_DateTypeDef RTC_DateStruct;
  RTC_GetTime(RTC_Format_BIN, &RTC_TimeStruct);
  RTC_GetDate(RTC_Format_BIN, &RTC_DateStruct);
  RTC_WaitForSynchro();

  if((PreviousYear == 99) && (RTC_DateStruct.RTC_Year == 0)){
    Century++;
  }
  PreviousYear = RTC_DateStruct.RTC_Year;

  /* Centuries */
  for(i = 0; i < Century; i++){
    calendarValue += (rtimer_clock_t)(DaysInCentury * SecondsInDay);
  }

  /* Years */
  for(i = 0; i < RTC_DateStruct.RTC_Year; i++){
    if((i == 0) || (i % 4 == 0)){
      calendarValue += DaysInLeapYear * SecondsInDay;
    }else{
      calendarValue += DaysInYear * SecondsInDay;
    }
  }

  /* Months */
  if((RTC_DateStruct.RTC_Year == 0) || (RTC_DateStruct.RTC_Year % 4 == 0)){
    for(i = 0; i < (RTC_DateStruct.RTC_Month - 1); i++){
      calendarValue += DaysInMonthLeapYear[i] * SecondsInDay;
    }
  }else{
    for(i = 0;  i < (RTC_DateStruct.RTC_Month - 1); i++){
      calendarValue += DaysInMonth[i] * SecondsInDay;
    }
  }

  /* Days */
  calendarValue += ((uint32_t)RTC_TimeStruct.RTC_Seconds +
            ((uint32_t)RTC_TimeStruct.RTC_Minutes * SecondsInMinute) +
            ((uint32_t)RTC_TimeStruct.RTC_Hours * SecondsInHour) +
            ((uint32_t)(RTC_DateStruct.RTC_Date * SecondsInDay)));

  return calendarValue;
}
/*---------------------------------------------------------------------------*/
void rtimer_arch_schedule(rtimer_clock_t wakeup_time)
{
  uint16_t rtcSeconds = 0;
  uint16_t rtcMinutes = 0;
  uint16_t rtcHours = 0;
  uint16_t rtcDays = 0;

  uint8_t rtcAlarmSeconds = 0;
  uint8_t rtcAlarmMinutes = 0;
  uint8_t rtcAlarmHours = 0;
  uint16_t rtcAlarmDays = 0;

  RTC_AlarmTypeDef RTC_AlarmStructure;
  RTC_TimeTypeDef RTC_TimeStruct;
  RTC_DateTypeDef RTC_DateStruct;

  /* Clear Previous Alarm */
  RTC_ClearFlag(RTC_FLAG_ALRAF);
  RTC_AlarmCmd(RTC_Alarm_A, DISABLE);


  RTC_GetTime(RTC_Format_BIN, &RTC_TimeStruct);
  RTC_GetDate(RTC_Format_BIN, &RTC_DateStruct);

  wakeup_time = wakeup_time - floor(MCU_WAKE_UP_TIME_TICKS + 0.5);              //
     Round ticks

  rtcSeconds = (wakeup_time % SecondsInMinute) + RTC_TimeStruct.RTC_Seconds;
  rtcMinutes = ((wakeup_time/SecondsInMinute) % SecondsInMinute) + RTC_TimeStruct.
    RTC_Minutes;
  rtcHours = ((wakeup_time/SecondsInHour) % HoursInDay) + RTC_TimeStruct.RTC_Hours
    ;
  rtcDays = (wakeup_time/SecondsInDay) + RTC_DateStruct.RTC_Date;

  rtcAlarmSeconds = (rtcSeconds) % SecondsInMinute;
  rtcAlarmMinutes = ((rtcSeconds/SecondsInMinute) + rtcMinutes) % SecondsInMinute;
  rtcAlarmHours   = ((((rtcSeconds/SecondsInMinute) + rtcMinutes) /
    SecondsInMinute) + rtcHours) % HoursInDay;
  rtcAlarmDays  = (((((rtcSeconds/SecondsInMinute) + rtcMinutes) / SecondsInMinute
    ) + rtcHours) / HoursInDay) + rtcDays;

  if((RTC_DateStruct.RTC_Year == 0) || (RTC_DateStruct.RTC_Year % 4 == 0))
  {
    if(rtcAlarmDays > DaysInMonthLeapYear[RTC_DateStruct.RTC_Month-1])
    {
      rtcAlarmDays = rtcAlarmDays % DaysInMonthLeapYear[RTC_DateStruct.RTC_Month
    -1];
    }
  }
  else
  {
    if(rtcAlarmDays > DaysInMonth[RTC_DateStruct.RTC_Month-1])
    {
      rtcAlarmDays = rtcAlarmDays % DaysInMonth[RTC_DateStruct.RTC_Month-1];
    }
  }

  RTC_AlarmStructure.RTC_AlarmTime.RTC_Seconds = rtcAlarmSeconds;
  RTC_AlarmStructure.RTC_AlarmTime.RTC_Minutes = rtcAlarmMinutes;
  RTC_AlarmStructure.RTC_AlarmTime.RTC_Hours   = rtcAlarmHours;
  RTC_AlarmStructure.RTC_AlarmDateWeekDay      = (uint8_t)rtcAlarmDays;
  RTC_AlarmStructure.RTC_AlarmDateWeekDaySel   = RTC_AlarmDateWeekDaySel_Date;
  RTC_AlarmStructure.RTC_AlarmMask             = RTC_AlarmMask_None;

  /* Enable the AlarmA */
  RTC_SetAlarm(RTC_Format_BIN, RTC_Alarm_A, &RTC_AlarmStructure);
  RTC_AlarmCmd(RTC_Alarm_A, ENABLE);
}
/*---------------------------------------------------------------------------*/
```

A.14 Contiki main

This main function file should be located at `/platform/loramote/contiki-loramote-main.c`.

```
#include "lora-contiki-interface.h"
/*---------------------------------------------------------------------------*/
SENSORS(&button_sensor,
        &radio_sensor,
        &temperature_sensor,
        &altitude_sensor,
        &pressure_sensor,
        &battery_sensor);
/*---------------------------------------------------------------------------*/
extern unsigned char node_mac[8];
static linkaddr_t rime_addr;
static uip_ipaddr_t ipaddr;
/*---------------------------------------------------------------------------*/
static void print_processes(struct process * const processes[]);
static void print_device_config(void);
static void set_rime_addr(void);
extern bool Virtual_ComPort_IsOpen(void);
/*---------------------------------------------------------------------------*/
int main(int argc, char **argv)
{
  /* Initialize hardware */
  BoardInitMcu_Contiki();
  leds_init();
  rtimer_init();
  serial_line_arch_init();
  printf("\nInitializing hardware... Done!\n");

  /* Initialize Contiki */
  printf("Initializing Contiki... "); fflush(stdout);
  clock_init();
  watchdog_init();
  process_init();
  process_start(&etimer_process, NULL);
  ctimer_init();
  serial_line_init();
  process_start(&sensors_process, NULL);
  printf("Done!\n");

  /* Initialize networking */
  printf("Initializing network... "); fflush(stdout);
  /* Restore node id if such has been stored in external mem */
#ifdef NODEID
  node_id = NODEID;
#else/* NODE_ID */
  node_id_restore(); /* also configures node_mac[] */
#endif /* NODE_ID */
  set_rime_addr();
  random_init(node_id);
  netstack_init();
#if UIP_CONF_IPV6
  memcpy(&uip_lladdr.addr, node_mac, sizeof(uip_lladdr.addr));
  queuebuf_init();
  process_start(&tcpip_process, NULL);
  uip_ipaddr_t ipaddr;
  uip_ip6addr(&ipaddr, 0xfc00, 0, 0, 0, 0, 0, 0, 0);
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
#endif /* UIP_CONF_IPV6 */
  printf("Done!\n");

  /* Initialize energy estimation */
  energest_init();
  ENERGEST_ON(ENERGEST_TYPE_CPU);

  /* Start user processes */
  print_device_config();
  leds_off(LEDS_ALL);
  printf("----------[ Running %s on LoRaMote ]----------\n\n",
    CONTIKI_VERSION_STRING);
  print_processes(autostart_processes);
  autostart_start(autostart_processes);

  /* Start the process scheduler loop */
  watchdog_start();
  while(1) {

    int r;
    do {
      watchdog_periodic();                        // Reset watchdog
      r = process_run();
    } while(r > 0);

    /* Avoid LPM when a device is connected to serial I/O */
    if(process_nevents() == 0 && !Virtual_ComPort_IsOpen()){
      ENERGEST_OFF(ENERGEST_TYPE_CPU);
      ENERGEST_ON(ENERGEST_TYPE_LPM);
      watchdog_stop();
      lpm_enter_stopmode();                       // Enter LPM: Stop mode with RTC
      watchdog_start();
      ENERGEST_OFF(ENERGEST_TYPE_LPM);
      ENERGEST_ON(ENERGEST_TYPE_CPU);
    }

  }

  return 0;
}
/*---------------------------------------------------------------------------*/
```

```c
static void print_processes(struct process * const processes[])
{
  /* const struct process * const * p = processes; */
  printf("Starting");
  while(*processes != NULL) {
    printf(" '%s'", (*processes)->name);
    processes++;
  }
  printf("\n");
}
/*---------------------------------------------------------------------------*/
static void print_device_config(void)
{
  int i;
  uint8_t longaddr[8];
  uint16_t shortaddr;

  printf("Rime started with address ");
  for(i = 0; i < sizeof(rime_addr.u8) - 1; i++) {
    printf("%d.", rime_addr.u8[i]);
  }
  printf("%d\n", rime_addr.u8[i]);

  shortaddr = (linkaddr_node_addr.u8[0] << 8) + linkaddr_node_addr.u8[1];
  memset(longaddr, 0, sizeof(longaddr));
  linkaddr_copy((linkaddr_t *)&longaddr, &linkaddr_node_addr);
  printf("MAC %02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x  ",
        longaddr[0], longaddr[1], longaddr[2], longaddr[3],
        longaddr[4], longaddr[5], longaddr[6], longaddr[7]);

  if(node_id) {
    printf("Node id is set to %u\n", node_id);
  }else{
    printf("Node id not set.\n");
  }

  printf("%s, %s, radio frequency %iMHz\n",
        NETSTACK_MAC.name, NETSTACK_RDC.name, RF_FREQUENCY/1000000);

#if UIP_CONF_IPV6
  printf("Tentative link-local IPv6 address ");
  {
    uip_ds6_addr_t *lladdr;
    int i;
    lladdr = uip_ds6_get_link_local(-1);
    for(i = 0; i < 7; ++i) {
      printf("%02x%02x:", lladdr->ipaddr.u8[i * 2], lladdr->ipaddr.u8[i * 2 + 1]);
    }
    printf("%02x%02x\n", lladdr->ipaddr.u8[14], lladdr->ipaddr.u8[15]);
  }

  if(!UIP_CONF_IPV6_RPL) {
    printf("Tentative global IPv6 address ");
    for(i = 0; i < 7; ++i) {
      printf("%02x%02x:", ipaddr.u8[i * 2], ipaddr.u8[i * 2 + 1]);
    }
    printf("%02x%02x\n", ipaddr.u8[7 * 2], ipaddr.u8[7 * 2 + 1]);
  }
#endif /* UIP_CONF_IPV6 */
}
/*---------------------------------------------------------------------------*/
static void set_rime_addr(void)
{
  memset(&rime_addr, 0, sizeof(linkaddr_t));
#if UIP_CONF_IPV6
  memcpy(rime_addr.u8, node_mac, sizeof(rime_addr.u8));
#else
  if(node_id == 0) {
    for(int i = 0; i < sizeof(linkaddr_t); ++i) {
      rime_addr.u8[i] = node_mac[7 - i];
    }
  } else {
    rime_addr.u8[0] = node_id & 0xff;
    rime_addr.u8[1] = node_id >> 8;
  }
#endif /* UIP_CONF_IPV6 */
  linkaddr_set_node_addr(&rime_addr);
}
/*---------------------------------------------------------------------------*/
```

A.15 OpenOCD configuration file

This configuration file is to be placed at `/tools/loramote/loramote.cfg`.

```
# This is the OpenOCD config file for a LoRaMote Demo board with a single
    STM32L151C8 chip
source [find interface/stlink-v2.cfg]
transport select hla_swd
set WORKAREASIZE 0x4000
source [find target/stm32l1.cfg]

# Use hardware reset, connect under reset
reset_config srst_only

# Print stdout to console (GDB commands)
init
arm semihosting enable
```

A.16 CPU Makefile

This make should be located at `/cpu/arm/stm32l1/Makefile.stm32l151`.

```
# Makefile for the STM32L151C8 Cortex M3 medium-density microcontroller

.SUFFIXES:


# CPU folder
CONTIKI_CPU = $(CONTIKI)/cpu/arm/stm32l1

# Source folders for Contiki CPU files, ARM CMSIS and STM32L1 libraries
CONTIKI_CPU_DIRS = . \
                ../common/CMSIS                 \
                $(CONTIKI_MCU_DIRS)

# Source files: proprietary sources for startup. Refer to CMSIS docs.
PROP_SYS_ARCH_C = system_stm32l1xx.c
PROP_SYS_ARCH_S = startup_stm32l1xx_md.s


ifndef IAR
  GCC = 1
endif


# Source files: Contiki arch source files
CONTIKI_CPU_ARCH = \
                clock.c                         \
                watchdog.c                      \
                lpm-arch.c                      \
                rtimer-arch.c                   \
                rtc-arch.c                      \
                uart-arch.c

ifdef GCC
  CONTIKI_CPU_PORT = syscalls.c
  STM32_TOOLS ?= $(CONTIKI)/tools/stm32
  GCC_BINS ?= $(STM32_TOOLS)/gcc-arm-none-eabi/bin
else
  CONTIKI_CPU_PORT =
endif

UIPDRIVERS = uip-arch.c

# To be implemented
ELFLOADER =

# Source files: STM32L1 libraries
CONTIKI_MCU_DIRS = \
                STM32L1xx_StdPeriph_Driver/inc                          \
                STM32L1xx_StdPeriph_Driver/src                          \
                ../../stm32_common/                                     \
                ../../stm32_common/STM32_USB-FS-Device_Driver/inc       \
                ../../stm32_common/STM32_USB-FS-Device_Driver/src

FULL_MCU_DIRS = ${wildcard ${addprefix $(CONTIKI_CPU)/, $(CONTIKI_MCU_DIRS)}}
CONTIKI_MCU_SOURCEFILES = ${foreach d, $(FULL_MCU_DIRS), ${subst ${d}/,,${wildcard
     $(d)/*.c}}}


# Add CPU folder to search path for .s (assembler) files
ifdef GCC
  vpath %.s $(CONTIKI_CPU)/arm-gcc
else
  vpath %.s $(CONTIKI_CPU)/arm-std
endif

# Include all files above
ssubst = ${patsubst %.s,%.o,${patsubst %.s79,%.o,$(1)}}
CONTIKI_SOURCEFILES += $(PROP_SYS_ARCH_C) $(CONTIKI_CPU_ARCH) $(CONTIKI_CPU_PORT)
    $(ELFLOADER) $(UIPDRIVERS) $(CONTIKI_MCU_SOURCEFILES)
PROJECT_OBJECTFILES += ${addprefix $(OBJECTDIR)/,$(CONTIKI_TARGET_MAIN:.c=.o)}
PROJECT_OBJECTFILES += ${addprefix $(OBJECTDIR)/,${call ssubst, $(PROP_SYS_ARCH_S)
    }}
#CONTIKI_OBJECTFILES += ${addprefix $(OBJECTDIR)/,${call ssubst, $(PROP_SYS_ARCH_S
    )}}


# Defines common for IAR and GCC ------------------------------------------------

# Set CPU speed in Hz, NB this might have unexpected side-effects if not at 32
# Mhz as it is not immediately clear how specialized the startup code etc is.
# That being said, setting to 24MHz seems to work fine, looking at Contiki clocks
# at least.
F_CPU = 32000000

CFLAGS += \
        -DHSE_VALUE=$(F_CPU)ul          \
        -DUSE_STDPERIPH_DRIVER          \
        -DSTM32L1XX_MD                  \
        -DIAR_ARM_CM3


# IAR ---------------------------------------------------------------------------

# GCC ---------------------------------------------------------------------------

```

```
### Compiler definitions
GCC     = 1
CC      = $(GCC_BINS)/arm-none-eabi-gcc
#LD     = $(GCC_BINS)/arm-none-eabi-ld
LD      = $(GCC_BINS)/arm-none-eabi-gcc
SIZE    = $(GCC_BINS)/arm-none-eabi-size
AS      = $(GCC_BINS)/arm-none-eabi-as
AR      = $(GCC_BINS)/arm-none-eabi-ar
NM      = $(GCC_BINS)/arm-none-eabi-nm
OBJCOPY = $(GCC_BINS)/arm-none-eabi-objcopy
STRIP   = $(GCC_BINS)/arm-none-eabi-strip
GDB     = $(GCC_BINS)/arm-none-eabi-gdb

ASFLAGS += -mcpu=cortex-m3 -mthumb

# This platform wields a STM32L151C8 medium-density device
CFLAGS += -DSTM32L1XX_MD=1

CFLAGS += \
        -I.                                                         \
        -I$(CONTIKI)/core                                           \
        -I$(CONTIKI_CPU)                                            \
        -I$(CONTIKI)/platform/$(TARGET)                             \
        ${addprefix -I,$(APPDIRS)}                                  \
        ${addprefix -I,$(CONTIKI_CPU_DIRS)}                         \
        -Wall -g -g2                                                \
        -DWITH_UIP -DWITH_ASCII                                     \
        -mcpu=cortex-m3                                             \
        -mthumb                                                     \
        -mfix-cortex-m3-ldrd                                        \
        -std=gnu99                                                  \
        -Wno-strict-aliasing                                        \
        -Wno-pointer-sign                                           \
        -Wno-unused-function                                        \
        -Wno-unused-variable                                        \
        -Wno-unused-but-set-variable

LDFLAGS += \
        -L$(CONTIKI_CPU)/arm-gcc                                    \
        -T$(LDSCRIPT)                                               \
        -mcpu=cortex-m3                                             \
        -mthumb                                                     \
        -mfloat-abi=soft                                            \
        -nostartfiles                                               \
        --specs=nosys.specs                                         \
        -Wl,-Map=$(OBJECTDIR)/contiki-$(TARGET).map,--cref          \
        $(LDLIBS)

LDLIBS = -lc -lm

ifeq ($(strip $(REDIRECT_STDIO)),1)
  CFLAGS += -DREDIRECT_STDIO
else  #REDIRECT_STDIO
  LDFLAGS += --specs=rdimon.specs
  LDLIBS += -lrdimon
endif  #REDIRECT_STDIO

ifeq ($(strip $(SMALL)),1)
  CFLAGS += -Os -ffunction-sections -fdata-sections
  LDFLAGS += -Wl,--gc-sections --specs=nano.specs
  LDFLAGS += -Wl,--undefined=_reset_vector__,--undefined=InterruptVectors,--
    undefined=_copy_data_init__,--undefined=_clear_bss_init__,--undefined=
    _end_of_init__
endif  #SMALL


# Build rules ------------------------------------------------------------------
CUSTOM_RULE_C_TO_OBJECTDIR_O=yes
CUSTOM_RULE_C_TO_CE=yes
```

```
CUSTOM_RULE_C_TO_CO=yes
CUSTOM_RULE_C_TO_O=yes
CUSTOM_RULE_S_TO_OBJECTDIR_O=yes
CUSTOM_RULE_LINK=yes

%.o: %.c
	$(TRACE_CC)
	$(Q)$(CC) $(CFLAGS) -c $< -o $@

%.o: %.s
	$(TRACE_AS)
	$(Q)$(AS) $(ASFLAGS) -c $< -o $@\

define FINALIZE_CYGWIN_DEPENDENCY
  sed -e 's/ \([A-Z]\):\\/ \/cygdrive\/\L\1\//' -e 's/\\\([^ ]\)/\/\1/g' \
	<$(@:.o=.P) >$(@:.o=.d); \
  rm -f $(@:.o=.P)
endef

$(OBJECTDIR)/%.o: %.c | $(OBJECTDIR)
	$(TRACE_CC)
	$(Q)$(CC) $(CFLAGS) -c $< -o $@

$(OBJECTDIR)/%.o: %.s | $(OBJECTDIR)
	$(TRACE_AS)
	$(Q)$(AS) $(ASFLAGS) $< -o $@

%.co: %.c
	$(TRACE_CC)
	$(Q)$(CC) $(CFLAGS) -c -DAUTOSTART_ENABLE -c $< -o $@

%.ce: %.o
	$(TRACE_LD)
	$(Q)$(LD) $(LDFLAGS) --relocatable -T $(CONTIKI_CPU)/merge-rodata.ld $< -o
     $@ $(LDLIBS)
	$(STRIP) -K _init -K _fini --strip-unneeded -g -x $@

%-stripped.o: %.c
	$(TRACE_CC)
	$(Q)$(CC) $(CFLAGS) -c $< -o $@
	$(STRIP) --strip $@

%-stripped.o: %.o
	$(STRIP) --strip $@ $<

%.o: ${CONTIKI_TARGET}/loader/%.S
	$(TRACE_AS)
	$(Q)$(AS) -o $(notdir $(<:.S=.o)) $<

ifdef IAR
  %.$(TARGET): %.co $(PROJECT_OBJECTFILES) contiki-$(TARGET).a $(STARTUPFOLDER) #
    $(OBJECTDIR)/empty-symbols.o
	$(TRACE_LD)
	$(Q)$(LD) $(LDFLAGS) -o $@ $(filter-out %.a,$^) $(filter %.a,$^) $(LDLIBS)
else
  CONTIKI_CPU_OBJS=$(CONTIKI_CPU_PORT:%.c=$(OBJECTDIR)/%.o)
  %.$(TARGET): %.co $(PROJECT_OBJECTFILES) $(PROJECT_LIBRARIES) contiki-$(TARGET).
    a $(OBJECTDIR)/symbols.o
	$(TRACE_LD)
	$(Q)$(LD) $(LDFLAGS) $(TARGET_STARTFILES) ${filter-out %.a,$^} -Wl,-\( ${
    filter %.a,$^} $(TARGET_LIBFILES) -Wl,-\) $(CONTIKI_CPU_OBJS) -o $@ $(LDLIBS)
	@echo >> $(OBJECTDIR)/contiki-$(TARGET).map
	$(Q)$(SIZE) $(SIZEFLAGS) $@ >> $(OBJECTDIR)/contiki-$(TARGET).map
endif

%.ihex: %.$(TARGET)
	$(Q)$(OBJCOPY) -O ihex $^ $@

%.hex: %.ihex
```

```
        #@rm $*.hex
        @mv -f $*.ihex $*.hex

%.bin: %.$(TARGET)
        $(Q)$(OBJCOPY) -O binary $^ $@

.PHONY: symbols.c

symbols.c:
        $(Q)cp ${CONTIKI}/tools/empty-symbols.c symbols.c
        $(Q)cp ${CONTIKI}/tools/empty-symbols.h symbols.h

# Don't use core/loader/elfloader.c, use elfloader-otf.c instead
$(OBJECTDIR)/elfloader.o:
        echo -n >$@
```

A.17 Platform Makefile

The LoRaMote platform makefile should be located at `/platform/loramote/Makefile.loramote`.

```
# User settings ------------------------------------------------------------

VERBOSE = 0              # Verbosity control
SMALL = 1                # Create small binaries
REDIRECT_STDIO ?= 1      # Redirect standard I/O to USART/USB
USER_CFLAGS +=


# Target settings ------------------------------------------------------------

STM32_TOOLS = $(CONTIKI)/tools/stm32
LORA_TOOLS = $(CONTIKI)/tools/loramote
STLINK = $(STM32_TOOLS)/stlink
DFU_UTIL = $(STM32_TOOLS)/dfu-util
OPENOCD = $(STM32_TOOLS)/openocd
SERIALDUMP = $(STM32_TOOLS)/serialdump


CONTIKIVERSIONX:=${shell git --git-dir ${CONTIKI}/.git describe --tags --always}
ifneq ($(findstring 3.,$(CONTIKIVERSIONX)),)
  CONTIKI3 = 1
  CFLAGS += -DCONTIKI3=1
endif

ifeq ($(strip $(VERBOSE)),1)
  V ?= 1
endif

ifneq ($(strip $(CONTIKI_WITH_RIME)),1)
  UIP_CONF_IPV6 = 1
  CONTIKI_WITH_IPV6 = 1
  CFLAGS += -DWITH_UIP6=1 -DUIP_CONF_IPV6=1
endif

CFLAGS += -DUSE_DEBUGGER -DUSE_NO_TIMER -DLOW_POWER_MODE_ENABLE -DUSE_BAND_868 -
    DUSE_MODEM_LORA -DUSE_USB_CDC $(USER_CFLAGS)

MODULES += core/net core/net/mac core/net/mac/contikimac

ifdef CONTIKI3
  MODULES += core/net/llsec
else
  MODULES += core/net/ip core/net/ipv6 core/net/rime core/net/rpl
```

```
endif

CONTIKI_TARGET_DIRS = . apps dev LoRaMac net \
                        LoRaMac/board LoRaMac/mac LoRaMac/peripherals LoRaMac/system LoRaMac/system/crypto \
                        LoRaMac/board/usb/cdc/inc LoRaMac/board/usb/cdc/src \
                        LoRaMac/radio LoRaMac/radio/sx1272

CONTIKI_TARGET_MAIN = contiki-loramote-main.c

FULL_TARGET_DIRS = ${wildcard ${addprefix $(CONTIKI)/platform/loramote/, $(CONTIKI_TARGET_DIRS)}}
CONTIKI_TARGET_SOURCEFILES = altitude-sensor.c battery-sensor.c button-sensor.c leds-arch.c lora-contiki-interface.c pressure-sensor.c radio-sensor.c serial-line-arch.c slip-arch.c temperature-sensor.c \
                                adc-board.c board.c gpio-board.c i2c-board.c spi-board.c rtc-board.c sx1272-board.c timer-board.c uart-board.c uart-usb-board.c usb-cdc-board.c \
                                usb_desc.c usb_endp.c usb_istr.c usb_prop.c usb_pwr.c \
                                gpio-ioe.c mpl3115.c sx1509.c sx9500.c \
                                adc.c fifo.c gpio.c i2c.c loratimer.c uart.c \
                                sx1272.c lora-radio-arch.c
#CONTIKI_TARGET_SOURCEFILES = ${foreach d, $(FULL_TARGET_DIRS), ${subst ${d}/,,${wildcard $(d)/*.c}}}

include $(CONTIKI)/cpu/arm/stm32l1/Makefile.stm32l151
CONTIKI_SOURCEFILES += node-id.c $(CONTIKI_TARGET_SOURCEFILES)


ifndef MOTELIST
  USBDEVPREFIX =
  MOTELIST = $(LORA_TOOLS)/motelist-lora
  MOTES = $(shell $(MOTELIST) -c 2>&- | \
        cut -f 2 -d , | \
        perl -ne 'print $$1 . " " if(m-(/dev/\w+)-);')
  CMOTES = $(MOTES)
endif



# Build rules -----------------------------------------------------------

CLEAN += *.loramote symbols.c symbols.h


# Show connected motes
motelist:
        $(Q)$(MOTELIST)

motes:
        @echo $(CMOTES)


# Compile flashable binary
flash-init:
        $(Q)[ -f latest_build_dfu~ ] && rm -f ./obj_loramote/lora-contiki-interface.o || :      # Make sure that the previously built DFU object is not used
        $(Q)rm latest_build_dfu~ -f
        $(Q)touch latest_build_flash~

%.flash: LDSCRIPT = stm32l1xx_md_flash.ld
%.flash: | flash-init %.bin
        $(Q)mv $(@:.flash=.$(TARGET)) $(@:.flash=.flash.$(TARGET))
        $(Q)cp $(@:.flash=.bin) $(@:.flash=.flash.bin)
        $(Q)$(MAKE) flash $(@:.flash=.flash.bin)


```

```
# Compile DFU-uploadable binary
upload-init:
	$(Q)[ -f latest_build_flash~ ] && rm -f ./obj_loramote/lora-contiki-interface.o || :    # Make sure that the previously built flash object is not used
	$(Q)rm latest_build_flash~ -f
	$(Q)touch latest_build_dfu~

%.upload: LDSCRIPT = stm32l1xx_md_flash_offset.ld
%.upload: CFLAGS += -DUSE_BOOTLOADER
%.upload: | upload-init %.bin
	$(Q)mv $(@:.upload=.$(TARGET)) $(@:.upload=.dfu.$(TARGET))
	$(Q)cp $(@:.upload=.bin) $(@:.upload=.dfu.bin)
	$(Q)$(MAKE) upload $(@:.upload=.dfu.bin)


# Erase flash memory
erase-flash:
	$(Q)$(STLINK)/st-flash erase

# Install bootloader
install-bootloader:
	$(Q)$(MAKE) erase-flash
	$(Q)$(STLINK)/st-flash --reset write $(LORA_TOOLS)/loramote-bootloader/loramote_bootloader.bin 0x8000000

# Flash executable code to loramote
flash:
	$(Q)$(STLINK)/st-flash --reset write $(filter $(wildcard *.flash.bin),$(MAKECMDGOALS)) 0x8000000

# Flash executable code to loramote, when a bootloader is present
bootloader-flash:
	$(Q)$(STLINK)/st-flash --reset write $(filter $(wildcard *.dfu.bin),$(MAKECMDGOALS)) 0x8003000

# Upload executable code to loramote
upload:
	$(Q)$(DFU_UTIL)/src/dfu-suffix --add $(filter $(wildcard *.dfu.bin),$(MAKECMDGOALS)) --pid df11 --vid 0483 > /dev/null
	$(Q)$(DFU_UTIL)/src/dfu-util --device 0483:df11 --dfuse-address 0x8003000:leave --download $(filter $(wildcard *.dfu.bin),$(MAKECMDGOALS))


# Start serialview to show output
ifdef MOTE
  serialview:
	$(Q)sleep .5
	$(Q)$(SERIALDUMP)/serialdump-linux -b115200 $(USBDEVPREFIX)$(MOTE) | $(CONTIKI)/tools/timestamp
else
  serialview:
ifeq ($(strip $(CMOTES)),)
	$(Q)$(MAKE) --no-print-directory serialview
endif
	$(Q)$(SERIALDUMP)/serialdump-linux -b115200 $(USBDEVPREFIX)$(firstword $(CMOTES)) | $(CONTIKI)/tools/timestamp
endif

# Start openocd to debug
openocd:
	@echo "REMARK: Set the REDIRECT_STDIO variable in the makefile's user settings section to 0 and rebuild the project before using the OpenOCD debugger!"
	$(Q)($(GDB) -ex 'target extended-remote localhost:3333' -ex 'monitor reset halt' -ex 'continue' -ex 'quit' &)    # Merge output of both commands to same CLI window
	$(Q)$(OPENOCD)/src/openocd -f $(LORA_TOOLS)/loramote.cfg -s $(OPENOCD)/tcl
```