Rainer Hofmann, BSc

X-Burst: Cross-Technology Communication for Off-the-Shelf IoT Devices

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dott. Dott. mag. Dr.techn. MSc Carlo Alberto Boano

Institute of Technical Informatics

Graz, March 2018

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Rainer Hofmann, BSc

X-Burst: Cross-Technology Communication for Off-the-Shelf IoT Devices

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Telematik

eingereicht an der

Technischen Universität Graz

Betreuer

Ass.Prof. Dott. Dott. mag. Dr.techn. MSc Carlo Alberto Boano

Institut für Technische Informatik

Graz, März 2018

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum

Unterschrift

Abstract

For more than a decade, the number of devices connected to the Internet is exponentially increasing. Most of these devices communicate wirelessly between each other, forming the so-called Internet of Things (IoT), and enabling key applications with high societal relevance such as smart homes, smart grids, smart cities, and smart production. Several wireless technologies have been developed to satisfy the different requirements of these IoT applications, such that they can offer the best possible performance. However, this heterogeneity of wireless technologies makes it impossible for co-existing IoT devices to communicate with each other or to share information due to the incompatibility of their physical layers. Giving these heterogeneous devices the ability of communicating with each other would allow them to autonomously coordinate frequency usage and minimize cross-technology interference, as well as to synchronize their clocks without the need of expensive and inflexible gateways.

This thesis presents a cross-technology communication approach called *X-Burst*, which uses precisely timed energy bursts to exchange information among off-the-shelf wireless devices with incompatible physical layers in the 2.4 GHz ISM band. *X-Burst* has been implemented on the popular TI CC2650 LaunchPad and integrated into the Contiki operating system (OS) in a seamless way, i.e., such that no changes to the core functions of the OS are needed. Furthermore, *X-Burst* can automatically adapt to the normal behavior of the OS, i.e., schedule transmissions and receptions whenever the radio is in low-power mode, by learning the duty cycle schedule of the employed MAC protocol. An experimental evaluation shows that *X-Burst* can establish a bidirectional communication between IEEE 802.15.4 (ZigBee) and Bluetooth Low Energy (BLE) devices with data rates up to 9.23 kbit/s. The evaluation further shows the robustness of *X-Burst* in the presence of external interference and its memory footprint.

Kurzfassung

Seit mehr als einem Jahrzehnt steigt die Anzahl der Geräte welche mit dem Internet verbunden sind, exponentiell. Dabei kommuniziert der Großteil dieser Geräte drahtlos untereinander und bildet dabei das sogenannte Internet der Dinge. Dies ermöglicht völlig neue, gesellschaftlich relevante Anwendungen wie intelligente Häuser, intelligente Stromnetze, intelligente Städte und intelligente Produktionen. Um die unterschiedlichen Anforderungen dieser Anwendungen zu erfüllen und um die bestmögliche Leistung zu erzielen, wurden verschiedene Funktechnologien entwickelt. Dies führte jedoch dazu, dass viele dieser Geräte nicht miteinander kommunizieren können, da diese, je nach verwendeter Technologie, unterschiedliche Bitübertragungsschichten besitzen, was einen Informationsaustausch auf herkömmlichen Weg unmöglich macht. Wäre es diesen Geräten jedoch möglich untereinander zu kommunizieren, könnten die verwendeten Frequenzen ausgetauscht und dadurch technologieübergreifende Störungen verringert werden. Zusätzlich erlaubt eine solche Kommunikation eine technologieübergreifende Synchronisation der Systemuhren der Geräte ohne auf teure und unflexible Gateways mit mehreren Funkmodulen zurückgreifen zu müssen.

Diese Masterarbeit stellt eine technologieübergreifende Kommunikationsmethode namens *X-Burst* vor, welche zeitlich präzise Energiestöße verwendet, um Informationen zwischen handelsüblichen Geräten, welche unterschiedliche Funktechnologien im 2.4 GHz ISM Frequenzband verwenden, austauschen zu können. *X-Burst* wurde auf dem TI CC2650 LaunchPad realisiert und nahtlos in das open source Betriebssystem Contiki integriert, d.h., ohne Änderungen am Betriebssystem vornehmen zu müssen. Im Weiteren passt sich *X-Burst* dem gewöhnlichen Verhalten des Betriebssystems an, indem es die Arbeitsphasen des MAC-Protokolls lernt. Dadurch ist *X-Burst* in der Lage, Übertragungen ausschließlich während des eigentlichen Stromsparmodus durchzuführen um die normalen Kommunikationen des Betriebssystems nicht zu stören. Eine experimentelle Evaluierung hat gezeigt, dass *X-Burst* eine Kommunikation zwischen IEEE 802.15.4 (ZigBee) und Bluetooth Low Energy (BLE) Geräten mit einer Datenübertragungsrate von bis zu 9,23 kbit/s in beide Richtungen ermöglicht. Die Evaluierung zeigt im Weiteren die Robustheit von *X-Burst* in Gegenwart von externen Störungen als auch den zusätzlich benötigten Speicherbedarf.

Acknowledgments

This master thesis was written during the year 2017/2018 at the Institute of Technical Informatics at Graz University of Technology.

First and foremost, I want to thank my supervisor Carlo Alberto Boano for his excellent support during my work on this thesis. No matter how busy he was, he always managed to find some time to help me with the problems I faced. This work would not be at its current scope without his feedback and supervision. Furthermore, I want to thank Michael Spörk for his help and valuable input during this thesis.

I also want to thank my partner Angela for her understanding and always motivating me. Finally, I want to thank my parents for giving me complete freedom about my educational training and for always supporting me. I would certainly not be at this point in my life without them. I especially want to thank my father for supporting me during my whole study.

Graz, March 2018 Rainer Hofmann

Danksagung

Diese Masterarbeit wurde im Jahr 2017/2018 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Zuallererst möchte ich meinem Betreuer Carlo Alberto Boano für seine hervorragende Unterstüzung während meiner Arbeit an dieser Masterarbeit danken. Ganz egal wie beschäftigt er war, er fand immer Zeit mir bei meinen Problemen zu helfen. Diese Arbeit wäre ohne sein Feedback und seine Betreuung nicht in diesem Ausmaß möglich gewesen. Zudem möchte ich auch Michael Spörk für seine Hilfe und Unterstüzung während dieser Masterarbeit danken.

Ich möchte auch meiner Partnerin Angela für ihr Verständnis und dafür, dass sie mich immer motiviert hat, danken. Abschließend möchte ich noch meinen Eltern dafür danken mir völlige Freiheit in der Wahl meiner schulischen Ausbildung gegeben zu haben und dass sie mich immer unterstützt haben. Ohne sie würde ich nicht an diesem Punkt meines Lebens stehen. Ich möchte besonders meinem Vater für seine Unterstützung während meines gesamten Studiums danken.

Graz, März 2018 Rainer Hofmann

Contents

List of Figures

List of Tables

List of Abbreviations

AFH Adaptive Frequency Hopping

BLE Bluetooth Low Energy

CCA Clear Channel Assessment

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance

CTC Cross-Technology Communication

HAL Hardware Abstraction Layer

IEEE Institute of Electrical and Electronics Engineers

IoT Internet of Things

IP Internet Protocol

ISM Industrial, Scientific and Medical

MAC Media Access Control

MCU Microcontroller Unit

OS Operating System

PRR Packet Reception Rate

RAM Random Access Memory

RDC Radio Duty Cycling

ROM Read Only Memory

RSSI Received Signal Strength Indicator

TI Texas Instrument

WPAN Wireless Personal Area Network

Chapter 1

Introduction

Since more than a decade, the number of wireless devices is exponentially increasing. Due to the emerging Internet of Things (IoT) era, this number will soon increase even further. More and more everyday objects are becoming smart nowadays, meaning that they can sense their environment and directly send information to each other or to the Internet wirelessly. This opens the possibility for attractive applications such as smart home systems, where wirelessly-connected sensors and actuators are used for controlling different appliances, measuring the power consumption or increasing the security of inhabitants [1, 2]. Another popular IoT application domain is smart health care, where sensors are used for monitoring daily activities and exercise, as well as the vital signs of patients [3, 4]. There are also many IoT applications with high societal relevance and impact such as smart cars (for increasing driving safety and comfort or providing accident and emergency identification and alert), smart cities (for measuring the air quality or efficiently lighting up the city), or smart grids (for improving the efficiency of production, distribution and consumption of energy) [5, 6, 7]. Besides the consumer market, also the industry is using the advantages of the IoT for controlling and optimizing production processes. Sensors are, for example, used for monitoring critical values, measuring the produced quantity, or sending an alert if an engine has stopped working unexpectedly [8]. Because of all these attractive applications, the number of devices connected to the Internet is now more than 8 billions, with the majority of these devices being wireless sensors embedded in everyday objects. This number is expected to grow to 20 billions by 2020 [9].

1.1 Problem Statement

Large number of heterogeneous wireless technologies. As IoT applications are largely different in nature and requirements (e.g., streaming a movie needs a very high data rate (MB/s) while measuring the temperature and send it wirelessly to a base station works fine with a low data rate (KB/s) but needs to be very energy-efficient to ensure a long lifetime of the device), several wireless technologies have been developed to satisfy the requirements of different applications and to offer the best performance. Typical requirements for wireless technologies are power consumption, communication range, data rate, bandwidth, latency or robustness.

Some of the most popular wireless technologies are WiFi (IEEE 802.11), ZigBee (IEEE 802.15.4) or Bluetooth (IEEE 802.15.1) and its evolution Bluetooth Low Energy (BLE). These standards specify different signal management functions, modulation schemes, data rates, channel bandwidths and separations, which make them incompatible to each other. In industrial measurement and data requisition systems, it is often necessary to use heterogeneous technologies for measuring the performance of a system or observing the same event. Without proper synchronization between the different technologies, the timestamps referring to the same event, measured by heterogeneous devices, would be different and the collected data would not be useful. Thus, a rational analysis of the collected data would not be feasible. A proper synchronization can be achieved by using very expensive multi-radio gateways, which allow communication among heterogeneous networks. Nevertheless, this approach suffers from several drawbacks such as additional hardware costs, complex network structure and increased traffic overhead. Additionally, a gateway is also a single point of failure since all communication between different networks is routed over it. Hence, a better solution for enabling a communication among heterogeneous networks has to be established.

Large number of IoT devices crowding the same frequencies. Wireless devices often make use of the (unlicensed) Industrial, Scientific and Medical (ISM) frequency bands. These are freely available portions of the radio spectrum reserved for industrial, scientific and medical purpose only. As the number of wireless devices is constantly increasing, the congestion of the ISM bands is rising, turning the radio spectrum into an expensive resource [10]. A well-known example of how crowded a portion of the radio spectrum can be is the 2.4 GHz ISM band. Its worldwide availability and the fact that it is free to use made this band one of the most popular choices for a lot of different technologies. Figure 1.1 shows the four most pervasive technologies operating in the 2.4 GHz ISM band: WiFi, ZigBee[1], Bluetooth and its evolution Bluetooth Low Energy.

[1] When we write ZigBee, we actually mean IEEE 802.15.4

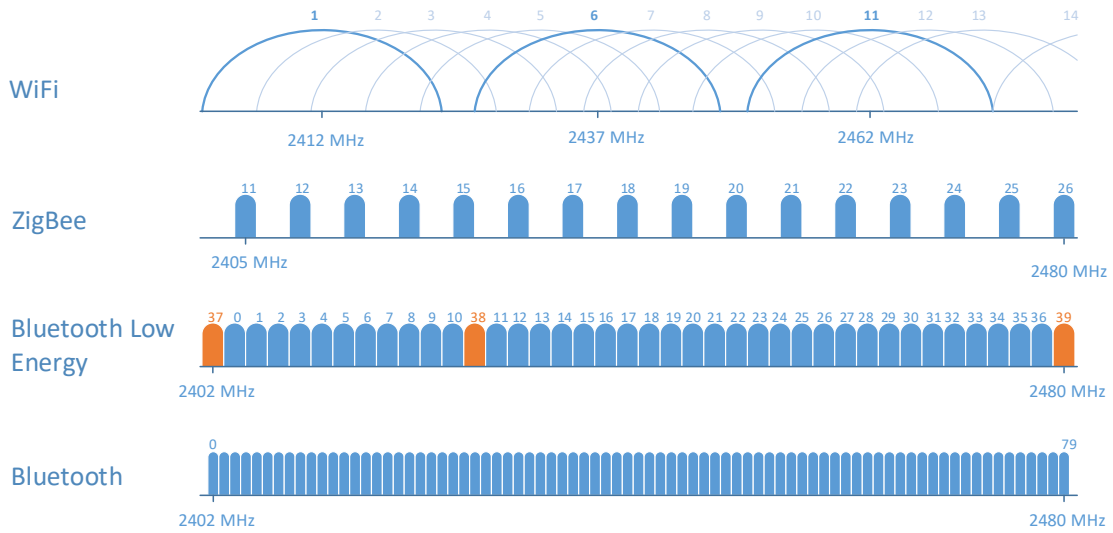


Figure 1.1: Overlapping channels of WiFi, ZigBee, BLE and Bluetooth.

As shown in Figure 1.1, all these technologies employ overlapping frequencies. As a result, standard-compliant devices need to compete for medium access and may experience cross-technology interference from surrounding appliances. To avoid collisions, WiFi is using Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) and ZigBee is using a Clear Channel Assessment (CCA) to check if the channel is free. Bluetooth and BLE are using Adaptive Frequency Hopping (AFH) to escape interference from other wireless technologies. Due to the high number of devices and the consequent congestion of the radio spectrum, these mechanisms are pushed to their limits and may no longer be sufficient to avoid collisions.
This leads to an increased packet loss and to a higher number of retransmissions affecting the latency, throughput and energy efficiency of the involved networks. As result, in non-critical applications (e.g., home automation), one can often experience packet loss and bad performance (e.g., higher latency when controlling appliances). In safety-critical applications (e.g., smart productions and smart grid applications), manual complex network planning is needed to ensure coexistence while fulfilling the requirements (i.e., throughput, latency) of applications [11]. Hence, a mechanism for communicating the used channel among devices with different physical layer is needed to reduce cross-technology interference.

An ideal solution to both aforementioned problems would be to allow devices from heterogeneous technologies to directly communicate with each other, without the need of additional hardware.

1.2 Thesis Contributions

In order to allow heterogeneous devices to communicate seamlessly, we explore Cross-Technology Communication (CTC), a mechanism that allows direct communication among wireless devices with incompatible physical layer. This is achieved by encoding data in such a way, that, independent from the used technology, every device in range is able to detect and decode it. Existing CTC approaches encode information by using the transmission power of packets [12] or the duration of energy bursts [13]. The information can be decoded by measuring the Received Signal Strength Indicator (RSSI) which is typically a feature offered by all IoT technologies. CTC can be used to make networks aware of each other and to proactively avoid cross-technology interference, as well as to seamlessly synchronize the clocks of nodes employing heterogeneous technologies.

Exploring cross-technology communication for IoT devices in the 2.4 GHz ISM band. In this thesis, we have developed a mechanism called *X-Burst*, which uses precisely timed energy bursts to exchange information among nodes with incompatible physical layer. Towards this goal, we have defined 16 different energy burst durations, each one encoding four bits of information, i.e., representing a different hex value in the range of 0x0 - 0xF. *X-Burst* focuses on ZigBee and BLE as they are the most used technologies for IoT devices. Furthermore, it allows a bidirectional communication without any hardware modifications or the need of a multi-radio gateway by only using **off-the-shelf devices**.

Integration of CTC primitives in Contiki. We have integrated *X-Burst* into the Contiki Operating System (OS) in a seamless way, i.e., no changes to the core functions of the system, the network stack, the radio driver or any other implementation are needed. Furthermore, we do not affect the normal behavior of the operating system, i.e., if a duty cycle is detected, *X-Burst* adapts to it and is only active during the usual *off-phases* of the device. Hence, we can guarantee that *X-Burst* does not violate the normal communication flow of the operating system. To achieve this, we have written our own radio driver for Contiki, which is responsible for managing the normal communications of the OS and the CTC related ones in a non disruptive way.

Evaluation on real hardware. We have evaluated *X-Burst* in terms of throughput, power consumption, memory footprint and robustness. Additionally, the behavior of *X-Burst* and its adaption to different use cases are shown. As hardware, we used the Texas Instrument (TI) multi-standard CC2650 LaunchPad, given that its radio supports both, ZigBee and BLE.

1.3 Thesis Structure

This thesis is structured as follows. In Chapter 2, an overview about the most recent works in the area of CTC is given. Furthermore, the limitations of the presented works are explained and a short summary, including a comparison with *X-Burst*, is given at the end of the chapter. The requirements for establishing a communication between heterogeneous devices are listed in Chapter 3. Besides the list of requirements, the overall concept of *X-Burst* is presented and the used wireless technologies are explained briefly. The design challenges, including, the generation and detection of energy bursts, the structure of CTC messages and the aspects that have to be considered for a seamless integration into an existing operating system, are explained in Chapter 4. Chapter 5 covers the seamless integration into the Contiki OS and explains the features of the developed radio driver in detail. Furthermore, the possible configurations of the new radio driver are explained and the portability of the implementation is discussed. Additionally, the adaption to different duty cycling mechanisms is shown and the implementation is described in detail. An evaluation of *X-Burst* is given in Chapter 6. In particular, a validation showing a working communication between two heterogeneous devices is presented, and the achievable throughput is evaluated and compared with the theoretical one. Furthermore, an evaluation of the energy and memory consumption, as well as of the robustness of *X-Burst* in the presence of interference is given. At the end of the chapter, the adaptation of *X-Burst* and its configurable behavior are shown. Finally, Chapter 7 completes the thesis with a conclusion and an outlook about the future development of *X-Burst*.

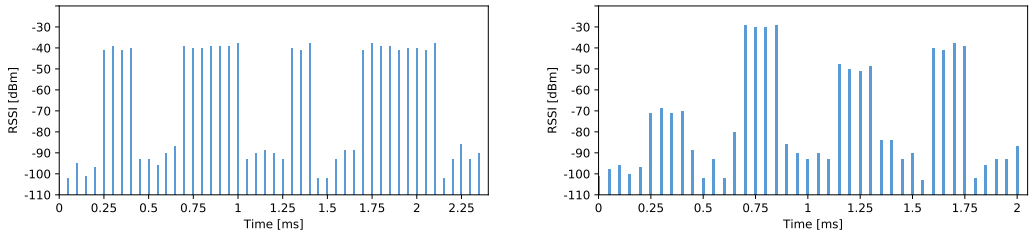
Chapter 2

Related Work

This chapter gives an overview about the most important works in the field of Cross Technology Communication (CTC) in Section 2.1. Thereafter, Section 2.2 discusses the limitations of the presented works and gives a short summary including a comparison with *X-Burst*, the CTC scheme presented in this thesis.

2.1 Existing CTC Approaches

In this section, an overview about four of the most important works in the field of CTC is given. Cross-technology communication is achieved by encoding data in such a way that it can be detected independently of the underlying technology. This is usually achieved by using the transmission power or the duration of transmitted packets, or by shifting periodic signals. Hence, CTC can be divided into two groups[1]: time modulation and energy modulation. The main requirement is that a receiver can detect this information, e.g., by measuring the quality of a radio channel. Figure 2.1 shows the classification of cross-technology communication.



(a) Time modulation

(b) Energy modulation

Figure 2.1: Classification of cross-technology communication - the information is encoded in (a): the timing of packets, (b): the energy of packets.

[1] In fact, there is also a third group: emulation of legitimate packets. This is in most cases not possible and usually not used to allow cross-technology communication.

2.1.1 B^{2}W^2: N-Way Concurrent Communication for IoT Devices

B^2W^2 [12] is a novel communication framework that allows BLE devices to communicate with WiFi devices while supporting the concurrent BLE to BLE and WiFi to WiFi communications. The authors use the transmission power of BLE packets to form a sine wave and encode the data into the frequency of the wave. The sine wave is created by directly adjusting the transmission power level of adjacent BLE packets. Therefore, a so called "Discrete Amplitude and Frequency-Shift Keying" (DAFSK) converter was developed, which converts the data into a sequence of symbols. Each symbol represents the transmission power of a BLE packet. These symbols are then mapped to a subset of channels that overlap with the target WiFi channel and the transmission power is adjusted accordingly. For demodulation on the WiFi side Channel State Information (CSI) is used. Therefore, the WiFi channel is divided into tens of subcarriers with a bandwidth several times narrower than a BLE channel. This allows the detection of the power level of the BLE packets within the CSI values of the subcarriers. The original message can be reconstructed by measuring the changes in the CSI values of all subcarriers over a specified time period. The frequency shift keying technique is used to be able to reconstruct the data even in dynamically changing environments. Modulating the data directly on the transmission power level of BLE packets would result in an extremely high Bit Error Rate (BER). With this approach, a throughput of about 1.5 kbps was achieved without creating extra traffic or changing the frequency hopping sequence of BLE. For the evaluation, a TI CC2650 device was used as BLE transmitter and a National Instruments (NI) RF testbed as WiFi receiver.

One big issue of this approach is the need of collisions. The power level of a BLE packet can only be detected if it collides with a WiFi packet. Usually, mechanisms are used to avoid collisions. WiFi uses CSMA/CA to check if the channel is free before sending a packet. BLE uses AFH which avoids (blacklists) channels with a high Packet Loss Rate (PLR). If each mechanism works correctly, this CTC approach might not work anymore. Another drawback of this method is the fact that it only works in one direction, from BLE to WiFi. There is no possibility for the WiFi device to communicate with the BLE one.

2.1.2 FreeBee: Cross-Technology Communication via Free Side-Channel

FreeBee [14] is a novel communication framework that allows direct communication among WiFi, ZigBee and BLE. The authors use the already existing periodic beacon frames of the three technologies to encode symbols. This technique allows CTC without the need of additional bandwidth or any impact on the normal communications. The symbols are encoded by slightly shifting (advance or delay) the transmission time of the periodic beacon frame. This is known as Pulse Position Modulation (PPM). To guarantee the detection of the shifted beacon, it is sent multiple times.
Due to the incompatible PHY layers, the receivers cannot directly detect the beacon frame of other technologies. Thus, the RSSI of the channel is measured frequently and quantized to binaries. To detect the periodic beacon frame, the signal processing technique *folding* is applied to the captured values. During an initial learning phase, the reference position of the beacon frame is learned. Afterwards symbols can be decoded from deviations of the reference position. To get rid of the learning phase, the authors present an extension called *A-FreeBee* (Asynchronous FreeBee). Thereby, the reference frame and the shifted frame are transmitted consecutively. The symbol can be directly interpreted from the difference of the two beacon streams.
(A-)FreeBee allows demodulation of simultaneous transmissions by choosing the beacon intervals of neighboring access points to be pairwise co-prime. This is referred as *interval multiplexing*. With that, even implicit addressing is possible by setting the demodulation interval to the corresponding one of the chosen transmitter.
Among all communication directions, WiFi to ZigBee achieves the fastest rate of 31.5 bps for FreeBee. The rate drops to 14.6 bps for the reverse direction. When sending from BLE to WiFi and ZigBee the throughput is 17.5 bps and 17.8 bps respectively. The performance of A-FreeBee is slightly less than half of that of FreeBee, due to the doubled sampling duration. For the performance evaluation, WARP, a wireless research platform, and various laptops were used as WiFi devices. Further, an off-the-shelf, IOGEAR BLE USB adapter and 30 ZigBee-compliant MICAz nodes were used.

In the paper, Bluetooth is always mentioned but the authors obviously meant Bluetooth Low Energy (BLE), which is something completely different. The proof-of-concept has only been considered with BLE as a transmitter and never as a receiver. Furthermore, only the three advertisement channels 37, 38 and 39 can be used for transmitting data. These channels are chosen in a way to have the least interference with the most common WiFi channels 1, 6 and 11. As result, CTC between BLE and WiFi will be very restricted. Another drawback of this scheme is the very low data rate of about 30 bps which might not be enough for real world applications. Furthermore, FreeBee requires a good channel to guarantee a reliable communication, thus, it strongly depends on the noise of the used channel.

2.1.3 BlueBee: a 10,000x Faster Cross-Technology Communication via PHY Emulation

BlueBee [15] is the latest published work on CTC. It allows a high data rate communication from BLE to ZigBee devices. This is achieved by emulating legitimate ZigBee frames using a BLE radio. Therefore, a ZigBee frame is encapsulated within the BLE packet payload by carefully selecting the bits of the payload. The emulated ZigBee packet is fully compatible with legacy devices and is not distinguishable from a normal ZigBee packet for the receiver. When an emulated frame reaches a receiver, it detects the payload part via preamble and starts demodulating the frame. The BLE header and trailer are treated as noise and are naturally ignored. BlueBee does not need any hardware modification of a BLE sender or ZigBee receiver.
The emulation is only possible due to the similar (de)modulation technique of BLE and ZigBee. BLE uses Gaussian Frequency Shift Keying (GFSK), which is normally realized by phase shift over time, and ZigBee uses Offset Quadrature Phase Shift Keying (OQPSK) combined with Direct Sequence Spread Spectrum (DSSS). ZigBee‘s OQPSK only considers sign (+ or -) of the phase instead of particular values to indicate symbols (chips). This offers a high error tolerance for the demodulation which makes BlueBee possible.
BLE uses frequency hopping to minimize the interference with other wireless radios. If the packet accept ratio of a channel is too low, BLE can use Adaptive Frequency Hopping (AFH) to mark (blacklist) this channel as bad. Thus, this channel is no longer be used for communication. BlueBee uses this feature to control the hopping behavior of BLE in a non-disruptive way to ensure a communication with ZigBee devices.
BlueBee achieves a throughput of 225 kbps which is relatively close to the maximum data rate of 250 kbps of ZigBee. It was implemented using a GNU radio BLE implementation called scapy radio with a USRP-N210 platform as transmitter. Furthermore, also a BLE CC2540 development kit and a commodity smartphone, the Nexus 5X, were used as a transmitter. For the receiver side, a BLE CC2540 development kit, the ZigBee-compliant CC2530 and CC2420 (i.e., MICAz and TelosB), as a 802.15.4 implementation on a USRP-N210 were used.

The authors evaluate that one of the main problem for this approach was the narrower bandwidth of BLE (1MHz) compared to ZigBee (2 MHz). This is not correct as BLE and ZigBee both have a bandwidth of 2 MHz. By taking a closer look to the overlapping channels between these two technologies one can notice that only 50 % of the channels are overlapping completely. The other channels only overlap in a range of 1 MHz, which could be an explanation for their statement. BlueBee uses normal BLE data packets to encapsulate a legitimate ZigBee frame. Due to the BLE specification it is only possible to send BLE data packets during an already established connection. Hence, an existing BLE communication between two BLE devices is needed to be able to communicate with a ZigBee device. Another point is the reverse communication from ZigBee to BLE. The demodulation of BLE is not as flexible as for ZigBee, which makes the demodulation very vulnerable to failures. Therefore, a reliable reverse communication would not be feasible.

2.1.4 Esense: Communication through Energy Sensing

Esense [13] is a communication framework that is based on sensing and interpreting energy profiles. It enables unidirectional communication from WiFi to ZigBee devices by encoding information into the duration of energy bursts. Therefore, an *alphabet set* of different energy burst durations was built, where each burst represents a specific information. The authors mentioned that the alphabet set strongly depends on the data rate of the WiFi sender and the measurement granularity of the receiver.
To minimize false positives, the packet size distribution of different WiFi traces was analyzed and all packet sizes whose frequency of occurrence was greater than a specified percentage were excluded. The authors concluded that the majority of all packets are either below 140 bytes (ACKs, beacons or management frames) or around 1500 bytes, which is the maximum packet length according to the Ethernet MTU. Additionally, a packet is sent multiple times within a specified time-window. Depending on the configuration, the packet is only accepted as an Esene one, if it was received a specified number of times within this time-window.
A receiver frequently measures the energy (noise) on the channel by checking if the RSSI is above a certain threshold. The duration of each detected energy burst is measured in clock ticks and afterwards compared to the values in the alphabet set. If the duration is not valid, it is considered as a regular WiFi packet and discarded, otherwise it is considered as an Esense packet.
The authors did a theoretical evaluation of the possible throughput of Esense. The best achievable data rate is about 5.13 kbps in absence of background traffic. In case of involving background traffic, the throughput depends on the mode of operation of the WiFi transmitter. Using 802.11b, the best achievable throughput is about 1.02 kbps and by using 802.11g about 1.63 kbps. An evaluation based on a preliminary implementation was done to determine the reliability of Esense. Towards this goal, a laptop equipped with an IEEE 802.11b/g WiFi card (with the Linux open-source madwifi driver) was used as transmitter and a Tmote Sky node as receiver. Depending on the configuration, a false-positive and negative rate below 5 % were achieved.

One major drawback of this CTC scheme is the lack of flexibility in exchanging data. Each energy burst duration represents a specified information such as a predefined message, commando or value. Hence, it is not possible to exchange arbitrary data. The authors only mentioned that it would be possible to build a vocabulary out of the values in the alphabet set to increase the number of possible messages.

2.2 Limitations of Existing CTC Approaches

All of the previous discussed CTC approaches have similar problems. Most of them only achieve a **unidirectional communication**. In the case of B^2W^2 and BlueBee, a bidirectional communication is not even possible. FreeBee is the only work that achieves a bidirectional communication between WiFi and ZigBee, but suffers from a very **low data rate**. Another limitation is the need of very special **requirements** such as the **need of collisions** (B^2W^2) or an already **established connection** between two BLE devices (BlueBee). Esense has the major drawback of being only able to send **predefined messages** or commands instead of arbitrary data.

Esense, nevertheless, is the most related work to *X-Burst*. It is also making use of precisely timed energy bursts to exchange information between two devices with incompatible PHY layers. However, in *X-Burst* instead of WiFi, BLE is used. The main difference is that *X-Burst* only needs 16 different energy bursts for representing arbitrary data, independent of its size. Thus, *X-Burst* is not limited to send only predefined messages. Besides that, a bidirectional communication and a much higher throughput is achieved. Furthermore, *X-Burst* was fully tested and evaluated on real hardware, whereas Esense was only partly evaluated.

Table 2.1 summarizes the related works on CTC and compares them with the approach proposed in this thesis, i.e., *X-Burst*.

	B^{2}W^2	**FreeBee**	**BlueBee**	**Esense**	**X-Burst**
Direction	BLE → WiFi	WiFi ↔ ZigBee BLE → ZigBee BLE → WiFi	BLE → ZigBee	WiFi → ZigBee	BLE ↔ ZigBee
Modulation	power level of BLE packets	shifting of periodic beacons	emulating ZigBee frames	energy burst durations	energy burst durations
Reception	Channel State Information (CSI)	RSSI sampling	normal reception	RSSI sampling	RSSI sampling
Throughput	1.5 kbps	up to 31.5 bps	225 kbps	up to 5.13 kbps	up to 9.23 kbps
Reliability	SER[2] $\sim$ 0% in Faraday Cage	SER $<$ 0.5 %	FRR[3] $>$ 85%	FP[4] & FN[5] $<$ 5 %	PRR[6] $>$ 97 %
Hardware Platform	WiFi: - NI RF testbed BLE: - TI CC2650	WiFi: - WARP & Laptop BLE: - IOGEAR USB adapter ZigBee: - MICAz nodes	BLE: - USRP N210 - CC2540 - Nexus 5X ZigBee: - USRP N210 - CC2530 & CC2440	WiFi: - Laptop ZigBee: - Tmote Sky	BLE & ZigBee: - TI CC2650 Launchpad
Limitations	- collisions needed - reverse direction not feasible	- low data rate - BLE only as transmitter	- BLE connection needed - reverse direction not feasible	- no exchange of arbitrary data - unidirectional	

Table 2.1: Summary of the discussed CTC approaches and comparison with *X-Burst*.

[2]Symbol Error Rate
[3]Frame Reception Ratio
[4]False Positives
[5]False Negatives
[6]Packet Reception Rate

Chapter 3

Cross-Technology Communication for Off-the-Shelf IoT Devices

This chapter describes the general requirements needed to allow a cross-technology communication among heterogeneous devices, as well as the specific requirements that should be fulfilled by *X-Burst* in Section 3.1. Section 3.2 presents the overall concept of *X-Burst* and explains its working principle in detail. Additionally, the used wireless technologies, ZigBee and BLE, are described briefly. For more information about the latter, the interested reader can refer to Appendix A.

3.1 Requirements

In this section, the requirements for *X-Burst* are listed in two categories. In Section 3.1.1 the general requirements to enable a communication among heterogeneous devices in general are detailed. Section 3.1.2 describes the requirements that should be fulfilled by *X-Burst*.

3.1.1 Cross-Technology Communication

To be able to establish a communication between different wireless technologies, all the following requirements have to be met.

Operating in the same frequency band. To be able to allow CTC, the different wireless standards have to operate in the same frequency band.

Wireless channels must overlap. At least one of the wireless channels of the different technologies has to overlap in the radio spectrum. The overlap can only be partial or complete, but the more is the overlap, the more robust will be the communication.

Agreeing on a common channel. The heterogeneous devices have to agree on a common channel that is used to carry out cross-technology communication. This is usually done manually, since the devices cannot communicate upfront to agree on a common channel.

Sensing information of a channel. To be able to detect and decode CTC messages, a receiver has to be able to sense information about a wireless channel, e.g., the RSSI. Furthermore, the transmitter and receiver need to agree on a data rate beforehand, which is limited by the sampling rate of the RSSI of the receiver.

3.1.2 X-Burst

In addition to the aforementioned requirements, we would like to satisfy also the following requirements:

Bidirectional communication. Compared to most of the published works on CTC, which only allow unidirectional communications, *X-Burst* has to achieve bidirectional communications among different wireless technologies, do that a complete data exchange between heterogeneous devices can be established.

Data rate of at least 1.63 kbps. *X-Burst* has to achieve at least the same throughput as Esense. The higher the data rate, the more data can be exchanged while reducing the interference with the usual transmissions by needing less time for the cross-technology communication.

Independently from other communications. As discussed in Chapter 2, BlueBee and B^2W^2 strongly depend on other transmissions, i.e., they need collisions or they require an already established connection with another device. *X-Burst* has to be independent of other communications, so that it can be used even if no other device using the same technology is in range.

Support for arbitrary data. *X-Burst* has to be able to transmit arbitrary data, independent of its size.

Supported by off-the-shelf devices. Often software defined radios or especially designed hardware are used to demonstrate CTC. *X-Burst* has to work on any commodity hardware that fulfills the general requirements. Hence, only standard compliant functions available in off-the-shelf devices can be used.

No hardware modifications. *X-Burst* has to support legacy devices. Since no hardware modifications are needed, CTC can easily be brought on legacy devices through a firmware-update.

Easy portability to other hardware platforms. *X-Burst* should not be limited to a specific hardware platform. Thus, it has to be designed in such a way that it is easily portable to other hardware platforms which fulfill the general requirements.

Seamless integration into an open source operating system. *X-Burst* has to be integrated seamlessly into an open source operating system, i.e., the normal communication flow of the OS should not be violated. Furthermore, only minimal changes of the OS are acceptable.

3.2 Concept

This section describes the working principle of *X-Burst* in detail. Section 3.2.1 gives an overview about the principle and briefly describes the used wireless technologies. Afterwards, the encoding and transmission of CTC messages are described in Section 3.2.2. The reception and decoding of CTC messages are shown in Section 3.2.3 and the structure of a CTC message is shown in Section 3.2.4. Figure 3.1 shows the concept of *X-Burst*.

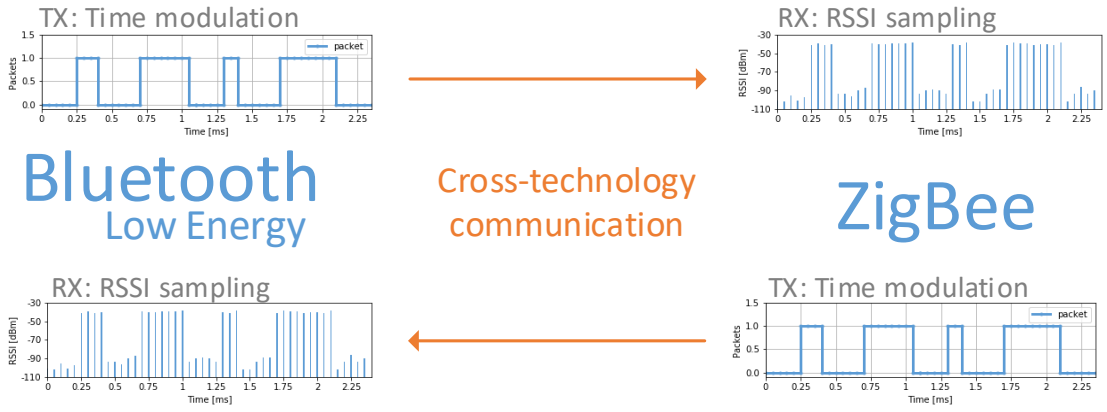


Figure 3.1: Concept of *X-Burst*.

3.2.1 Overview

To enable a communication between different wireless technologies, a common way to encode and decode data has to be established. Most of the wireless standards provide information about the current state of a channel, e.g., they allow to determine the energy of a channel by reading the RSSI register. This information is typically used to determine the current noise of a channel and to avoid collisions with other transmission (CCA). Nevertheless, the noise (or energy) level of a channel can be used to detect transmissions from other devices using heterogeneous technologies.

In an energy based CTC scheme, energy bursts, i.e., noise, are simply generated by sending ordinary data packets. Sensing the noise level of a channel allows any device, independent of the used technology, to detect those bursts. Thereby, three different values can be distinguished: the intensity of a burst, i.e., the absolute received signal power of a packet, the gap between two bursts, and the duration of a burst.
Using the intensity of an energy burst to encode data is not a good choice as it strongly depends on the environmental characteristics, the configured transmission power of a transmitter, and the distance between receiver and transmitter. The gap between two energy bursts is also not a good parameter, since it can not be fully controlled by the transmitter. Another node could, for example, send a message exactly between two energy bursts, which will shorten the length of the gap and thus, alter the encoded information. The third parameter, the duration of an energy burst, seems promising. It is independent of the environment and can be fully controlled by the transmitter. Thus, the duration of energy bursts was chosen to build a novel CTC scheme, i.e., *X-Burst*.

Working Principle

X-Burst is using precisely timed energy bursts to convey information among devices with incompatible PHY layers, i.e., that are using different wireless technologies. The data is encoded as the duration of different energy bursts. In particular, 16 different energy bursts have been defined, where four bits of information is encoded in the duration of each burst. Hence, every burst is representing a different hex value in the range of 0x0 - 0xF. Since any message can be represented by consecutively sending different energy bursts, according to the hex representation of the message, arbitrary data can be transmitted. The energy bursts can be detected by any device that is capable of reading the RSSI of a specified channel. The original message is decoded by translating the received energy bursts into their corresponding hex values and reassembling them correctly.

X-Burst was implemented and tested using the wireless technologies ZigBee and BLE. These standards were chosen because they are the most prevalent technologies used for IoT devices. Both are operating in the 2.4 GHz ISM band and employ overlapping channels. A brief description of both can be found below. For more details, the reader should refer to Appendix A.
However, *X-Burst* is not restricted to ZigBee and BLE, it can be used with every technology and device that fulfills the general requirements described in Section 3.1.1.

ZigBee

ZigBee is a wireless standard for low-rate Wireless Personal Area Networks (WPANs) that builds upon the IEEE 802.15.4 physical radio specification [16]. It operates in the 2.4 GHz ISM band and can use 16 different channels (11-26). Each channel has a bandwidth of 2 MHz and is separated by 5 MHz from the next one. ZigBee achieves a data rate of 250 kbit/s and was designed for low-cost, low power battery operated devices which makes it a good choice for the IoT domain.

Bluetooth Low Energy

BLE is a standardized ultra-low-power wireless technology for short-range WPANs. It achieves a data rate of 1 Mbit/s and is incompatible with Classic Bluetooth. BLE operates in the 2.4 GHz ISM band and can use 40 channels whereby three (37, 38, 39) are only used for advertising. Each channel has a bandwidth of 2 MHz and is separated by 2 MHz from the next one, i.e., the channels are directly next to each other in the radio spectrum. BLE was designed for low-power battery-operated devices with limited hardware resources, making it a good choice for the IoT domain.

3.2.2 Transmitting Messages

As explained in the previous section, *X-Burst* uses 16 different energy bursts to exchange information among heterogeneous devices. Each burst is representing a different hex value. To transmit a message, the data has to be divided into separate bytes and represented as hex values. For each byte, two energy bursts with the corresponding durations are sent successively. The burst representing the hex value of the four most significant bit (high nibble) of a byte is transmitted first, followed by a burst representing the hex value of the four least significant bits (low nibble).
Energy bursts can simply be generated by sending standard-compliant data packets. In this way, the duration of the burst, i.e., the time in which the presence of a data packet in the radio channel is detectable, only depends on two parameters: (i) the data rate of the transmitter and (ii) the length of the message, i.e., the amount of transmitted bytes. Some technologies are able to use different data rates for transmitting messages, thus, the data rate should be fixed for the CTC to prevent decoding errors. If the data rate is fixed, the duration of a burst only depends on the length of a message. Hence, by changing the amount of transmitted bytes, the duration of an energy burst will change accordingly. In particular, only payload bytes in standard-compliant packets are adjusted to generate energy bursts with the required durations. The generation of the different energy bursts is described in more detail in Section 4.1.
Using a general mapping from hex values to payload bytes would lead to a major problem. Heterogeneous technologies are using different data rates for transmission, thus, sending the same amount of bytes would result in different energy bursts, depending on the underling technology. To solve this problem, a general mapping from hex values to durations is used among all technologies. Hence, a receiver can always detect a CTC message, independent of the transmitter of the message. Furthermore, every technology has its own mapping from durations to the required amount of payload bytes needed for generating the different energy bursts. The mapping from durations to the amount of payload bytes has to be done beforehand for each technology. Towards this goal, a number of aspects have to be considered. First, the various durations have to be clearly distinguishable by a receiver. Hence, the durations of an energy burst have to be chosen depending on the measurement granularity of a receiver, otherwise energy bursts could be interpreted falsely. Another important point is the gap between two consecutive energy bursts: if the gap is too short, the receiver will not be able to detect it and the two bursts are thus merged together. If the gap is too long, the receiver will interpret the two related energy bursts as separate ones, which will result in a wrong decoding of the message.

3.2.3 Receiving Messages

To detect CTC messages, a receiver measures frequently the received signal strength of the configured channel. To be able to distinguish between background noise and transmitted messages, a threshold for the RSSI has to be defined. The stream of measured values is analyzed and compared to the specified threshold allowing detection and determining the corresponding duration of energy bursts. The duration measurement of energy bursts is described in more detail in Section 4.2.

After an energy burst was detected and its corresponding duration was determined, the duration has to be verified. If it is valid, i.e., there exists a mapping to a hex value, the corresponding value is saved for further processing, the energy burst will otherwise be ignored. Due to a low measurement granularity of the RSSI or because of interference from other radios, the durations will always vary by a small factor. Thus, to be more flexible in decoding energy bursts, a duration is also valid if it is within a specified range around a defined value, i.e., the defined duration $\pm\ \epsilon$, where ϵ depends on the used mapping.

Since one energy burst is only encoding four bits of information, i.e., a hex value, a receiver has to be able to reassemble more energy bursts together for reconstructing larger messages. Therefore, a receiver waits for a specified time after receiving an energy burst. If no other burst is received within the defined time, the receiver assumes that the transmission has ended and starts reconstructing the original message. Otherwise, each following energy burst that is detected within the specified time, is treated as one message and, therefore, saved for further processing.

Since only whole bytes are transmitted, represented by two successive energy bursts, the information of two bursts is always merged together and the original byte is reconstructed. Afterwards, all reconstructed bytes have to be interpreted accordingly to reassemble the original message.

With this approach, a receiver will try to decode every duration that is detectable on the configured channel. To counteract this problem, a preamble is used that is sent before every CTC related message. Therefore, a receiver can filter out CTC related messages among the usual communications caused by other devices in its proximity. The receiver only listens for the defined preamble and only starts decoding the following durations if the preamble was detected. The preamble and the format of a CTC message is described in more detail in Section 3.2.4.

3.2.4 Structure of CTC Messages

Since each energy burst is representing one hex value, four bit of information is encoded in each burst. Therefore, every data exchange is measured in symbols, which represents those four bits. Figure 3.2 shows the format of a CTC message.

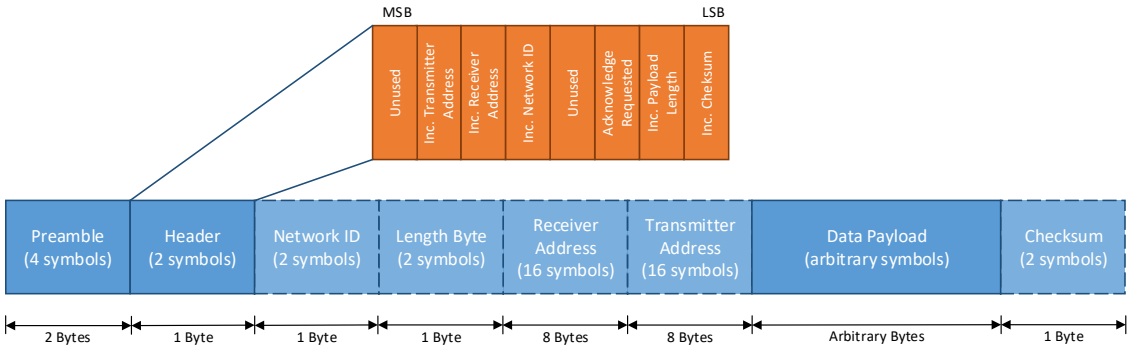


Figure 3.2: Format of CTC messages.

As can be seen in Figure 3.2, some fields are brighter and have dashed lines. Those are optional, i.e., they are not absolutely necessary for a communication. The optional fields only provide more functionality and allow a better customization of a CTC message. Each field is described in more detail below.

Preamble
The preamble is used to identify the beginning of a CTC message, i.e., to detect the transition between background noise and a valid CTC message. The preamble is four symbols long, i.e., four energy bursts, and is defined as 0x0101. After a preamble was detected, all following energy bursts are treated as CTC packets and processed accordingly until a timeout occurs, i.e., the time between two successive bursts is longer than a specified duration.

Header
The header is two symbols long and used to specify which additional data, i.e., the fields that are brighter and have dashed lines in Figure 3.2, are attached to the CTC message. Therefore, a better customization of a CTC message is possible.
Table 3.1 shows each bit of the header and describes its function briefly.

Bit	Part	Description
7	Unused	-
6	Transmitter Address	If set, the address of the transmitter is attached. Needed for acknowledgments.
5	Receiver Address	If set, the address of the receiver is attached. The message will be only accepted by the specified receiver. Otherwise, the message will be sent as broadcast.
4	Network ID	If set, the network ID is attached. Only receivers with the corresponding network ID will accept the message.
3	Unused	-
2	Acknowledge Requested[1]	If set, the receiver has to send back an acknowledgment to the transmitter.
1	Payload Length	If set, the payload length is attached. Hence, detecting noise that was falsely attached to the message is possible.
0	Checksum	If set, a checksum is attached. Used for detecting transmission errors.

Table 3.1: Header of a CTC message.

To keep the transmission time of the header as short as possible, the bits were chosen in a specific way. Bits which are representing optional fields that are used to be set more frequently have a lower significance in the hex representation of the header byte. In particular, setting only the bits representing the checksum and the network ID, the value of the header byte will be 0x11. Hence, only two short energy burst with a duration representing the hex value 0x1 each, have to be transmitted.
Therefore, the time needed for transmitting the header is kept as short as possible, since the durations are getting longer the higher the value of the transmitted hex value will be. As can be seen in Table 3.1, two bits are still unused. These bits can be used for future improvements.

Network ID
The network ID consists of two symbols and is used to distinguish between different networks. This is needed when different technologies are used to build a bigger group or network and the data that is exchanged between these devices should not leave the group, i.e., should not be received from devices outside the network. If a device receives a CTC message with a different network ID, the message will be discarded. One concrete example would be synchronization of the clock of heterogeneous devices where the devices are separated into different groups (networks) for measuring different values.

[1] Not used in the current implementation, reserved for future development of *X-Burst*

Length Byte
The length byte is two symbols long and can be used to detect falsely attached data to a CTC message. It could happen that the receiver still detects and decodes energy bursts after the actual message was already received. Due to other transmissions, energy bursts from other communications could be misinterpreted. With the included length byte, the receiver knows exactly of how many bytes the original message consists and can truncate the falsely attached noise from the received data.

Receiver Address
The receiver address specifies the receiver of a CTC message. Any other device receiving the message will discard it. Hence, it is possible to communicate with a single device, regardless of its technology. If no receiver address is specified, the message will be sent as broadcast, i.e., to every device in range. For the receiver address, the 64-bit EUI IPv6 address of the device is used, which is represented by 16 symbols.

Transmitter Address
The transmitter address is used to inform the receiver of a CTC message about the sender of the message. This allows a receiver to send back a response directly to the transmitter. Therefore, a data exchange between two specific devices is possible. The address is also needed to send back an acknowledgment to the transmitter if the *acknowledge requested* bit was set in the header of the message. For the transmitter address, also the 64-bit EUI IPv6 address of the device is used, which is represented by 16 symbols.

Data Payload
The payload can be of arbitrary size and contains the actual data.

Checksum
The checksum consists of two symbols and can be used to detect transmission errors of the received data. It is calculated by simply adding up each byte of the payload modulo 255 (the checksum is only 1 byte, hence, only values up to 255 can be represented). The result will then be used as checksum. If a receiver receives a message with a checksum attached, it calculates the checksum of the payload by its own and compares the computed value with the received one. If they do not match, the message will be discarded.

Chapter 4

Design Challenges

This chapter describes the challenges that need to be addressed when developing *X-Burst*. The generation of energy bursts for both wireless technologies used in this thesis, i.e., ZigBee and BLE, are described in Section 4.1. How to determine the durations of energy bursts in a reliable fashion is described in detail in Section 4.2. Section 4.3 describes the aspects that have to be considered for a seamless integration of *X-Burst* into an existing operating system.

4.1 Generation of Energy Bursts

In this section, the generation of precisely timed energy bursts using ZigBee and BLE is described in more detail. As defined in Section 3.1.2, only standard compliant functions can be used. Furthermore, different mappings from hex values to durations, i.e., one for each technology, are unsuitable. Hence, a common mapping among all technologies has to be found. Therefore, it is always possible for a receiver to decode a received CTC message, independent of the technology used by the transmitter. The durations have to be chosen in such a way that they can be generated and detected by each supported technology.

4.1.1 ZigBee

To generate the various energy bursts, while fulfilling the requirements defined in Section 3.1.2, only standard data packets of ZigBee are used. The structure of a ZigBee PHY frame can be seen in Figure 4.1.

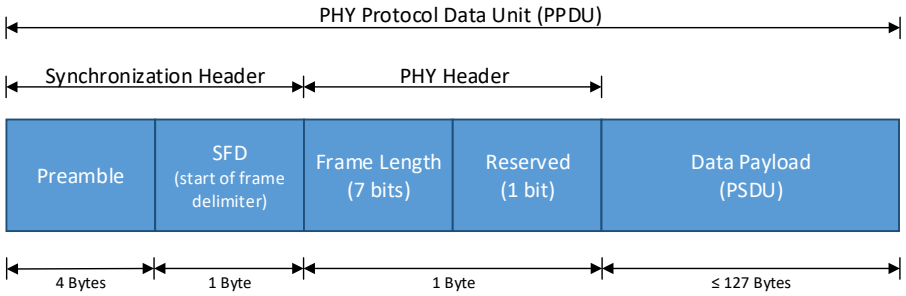


Figure 4.1: Format of a ZigBee PHY frame.

Besides the variable payload, additional values are attached for each ZigBee packet. First, the preamble and the start of frame delimiter (SFD) are sent, which are needed for synchronization and detection of ZigBee packets. Then, the PHY header, including the length of the entire frame is transmitted. At the end, a variable payload containing the actual data is sent.
The different energy burst durations are generated by only changing the amount of bytes in the payload of ZigBee's PHY packets. The payload can be varied between 0 and 127 bytes. Since the synchronization and PHY header are fixed and always transmitted, the minimum amount of data which has to be sent using standard ZigBee packets is 6 bytes.

ZigBee has a data rate of 250 kbit/s. Hence, the theoretical time for transmitting one bit can be calculated as follows:

$$\delta_Z = \frac{1}{250000} = 4 * 10^{-6}$$

Therefore, transmitting one bit of information takes 4 microseconds.

The different energy burst durations, by varying the payload in ZigBee's PHY packets, can be calculated as follows:

$$(6 + p) * 8 * \delta_Z = d$$

where 6 is the minimum amount of bytes of overhead in a ZigBee PHY packet, p is the payload in bytes, 8 is used to transform the bytes into bits, δ_Z is the time needed to transmit one bit in microseconds and d is the total time needed to transmit the entire packet, i.e., the duration of the generated energy burst, in microseconds.

Therefore, using $\delta_Z = 4\ \mu s$, the minimum and maximum achievable durations using ZigBee's standard PHY packets are:

$$min : (6 + 0) * 8 * 4 = 192\ \mu s$$
$$max : (6 + 127) * 8 * 4 = 4256\ \mu s$$

Hence, energy bursts with durations between 192 μs and 4256 μs can be generated.

Table 4.1 shows a possible mapping from hex values to energy burst durations and the corresponding amount of payload bytes for ZigBee. This mapping achieves the highest data rate by using only standard PHY packets of ZigBee, as the different durations are represented by changing the payload of a packet by only one byte.
To be able to use this mapping, the receiver has to be able to distinguish between the different durations properly, i.e., has to offer a high sampling rate of the RSSI of a channel. In particular, a sampling rate of at least 15 μs is required to distinguish properly between the durations shown in Table 4.1. Besides the sampling rate, the kind of how the RSSI is measured is also an important point when defining the durations of *X-Burst*, which is discussed in Section 4.2.

Hex value	Energy burst duration	Payload bytes
0x0	192 μs	0
0x1	224 μs	1
0x2	256 μs	2
0x3	288 μs	3
0x4	320 μs	4
0x5	352 μs	5
0x6	384 μs	6
0x7	416 μs	7
0x8	448 μs	8
0x9	480 μs	9
0xA	512 μs	10
0xB	544 μs	11
0xC	576 μs	12
0xD	608 μs	13
0xE	640 μs	14
0xF	672 μs	15

Table 4.1: Amount of payload bytes for generating energy bursts to achieve the best possible throughput for ZigBee.

An alternative to the use of standard PHY packets, for generating the various energy bursts, would be the use of specific test modes of ZigBee. Some radios, e.g., the TI CC2420 [17], offer the possibility to send unmodulated data packets for a specified amount of time. Hence, the overhead of transmitting the synchronization and PHY header could be reduced and a faster data rate could be achieved. Since not every radio offers such test modes, the standard PHY packets of ZigBee were used for *X-Burst*.

4.1.2 Bluetooth Low Energy

Generating the various energy bursts, while fulfilling the requirements defined in Section 3.1.2, is a little bit more difficult when BLE is used. BLE has two different modes of communication: (i) a connectionless communication for broadcasting and advertising (using advertisement packets) and (ii) a connection-oriented one, for data exchange between two devices, using data packets. Both types are using the same link layer packet format, where only the PDU is slightly different. The format of a BLE link layer packet can be seen in Figure 4.2.

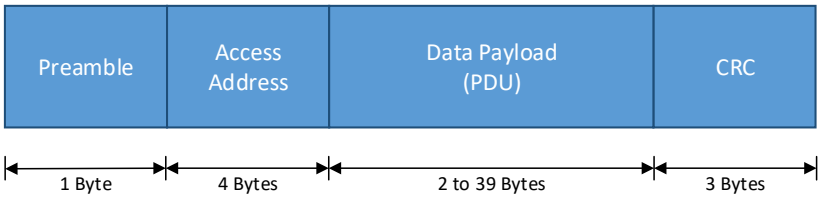


Figure 4.2: Format of BLE link layer packets.

Besides the variable payload, additional values are attached for each BLE data packet. The preamble is transmitted first, which is used for synchronization and detection of BLE packets. Next, the access address is sent, which is different for each link layer connection between two devices. Then, the variable payload, containing the actual data, is transmitted. At the end, a 24-bit CRC value for detecting transmission errors is sent.
The different energy burst durations are generated by only changing the amount of bytes in the payload of BLE's link layer packets. The payload can be varied between 2 and 39 bytes, i.e., a PDU header is always included within the payload. Because of the fixed preamble, access address, CRC value and the two payload bytes, the minimum amount of data that has to be sent using BLE link layer packets is 10 bytes.

BLE has a data rate of 1 Mbit/s. Hence, the theoretical time for transmitting one bit can be calculated as follows:

$$\delta_{BLE} = \frac{1}{1000000} = 1 * 10^{-6}$$

Therefore, transmitting one bit of information takes 1 microsecond.

The different energy burst durations, by varying the payload of BLE's link layer packets, can be calculated as follows:

$$(8 + p) * 8 * \delta_{BLE} = d$$

where 8 is the minimal amount of bytes of overhead in a BLE link layer packet (excluding the two payload bytes), p is the payload in bytes, 8 is used to transform the bytes into bits, δ_{BLE} is the time needed to transmit one bit in microseconds and d is the total time needed to transmit the entire packet, i.e., the duration of the generated energy burst, in microseconds.

Therefore, using $\delta_{BLE} = 1$ μs, the minimum and maximum achievable durations using standard BLE link layer packets are:

$$min : (8 + 2) * 8 * 1 = 80\ \mu s$$
$$max : (8 + 39) * 8 * 1 = 376\ \mu s$$

Hence, energy bursts with durations between 80 μs and 376 μs can be generated.

This is a major problem, as the fastest possible mapping for ZigBee requires energy burst durations up to 672 μs. Another problem is, that, according to the BLE specification, data packets can only be sent during an already established connection between two BLE devices. Thus, to use CTC, while fulfilling the requirements defined in Section 3.1.2, only advertising packets can be used. Since those packets can only be sent on specific advertisement channels, the CTC scheme will be very restricted. Additionally, also the robustness of the CTC will be lower since those channels are heavily used by other BLE devices.
Because of the the restriction to the three advertisement channels and the fact that only very short energy bursts can be generated, a communication with ZigBee devices using standard BLE link layer packets is not feasible.

To solve the mentioned issues, BLE test packets are used instead of link layer packets. Since those packets are specified in the BLE specification [18], every standard-compliant BLE device will be able to send those packets. The format of a BLE test packet can be seen in Figure 4.3.

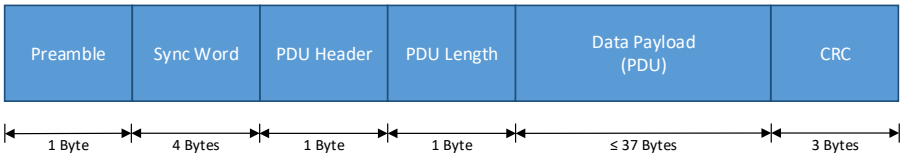


Figure 4.3: Format of BLE test packets.

The format is not much different to BLE link layer packets. It also consists of a preamble and a CRC value. Instead of an access address, a synchronization word is sent. Additionally, the PDU header is separated from the payload and a PDU length is sent. Nevertheless, the variable amount of payload bytes is also restricted to 37 bytes. Furthermore, the minimum amount of bytes that can be sent using BLE test packets is the same as for BLE link layer packets, i.e., 10 bytes. Therefore, the same minimum and maximum energy burst durations, can be achieved. The big advantage of BLE test packets is, that according to the BLE specification, the test packets are not restricted to specific channels. Hence, test packets can be sent on any BLE channel. Additionally, it is possible to send multiple test packets consecutively without any delay. Thus, any durations can be achieved, starting from a minimum of 80 microseconds. Some radios, e.g., the TI CC2650, support BLE test packets with a payload up to 255 bytes. Hence, energy bursts with a duration up to 2120 μs can be generated by only sending one BLE test packet.

The following table shows the mapping from hex values to energy burst durations and the corresponding amount of payload bytes for BLE. This mapping can be used to communicate with ZigBee devices that are using the mapping from Table 4.1. The only requirement is to have devices that can distinguish between the energy bursts properly, i.e., measuring the RSSI at least every 15 μs.

Hex value	Energy burst duration	Payload bytes
0x0	192 μs	1*14 (14)
0x1	224 μs	1*18 (18)
0x2	256 μs	1*22 (22)
0x3	288 μs	1*26 (26)
0x4	320 μs	1*30 (30)
0x5	352 μs	1*34 (34)
0x6	384 μs	2*19 (38)
0x7	416 μs	2*21 (42)
0x8	448 μs	2*23 (46)
0x9	480 μs	2*25 (50)
0xA	512 μs	2*27 (54)
0xB	544 μs	2*29 (58)
0xC	576 μs	2*31 (62)
0xD	608 μs	2*33 (66)
0xE	640 μs	2*35 (70)
0xF	672 μs	2*37 (74)

Table 4.2: Amount of payload bytes for generating energy bursts using BLE that are compatible with the ZigBee mapping shown in Table 4.1.

To generate the needed durations, more than one test packet has to be sent. In Table 4.2, the amount of needed test packets and their length is shown, e.g., 2*37 means that 2 test packets, with 37 bytes each, has to be sent successively. The values in brackets are the total amount of bytes that have to be sent to generate the corresponding duration of an energy burst. Since some radios support more payload bytes, these values can be used to send only one larger test packet.

For this thesis, BLE in the version 4.1 was used. Since BLE version 5.0 is now getting more widespread, using this version would simplify the implementation of *X-Burst*. In the new version, the maximum amount of payload bytes in both the link layer and the test packets is increased to 255 bytes. Therefore, energy bursts with durations between 80 and 2120 microseconds can be generated by only sending one packet. Another advantage is the *extended advertising*, which uses the *secondary advertising channels*. This allows advertising on any channel, i.e., also on data channels. Hence, the energy burst generation can be done by only using one advertising packet, which is not restricted to the three advertisement channels.

4.2 Measuring the Duration of Energy Bursts

In this section, the detection of energy bursts and especially the correct measurement of their durations are discussed. Detection of an energy burst is easy, as it can simple be detected by checking if the RSSI is above a specified threshold. The challenging part is to determine the exact duration of an energy burst. Towards this goal, we distinguish between an instantaneous and a non-instantaneous measurement of the received signal strength of a specified channel.

4.2.1 Instantaneous RSSI Measurement

In an instantaneous RSSI measurement, the obtained value actually represents the current received signal strength in the specified channel, as shown in Figure 4.4. The top figure shows the time in which the packet is actually sent on air. The bottom figure shows that the transition between -107 dBm and -45 dBm happens instantaneously, i.e., within two consecutive RSSI measurements.

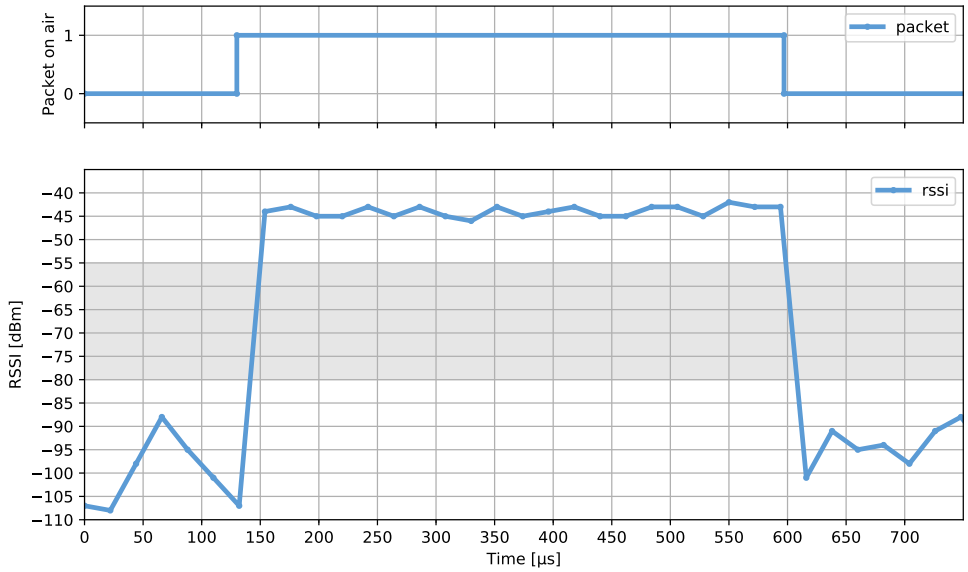


Figure 4.4: Instantaneous measurement of the received signal strength.

As can be seen in Figure 4.4, it takes indeed only one measurement to determine the signal strength of the currently sent packet. Hence, measuring the duration is as simple as the detection of a burst. Defining a threshold for the RSSI is enough to measure the duration of an energy burst. If the threshold is exceeded, the measurement is started and when the indicator falls back below the threshold, the measurement is stopped.
To obtain Figure 4.4, a TI CC2650 LaunchPad in BLE mode was used, which frequently measures the received signal strength of the specified channel while a packet was sent. In

particular, the values were measured at a rate of 45 kHz, i.e., every 22 μs. If the threshold is chosen to be in the gray area, the measurement of the duration will be sufficient. The measurement will be triggered by the first value above the threshold, i.e., by the value at 154 μs. After the measurement is started, it will be stopped by the first value below the threshold, i.e., by the value at 616 μs. Since the detection of the beginning and the end of an energy burst are both delayed by one measurement (in the case of the TI CC2650 one measurement takes about 22 μs), the measured duration corresponds to the real duration of the detected energy burst. Hence, the duration of an energy burst can be determined by only specifying a threshold for the RSSI when an instantaneous measurement is used.

4.2.2 Non-Instantaneous RSSI Measurement

Certain radios measure the RSSI as an average over the x bit symbols [17]. This is the case for the TI CC2420 radio [17] and for the TI CC2650 in IEEE mode, where the RSSI returned by the radio actually represents the average of the last seven symbols as can be seen in Figure 4.5: it takes a certain amount of time (the gray areas in Figure 4.5) to reach -50 dBm and to go back to -100 dBm.

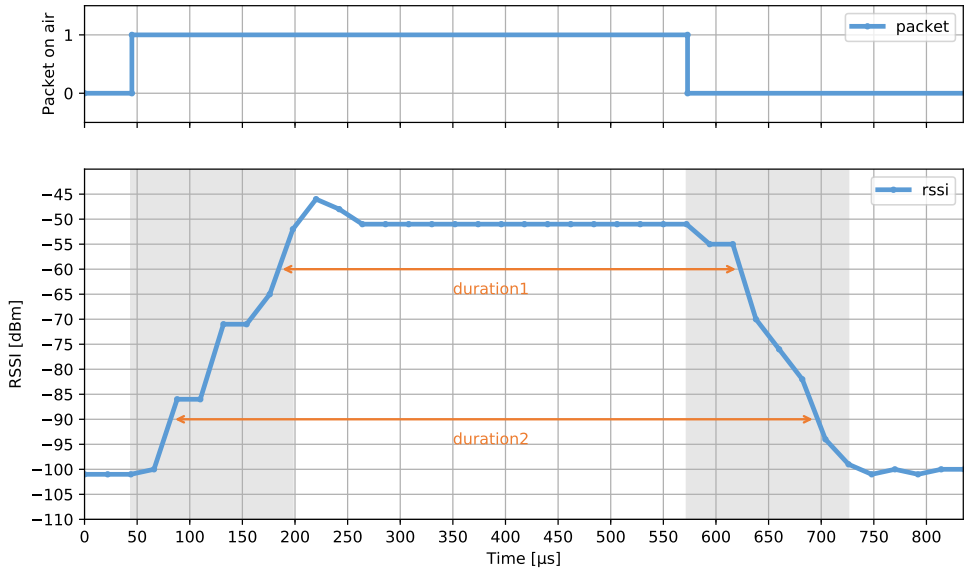


Figure 4.5: Non-instantaneous measurement of the received signal strength.

Using the same approach as for an instantaneous measurement of the received signal strength, i.e., specifying a threshold for the RSSI to determine the real duration of an energy burst, would lead to a major problem when making use of non-instantaneous RSSI measurements. The measured duration of an energy burst strongly depends on the used threshold and the received signal strength of the detected energy burst. In particular, if the

threshold is specified too high, the measured duration will be always too short compared to the real duration of the energy burst, e.g., *duration1* in Figure 4.5. If the threshold is specified too low, the measured duration will be always too long, e.g., *duration2* in Figure 4.5. As one measurement takes about 22 μs, the measured duration will be about 154 μs too long or too short in the worst case. Hence, the duration cannot be measured accurately by simply specifying a threshold for the RSSI.

To solve this problem, the energy burst durations could be defined with a separation of at least 308 μs between each other. Additionally, every duration has to be accepted as a valid duration in the range of $\pm$ 154 μs by a receiver (by setting $\epsilon = 154$). However, this approach would significantly decrease the possible throughput of *X-Burst*.
Another approach would be to measure the real duration of an energy burst with a sufficient granularity. The rise and fall time, i.e., the areas marked in gray in Figure 4.5, of the RSSI have always the same length if the detected energy burst has a duration of at least 154 μs, i.e., seven measurements. Therefore, the measurements of the RSSI over time will be symmetrical which can bee seen in Figure 4.6. Using this characteristic, the real duration of an energy burst can be determined with a sufficient granularity by measuring multiple durations of the same burst. In particular, multiple thresholds are defined and a duration is measured for each threshold. The *average* of the measured durations corresponds to the real duration of a detected energy burst.

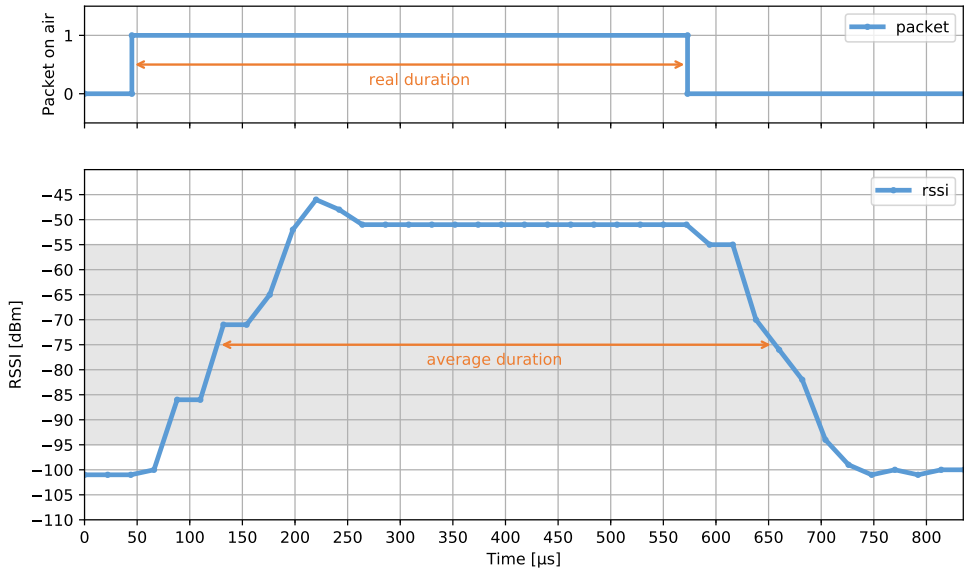


Figure 4.6: Determining the duration of an energy burst when a non-instantaneous measurement of the received signal strength is used.

As can be seen in Figure 4.6, the *real duration* of an energy burst is represented by the *average duration* of a detected energy burst. Hence, multiple thresholds are specified, e.g., every 5 dBm within the gray area shown in Figure 4.6. For each threshold, a corresponding measurement is started when the threshold is exceeded and stopped when the indicator falls back below the threshold. Therefore, multiple durations for one energy burst are measured and the *average duration* can be calculated. With this approach, the real duration of an energy burst can be determined sufficiently enough without affecting the throughput of *X-Burst*. The only requirement is that the shortest energy burst duration is at least longer than 154 μs, i.e., seven measurements.

4.3 Integration into an Existing Operating System

In this section, the aspects that have to be considered for a seamless integration into an existing operating system are described in detail. Furthermore, Radio Duty Cycling (RDC), a mechanism usually used by IoT devices for saving energy, is introduced and explained briefly. Additionally, the changes in the integration of *X-Burst* when a RDC mechanism is used or not are discussed in Section 4.3.1 and Section 4.3.2, respectively. We then show in Section 4.3.3 that, for a seamless integration, some specific configurations are needed.

As discussed in Section 3.1.2, the normal communication flow of the operating system should not be violated. Without a proper management between the CTC-related operations and the usual communications of the operating system, i.e., data exchange among other devices, the radio accesses of those two would be arbitrary. As a consequence, both will interfere each other and in fact, communications will not work properly anymore.
To address this problem, a smart management that is responsible for scheduling the accesses of the radio between the operating system and *X-Burst* has to be developed in an unobtrusive way. Ideally, the behavior of the operating system with respect to RDC has to be learned and *X-Burst* has to adapt to it accordingly.
Since the radio consumes typically the most energy of an IoT device, RDC is usually used for saving energy and thus, increasing the lifetime of a device. RDC is a mechanism which allows devices to turn off the radio as much as possible, while still being able to receive and transmit messages. Figure 4.7 shows the principle of duty cycling.

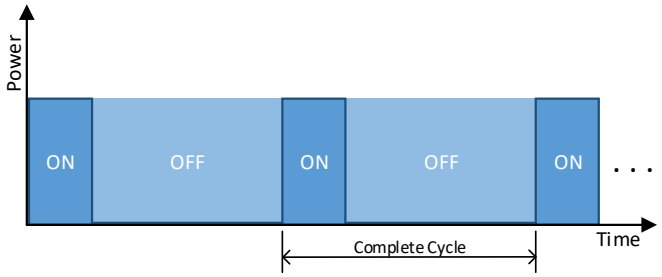


Figure 4.7: Principle of RDC.

As can be seen in Figure 4.7, a duty cycle of 25 % is created, meaning, only a component, i.e., the radio, or even the whole device, is turned on for only 25 % of the time. During the remaining 75 % of the time the component or even the whole device is turned off for saving energy.

For a seamless integration, *X-Burst* has to know whether RDC is used or not. Depending on that, further informations are needed, e.g., the duration of one cycle. In the following, the differences of integrating *X-Burst* in an operating system when RDC is used and when it is not used are explained.

4.3.1 With Radio Duty Cycling

When RDC is used, *X-Burst* has do adapt to it, i.e., it has to learn the used radio duty cycle and access the radio only during the *off-phases* i.e., during the time in which the radio would be normally turned off or in sleep mode. Thus, it is guaranteed that no interference between the usual transmissions and the cross-technology related ones occurs. Furthermore, the normal communication flow will not be violated, as the CTC operations are only running in the *off-phases*. A schematic representation of the adaptation of *X-Burst* to an operating system that is using RDC can be seen in Figure 4.8. The *OS* time slots in the following figures are representing the usual communications of the operating system.

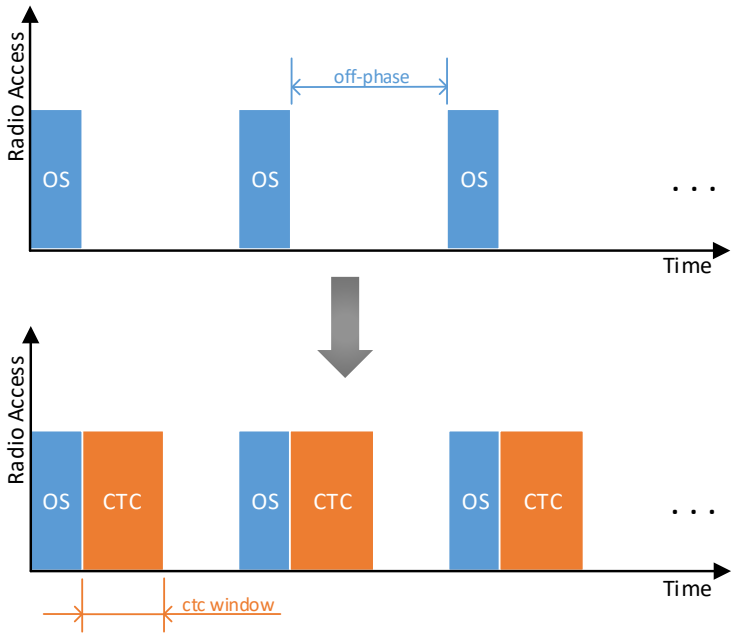


Figure 4.8: Adaptation of *X-Burst* to an operating system which is using RDC.

As can be seen in Figure 4.8, specific time slots for the CTC related operations are defined and only assigned within the *off-phases* of the duty cycle. To guarantee an unobtrusive integration, the maximum duration of a CTC time slot has to be shorter than the

duration of the *off-phase*.
The only drawback of this approach is that it will increase the energy consumption of a device since the radio is turned on for a longer duration. To minimize the increase of the energy consumption, the CTC time slots has to be kept as short as possible and the time between two slots has to be maximized. Hence, a configuration of the adaptation is needed. In particular, by configuring the the *ctc window* parameter, the duration of a CTC time slot can be defined. The occurrence, i.e., the period, of the time slots can be configured by the *policy* which is explained in more detail in Section 4.3.3.

4.3.2 Without Radio Duty Cycling

In the case that RDC is not used, a schedule of the radio accesses between the operating system and the cross-technology communication has to be established. Therefore, different time slots for accessing the radio are defined. A schematic representation of the segmentation of the radio accesses can be seen in Figure 4.9.

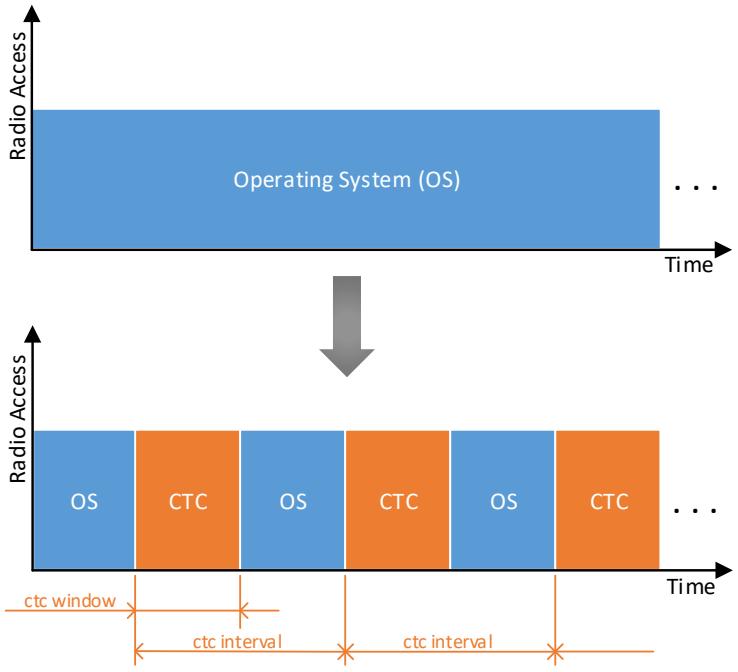


Figure 4.9: Adaptation of *X-Burst* to an operating system without RDC.

As can be seen in Figure 4.9, the usual communications and the CTC ones have well defined time slots to access the radio. Hence, interference between those are no longer possible.
Since only CTC-related operations can access the radio during a time slot assigned to CTC, the normal communication flow will be violated, as no other transmissions can be done in this time. Therefore, the CTC time slots has to be kept as short as possible and the time between two CTC slots has to be maximized. Thus, the segmentation of the

radio accesses has to be configurable. In particular, the *ctc window* defines the duration of the CTC time slots, and the *ctc interval* defines the occurrence of the CTC assigned time slots as it can be seen in Figure 4.9, and as detailed in Section 4.3.3.

4.3.3 Configuration

The required configuration options for a seamless and energy efficient integration of *X-Burst* into an existing operating system are shown below.

Enable / Disable CTC
To be very flexible and have full control over the impact on the operating system and the power consumption of a device, it is very important to be able to turn *X-Burst* on and off at any time. Hence, using these primitives, *X-Burst* should be turned off when not needed to save energy, and back on only when strictly necessary.

CTC Window
The *ctc window* is needed to define the duration of a CTC time slot (as can be seen in Figure 4.9 and 4.8). During this time, only CTC-related operations, such as sending a CTC message or measuring the RSSI of a channel to look after other CTC messages, can access the radio. Hence, a proper configuration of the *ctc window* is needed for a good and energy-efficient integration.

CTC Interval
When no RDC is used, the *ctc interval* defines the occurrence, i.e., the period, of the CTC related time slots. The purpose of the *ctc interval* can also be seen in Figure 4.9.

Policy
The *policy* is equivalent to the *ctc interval*, but is used for operating systems that use RDC. Hence, it also defines the occurrence of the CTC related time slots. Due to the adaptation of *X-Burst* to an existing duty cycle, a direct definition as with the *ctc interval* is not feasible. Instead, the *policy* defines which *off-phases* are really used. For a better understanding, an example of the *policy* is shown in Figure 4.10.

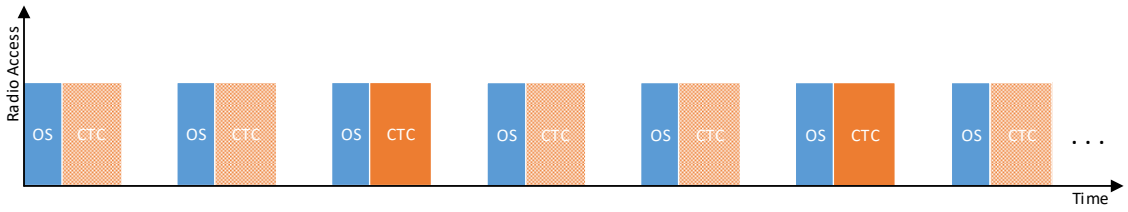


Figure 4.10: Adaptation of *X-Burst* with a *policy* of three.

In the example above, the *policy* was set to three, i.e., only each third *off-phase* is really used for sending or receiving CTC messages using *X-Burst*. Hence, the higher the *policy*, the longer will be the time between two CTC time slots.

Priority

When sending or receiving a bigger amount of CTC data, the *priority* needs to be changed. Usually, *X-Burst* adapts to the RDC mechanism used by the OS and will thus never have more time available as the one provided by the *off-phase* of the duty cycle. Hence, *X-Burst* has the lower *priority* by default. To be more flexible, however, the *priority* can be set to high: this will allow *X-Burst* to exceed the defined *ctc window* or even the *off-phase* of the duty cycle, so that larger CTC messages can be sent and more time can be used for scanning incoming CTC messages. For a better understanding, an example of how the *priority* parameter works is shown in Figure 4.11. It should be mentioned that this will violate the normal communication flow of the operating system.

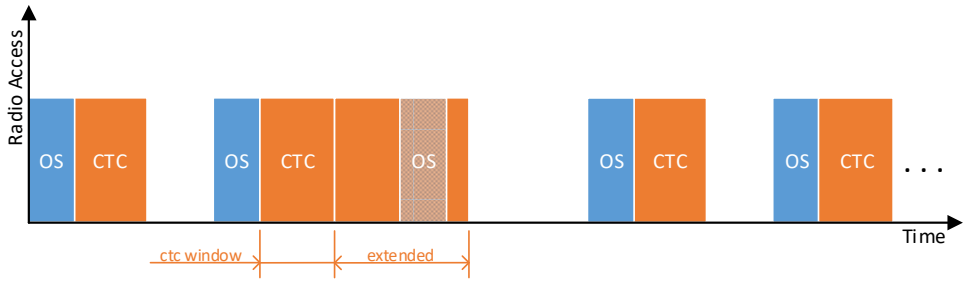


Figure 4.11: Changing the *priority* of *X-Burst*.

In the example above, the *priority* of *X-Burst* was set to high before the second time slot. Therefore, it was possible to exceed the assigned time slot and be able to send or receive a bigger amount of data. As a consequence, the time slot for the operating system could not be used. Afterwards the *priority* of *X-Burst* was set back to low.

The *priority* of *X-Burst* can also be changed if no RDC is used. The principle is exactly the same as the one in which the *priority* is set to high: *X-Burst* will exceed the assigned CTC time slot and will hence be able to send and receive larger messages.

Chapter 5

Integration into Contiki

This chapter describes the integration into the open source operating system Contiki. A better overview and explanation about the operating system is given in Section 5.1. The seamless integration into Contiki's network stack and the differences in the implementation when using ZigBee or BLE radios are shown in Section 5.2. For the seamless integration and for managing the radio accesses between Contiki and *X-Burst*, a special radio driver was developed. Latter is described in detail in Section 5.3. Furthermore, this sections details on the creation of an own schedule when no RDC is used, the adaptation to the ContikiMAC RDC mechanism [19] and to BLE connectionless and connection-oriented communications are shown.

5.1 The Contiki Operating System

Contiki [20] is an open source operating system for the Internet of Things. It is designed for low-cost, low-power devices with very limited hardware resources. The OS is implemented in the C language and is available for a wide range of different platforms. The three primary ones are the TI CC2538, the TI Sensortag and the TI CC2650.
Due to its design, Contiki is highly memory efficient and thus it is even suitable for very constrained devices with only a few kilobytes of memory. It runs on systems with less than 10KB of Random Access Memory (RAM) and 30KB of Read Only Memory (ROM). Although Contiki is a very lightweight OS, it provides a full Internet Protocol (IP) network stack with support for standard IP protocols such as the User Datagram Protocol (UDP), the Transmission Control Protocol (TCP) and the Hypertext Transfer Protocol (HTTP). The IPv6 stack is fully certified under the IPv6 Ready Logo program. Contiki also provides low-power standards for IPv6 networking such as IPv6 over Low power Wireless Personal Area Networks (6LoWPAN), Routing Protocol for Low power and Lossy Networks (RPL) and the Constrained Application Protocol (CoAP).
Besides the IP network stack, Contiki additionally provides a tailored wireless network stack called Rime that only supports simple operations such as sending data (unicast and broadcast) or network flooding.
To let developers get a better overview about the power consumption of a device, Contiki has a built-in software-based power profiling tool called Energest. Hence, it is possible to

get a good estimation about the current power consumption of the system, without any hardware modifications,. Furthermore, it is also possible to get more insights in which portion of code or hardware the power is spent.

One of the key features of an operating system for networked embedded systems is to provide concurrency. For memory-constrained devices, operating systems are rather event-driven than multi-threaded because of the better memory efficiency (in multi-threaded systems every thread has its own stack which results in a very high memory consumption). Event-driven systems do not need any looking mechanisms since it is not possible to run multiple event-handler simultaneously. Unfortunately, it is not always easy to develop an application in an event-driven way.
To combine the advantages of an event-driven system with a multi-threaded system, Contiki uses a mechanism called protothreads. Thereby, an event-driven kernel is combined with preemptive threads to provide a sequential flow of control.

5.1.1 Network Stack

The network stack of Contiki consists of four layers: the radio layer, the RDC layer, the Media Access Control (MAC) layer and the network layer, as shown in Figure 5.1.

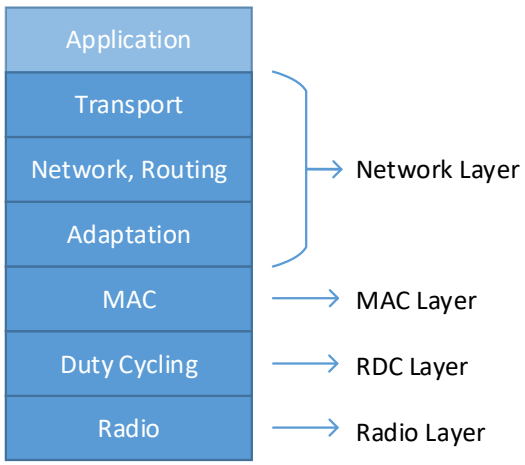


Figure 5.1: The Contiki network stack.

The **radio layer** is the lowest layer in Contiki's network stack. In this layer, the hardware dependent functionalities, such as sending and receiving data, for the corresponding radio are implemented.
The **RDC layer** is responsible for the duty cycling of the radio. It allows energy savings by keeping the radio off for most of the time. Contiki has four duty cycling mechanisms: ContikiMAC, X-MAC, Low-Power Probing (LPP) and nullRDC.
The **MAC layer** provides addressing and channel access control mechanism to allow a communication with neighbor devices or networks. In Contiki, the access control mechanism CSMA/CA is used.

The **network layer** is responsible for data adaptation (IPv6 and UDP header compression, data fragmentation), provides routing & network functionality as well as transport logic.

To fulfill the various requirements of different applications, Contiki's network stack can be easily configured individually depending on the application. Therefore, a special configuration file called *project-conf.h* exists for each application in which the network stack can be configured. This file is usually located in the root directory of an application.

5.1.2 BLEach

BLEach [21] is the first full fledged IPv6-over-BLE stack that was developed for the Contiki OS. Compared to all other available BLE stacks, BLEach is open source and allows modification of the BLE radio driver and the link-layer implementation. Due to its flexibility in terms of development, BLEach was used as BLE stack for this thesis.

Contiki was originally designed for IEEE 802.15.4. compatible devices. Therefore, also the network stack and especially the function of each layer are tailored to the behavior of IEEE 802.15.4. Nevertheless, BLEach was designed to work with the existing network stack without any modifications to Contiki. Only the usual behavior of each layer had to be changed to be able to use BLE. Figure 5.2 shows the architecture of BLEach and the corresponding layers in Contiki's IPv6-over-IEEE 802.15.4 stack.

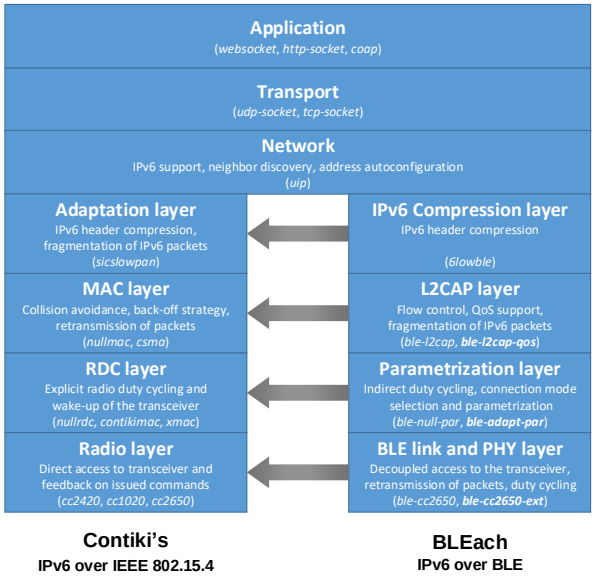


Figure 5.2: Architecture of BLEach and the corresponding layers in Contiki's IPv6-over-IEEE 802.15.4 stack. Adapted from [21].

5.2 Seamless Integration into Contiki's Network Stack

Integrating *X-Burst* into Contiki without modifying the structure of Contiki's network stack or changing existing implementations, while fulfilling the requirements from Section 3.1.2, was not an easy task. In the following, different approaches and their corresponding problems are explained.

The first approach was to integrate the scheme completely in the radio layer. This would require a lot of reprogramming and restructuring of an exiting radio implementation. Therefore, each implementation of supported radios has to be completely reprogrammed. Hence, porting the scheme to other hardware platforms would be very difficult and time consuming. Moreover, requirements regarding the portability and the integration into an operating system, as discussed in Section 3.1.2, would be violated.

The second approach was to implement a complete new stack besides the original one of Contiki. This would be a proper solution, as the CTC stack would have several layers responsible for different functionality. Additionally, no modifications of exiting implementations would be needed. Nevertheless, this approach has a major drawback. Having two simultaneously running stacks that access the same radio requires proper coordination. Without an entity manage the access to the radio, both stacks would interfere each other and in fact, nothing would work properly anymore. Hence, a coordination has to be implemented. Implementing it into the existing radio layer would result in the same problem as with the first approach. A proper solution would be to add an additional layer above the radio layer, which is shared among both stacks. The new layer would be responsible for managing the accesses of the radio for both stacks. However, this approach would violate the requirements since Contiki's network stack has to be modified.

The third and final approach was to create a *virtual radio*. Instead of including the actual radio driver, a virtual one is included. This driver is responsible for managing the coexistence of the usual communications and the CTC related ones, i.e., it schedules the accesses of the radio in an unobtrusive way between *X-Burst* and the operating system. A semantic representation of the final integration can be seen in Figure 5.3.

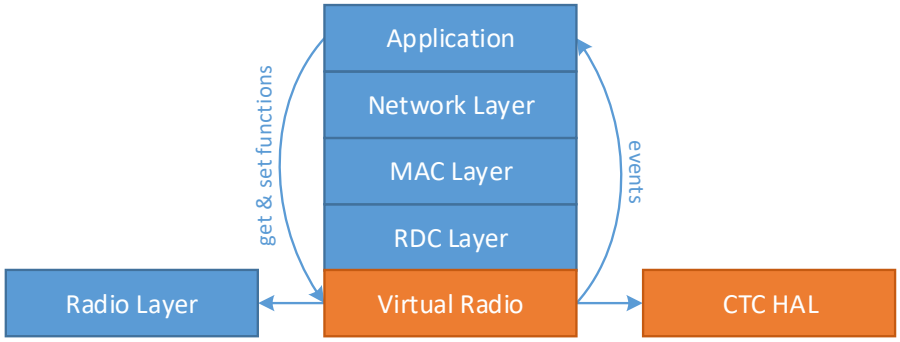


Figure 5.3: Seamless integration of *X-Burst* into Contiki's network stack.

As shown in Figure 5.3, the new *virtual radio* was included instead of the usual radio driver. It contains all the necessary functionality for *X-Burst*, except the hardware dependent implementations. Therefore, a separate layer, the Hardware Abstraction Layer (HAL), was created to hide the hardware dependent implementation details from the *virtual radio*. Hence, every supported hardware platform has only to provide a hardware specific implementation of the CTC-HAL. Thus, the *virtual radio* implementation is hardware independent and stays the same for each supported hardware platform. This increases the portability since only the CTC-HAL has do be implemented.
The already existing *get_value()*, *set_value()* and *set_object()* functions of the network stack are used for changing the configuration of the *virtual radio* from the application at runtime. Hence, no modifications of the operating system are needed. To notify the application about a received CTC message, the *virtual radio* will post an event which can be detected by the application. More information about the *virtual radio* and its working principle can be found in Section 5.3.

5.2.1 ZigBee

The following figure shows the integration of *X-Burst* when ZigBee is used. Furthermore, the configuration of each layer of the stack can be seen. In particular, the name of the corresponding file that is used for each layer is shown. As radio, the TI CC2650 was used.

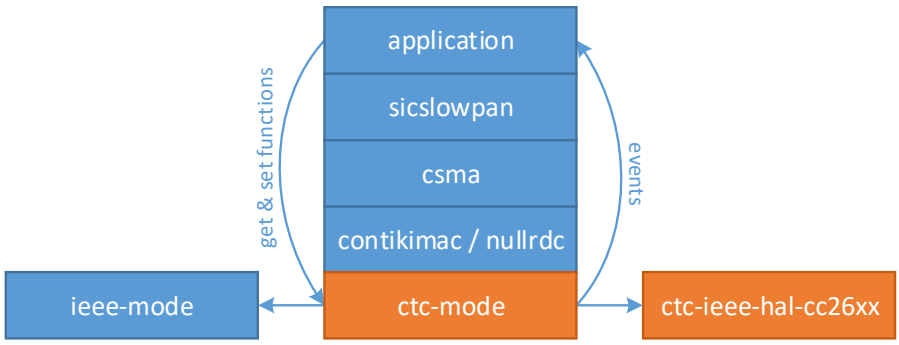


Figure 5.4: Seamless integration of *X-Burst* into Contiki's network stack when ZigBee is used.

5.2.2 Bluetooth Low Energy

The Integration of *X-Burst* when using a BLE radio is a bit different than the approach followed for ZigBee nodes as shown in Figure 5.5. Furthermore, the configuration of each layer of the stack can be seen. In particular, the name of the corresponding file that is used for each layer is shown. As radio, the TI CC2650 was used.

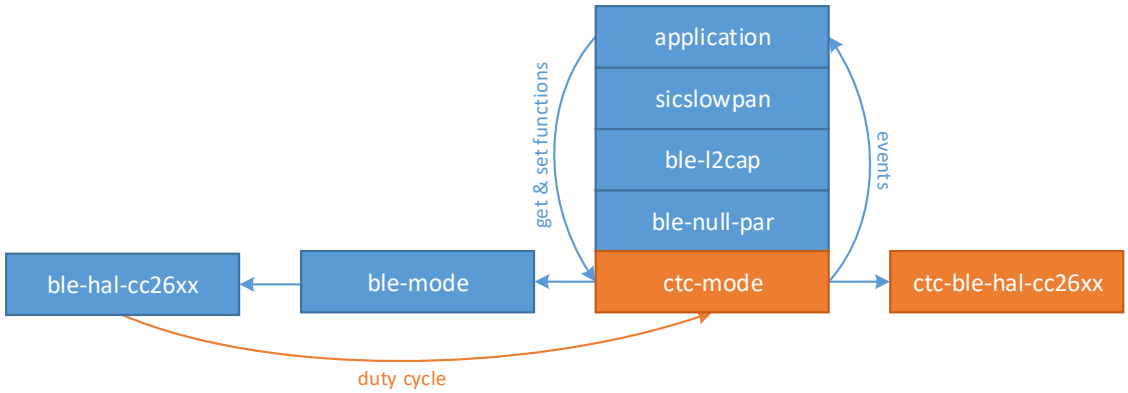


Figure 5.5: Seamless integration of *X-Burst* into Contiki's network stack when BLE is used.

The implementation of the BLE stack uses also a hardware abstraction layer, the BLE-HAL. Hence, the BLE radio (*ble-mode* in Figure 5.5) is hardware independent as all the hardware specific code is implemented in the BLE-HAL (*ble-hal-cc26xx* in Figure 5.5).
Another difference, compared to the integration with ZigBee, is that the radio duty cycling is done in the BLE radio layer. Usually, the duty cycling is done in the RDC layer of Contiki's network stack. Since the RDC layer is above the radio layer, the *virtual radio* knows when the radio is turned on and when not. This information is needed to guarantee a proper management between *X-Burst* and the operating system.
Since the BLE radio layer is below the *virtual radio*, it does not get any information about the state of the radio, i.e., if it is turned on or not. To solve this problem, the BLE-HAL informs the *virtual radio* about every change of the state of the radio. This is achieved by using the already existing *set_value()* function of the network stack, which sets specific values of the *virtual radio*. Hence, to use *X-Burst* with BLE, minimal changes of the hardware abstraction layer of BLE are needed.

5.3 The Contiki CTC Radio Driver

In this section, the working principle and implementation of the *virtual radio* are described. Section 5.3.1 describes the structure of the implementation. Furthermore, the content and requirements for implementing each file are discussed and their integration into Contiki's file structure is shown. In Section 5.3.2, the configuration of the *virtual radio* is explained in detail. The adaptation of *X-Burst* to different radio duty cycles is explained in Section 5.3.3. At the end, in Section 5.3.4, the actual implementation of the *virtual radio* is shown in detail. Moreover, how to receive and transmit CTC messages is explained in detail.

5.3.1 File Structure and Location Within Contiki

The implementation of the *virtual radio* is divided into different files, so that a better portability can be achieved. Therefore, all hardware specific implementations are shifted to a separate file which represents the CTC-HAL implementation of a specific hardware platform. In the following, all files that are needed for the CTC scheme are shown and their content is explained. Additionally, the integration into Contiki's file structure is shown.

Virtual Radio

For a seamless integration of *X-Burst*, a management of the accesses of the radio between the OS and *X-Burst* is needed. As defined in Section 3.1.2, the operating system must not be modified, i.e., also the structure of Contiki's network stack can not be altered. To integrate *X-Burst*, while fulfilling all the requirements, a new radio driver has to be developed for Contiki. In particular, a *virtual radio* was created. The *virtual radio* is actually not an implementation of a real radio: instead, this novel radio driver is managing the coexistence between the transmissions of the operating system and of *X-Burst*, i.e., it schedules the accesses of the radio in an unobtrusive way. To be able to include the *virtual radio* instead of the usual radio implementation, it has to provide a specific interface for the upper layers of Contiki's network stack. The interface, defined in the *radio.h* file, specifies 14 different functions that have to be provided by the *virtual radio*:

- *init*
  Initializes the radio.
- *prepare*
  Prepares the radio with a packet to be sent.
- *transmit*
  Sends the packet that has previously been prepared.
- *send*
  Prepares and transmits a packet.
- *read*
  Reads a received packet into a buffer.

- *channel_clear*
  Performs a CCA to find out if there is activity in the channel.
- *receiving_packet*
  Checks if the radio driver is currently receiving a packet.
- *pending_packet*
  Checks if the radio driver has just received a packet.
- *on*
  Turns the radio on.
- *off*
  Turns the radio off.
- *get_value*
  Gets a radio parameter value.
- *set_value*
  Sets a radio parameter value.
- *get_object*
  Gets a radio parameter object.
- *set_object*
  Sets a radio parameter object.

Most of the functions above will only forward the request to the actual radio driver. Nevertheless, implementing those functions within the *virtual radio* allows its inclusion instead of the actual radio driver. Hence, *X-Burst* can be used together with Contiki in a seamless way.

The implementation of the *virtual radio* is divided into two parts. The actual implementation is done in the *ctc-mode.c* file and all needed definitions are implemented in the *ctc.h* file. Both files are stored in a new folder (*ctc*) in the core directory of Contiki. The full path to the implementation of the *virtual radio* is: *Contiki\core\ctc*.

CTC-HAL

As already mentioned in Section 5.2, a new layer, the CTC hardware abstraction layer, was introduced to include contains all the hardware-specific implementations that are needed for the cross-technology communication. Therefore, supporting different hardware platforms is simplified since each supported device has only to provide a hardware-specific implementation of the CTC-HAL. The hardware abstraction layer is not a separate layer in Contiki's network stack, but is rather used by the *virtual radio* to communicate with the radio of the used hardware platform.

Each CTC-HAL implementation has to provide standardized functions in order to supply the *virtual radio* with the required hardware-specific implementations and enable cross-technology communication. The minimum functions that have to be provided are:

- *name*
  The name of the CTC-HAL implementation.
- *on*
  Turns the radio on.
- *off*
  Turns the radio off.
- *radio_accessible*
  Checks if the radio is currently in use. Since it is not possible that two processes run simultaneously in Contiki, the only possibility is that the radio is currently sending an automatic transmission of an acknowledgment frame. In the case that RDC is used, this check is not needed because *X-Burst* accesses the radio only during the usual *off-phases* of the radio.
- *set_channel*
  Sets the channel to the one used for transmitting and receiving CTC messages.
- *restore_channel*
  Sets the channel back to the one used for the usual communications of the operating system.
- *prepare_scanning*
  In the case that the radio has to be prepared to read the current RSSI of a channel.
- *get_rssi*
  Returns the current RSSI of the configured channel.
- *send_byte*
  Transmits one byte via CTC.

A CTC-HAL implementation of a specific hardware platform consists of two files: the c file, which contains the actual implementation, and a corresponding header file used for definitions, i.e., the amount of payload bytes needed for generating the required energy bursts. Within the Contiki file system, the implementation of the CTC-HAL has to be included in the corresponding directory of the used hardware.

Two versions of the CTC-HAL for the TI CC2650 Launchpad were implemented, i.e., one for ZigBee *(ctc-ieee-hal-cc26xx.c* and *ctc-ieee-hal-cc26xx.h)* and one for BLE (*ctc-ble-hal-cc26xx.c* and *ctc-ble-hal-cc26xx.h*). The files are stored in a new folder (*ctc-hal*) in the cc26xx-cc13xx directory. The full path to the CTC-HAL implementations of the TI CC2650 Launchpad is: *Contiki\cpu\cc26xx-cc13xx\rf-core\ctc-hal*.

NullCTC

In the case of including the *virtual radio* without defining a CTC-HAL implementation, an empty implementation, called nullCTC, will be included automatically. Hence, the program will compile and the non-CTC related part will still work. NullCTC only provides a framework of all required functions, but apart from that it will not do anything. It can be used as a template for new implementations or simply when a CTC-HAL implementation is not needed, e.g., for testing purposes.
The implementation is located in the same directory as the one of the *virtual radio*: *Contiki\core\ctc*.

Implemented Files and Location within Contiki's File System

Table 5.1 gives an overview about all files that were created for the implementation of *X-Burst*. Furthermore, the directory within Contiki's file structure of each file is shown and a short description about the content of each file is given.

File	Directory	Description
ctc.h	Contiki\core\ctc\	General definitions
ctc-mode.c	Contiki\core\ctc\	*Virtual radio* implementation
nullctc.c	Contiki\core\ctc\	CTC-HAL template
nullctc.h	Contiki\core\ctc\	CTC-HAL template
ctc-ieee-hal-cc26xx.c	Contiki\cpu\cc26xx-cc13xx\rf-core\ctc-hal\	CTC-HAL implementation
ctc-ieee-hal-cc26xx.h	Contiki\cpu\cc26xx-cc13xx\rf-core\ctc-hal\	CTC-HAL definitions
ctc-ble-hal-cc26xx.c	Contiki\cpu\cc26xx-cc13xx\rf-core\ctc-hal\	CTC-HAL implementation
ctc-ble-hal-cc26xx.h	Contiki\cpu\cc26xx-cc13xx\rf-core\ctc-hal\	CTC-HAL definitions

Table 5.1: Overview of all *X-Burst*-specific files and their locations.

5.3.2 Configuration

In this section, the configuration of the *virtual radio* is described in detail. To be able to use cross-technology communication, the network stack of Contiki has to be configured accordingly. This can simply be done within the *project-conf.h* file of the used application. An example of the configuration is shown in Figure 5.6.

```
/*---------------------------------------------------------------------------*/
#ifndef PROJECT_CONF_H_
#define PROJECT_CONF_H_
/*---------------------------------------------------------------------------*/
#define NETSTACK_CONF_RADIO         ctc_mode_driver
#define CTC_CONF_RADIO              ieee_mode_driver    | ble_cc2650_driver
#define CTC_CONF_HAL                ctc_ieee_hal_cc26xx | ctc_ble_hal_cc26xx
/*---------------------------------------------------------------------------*/
#endif /* PROJECT_CONF_H_ */
/*---------------------------------------------------------------------------*/
```

Figure 5.6: Configuration of the *virtual radio* in the *project-conf.h* file of an application.

In Figure 5.6, the radio layer, i.e., the used radio implementation, is configured by setting the *NETSTACK_CONF_RADIO* definition accordingly. In the example above, it is set to the *virtual radio* (*ctc_mode_driver*). Since the *virtual radio* has to know which radio and CTC-HAL implementation has to be used, both have to be defined accordingly. Hence, the usual radio layer is configured by the *CTC_CONF_RADIO* definition and the CTC-HAL is configured by the *CTC_CONF_HAL* definition. Two different configurations can be seen in the figure above. Both are for the TI CC2650 Launchpad and configure the *virtual radio* depending on which wireless technology, i.e., ZigBee (*ieee_mode_driver* and *ctc_ieee_hal_cc26xx*) or BLE (*ble_cc250_driver* and *ctc_ble_hal_cc26xx*), is used.

The following tables show the configuration options of the *virtual radio* implementation. Each configuration is defined in the *ctc.h* file and has to be adjusted accordingly before using the *virtual radio*.
Table 5.2 shows the common definitions of the *virtual radio* and Table 5.3 shows the required configurations for detecting and calculating the duration of an energy burst. These definitions must be configured to use *X-Burst*.
Additionally, the corresponding default value and a short description of each definition is given.

Name	Default	Description
CTC_ENABLED	1	Enables (1) or disables (0) the CTC scheme.
CTC_WINDOW	80	Duration of the *CTC window*, CTC related operations are only done within this window, in milliseconds.
CTC_WAITING_TIME	3	Time before the process of the *virtual radio* is started, in seconds.
CTC_AUTO_CONFIG	1	Enables (1) or disables (0) the *auto configuration*.
CTC_SEND_WHOLE_WINDOW	0	If enabled (1), a CTC message will be sent during the whole *CTC window*.
CTC_MAX_RETRANSMISSIONS	3	The maximum amount of retransmissions in the case that a message could not be sent.
CTC_DURATION_PLUS	5	Defines the upper bound for a valid duration.
CTC_DURATION_MINUS	4	Defines the lower bound for a valid duration.
CTC_RX_TIMEOUT	500	Timeout used to distinguish between not related energy bursts, in microseconds.
CTC_MIN_RX_TIME	30	The minimum amount of time that has to be left for starting the receiving procedure, in milliseconds.
CTC_PRIORITY	0	Defines the priority of the CTC operations: high (1) or low (0).
CTC_NETWORK_ID	-	Used to distinguish between devices of different networks.
CTC_MAX_PAYLOAD_LENGTH	120	Maximum payload that can be sent via CTC.
CTC_CHANNEL_IEEE	20	Radio channel used for CTC with ZigBee.
CTC_CHANNEL_BLE	22	Radio channel used for CTC with BLE.

Table 5.2: Configuration of the *virtual radio* - common values.

When using BLE in connection oriented mode and *auto configuration* is used, the *CTC_WAITING_TIME* has to be chosen accordingly to make sure that the connection has already been successfully established. Otherwise, the *virtual radio* will learn the duty cycle of the connectionless communication that is used to establish a connection between two BLE devices.

Name	Default	Description
CTC_RSSI_THRESHOLD_MIN	-80	Defines the threshold for detecting energy bursts, in decibel.
CTC_RSSI_THRESHOLD_MAX	-35	Defines the upper bound of the thresholds which are used for calculating the duration of an energy burst, in decibel.
CTC_RSSI_THRESHOLD_STEPS	5	The gap between the used thresholds for the duration measurement between the minimum and maximum threshold.
CTC_RSSI_MIN_DURATION	8	Every detected energy burst with a duration smaller than this value will be discarded, in *rtimer* ticks.
CTC_RSSI_DURATION_RANGE	10	Used for outlier detection. If a measured duration of a burst is not within the defined range to the previous measured duration, it will not be used for determining the average duration, in *rtimer* ticks.

Table 5.3: Configuration of the *virtual radio* - rx values.

Depending on the configuration of the *AUTO_CONFIG* definition, i.e., if *auto configuration* is used or not, further definitions have to be adjusted. If *auto configuration* it is enabled, the definitions of Table 5.4 have to be configured.

Name	Default	Description
AUTO_CONFIG_RADIO_OFF	0	If set (1), the radio will be turned off during the duty cycle measurement.
NUMBER_OF_DC	4	The minimum number of duty cycles that have at be measured.
DC_THRESHOLD	3	Measured duty cycles with a duration less than the threshold are ignored and not used for further calculations, in milliseconds.
DC_MEASURING_TIME	3	The duration of the duty cycle measurement, in seconds.

Table 5.4: Configuration of the *virtual radio* - auto configuration.

If the *auto configuration* is disabled, all definitions regarding the radio duty cycle have to be configured manually. Therefore, the definitions of Table 5.5 have to be adjusted.

Name	Default	Description
WITH_DUTY_CYCLE	1	Informs the *virtual radio* whether radio duty cycling is used or not.
WITH_MULTIPLE_ON_PERIODS	0	Defines the multiple on periods within a duty cycle.
DUTY_CYCLE	-	Duration of the used radio duty cycle, in milliseconds.

Table 5.5: Configuration of the *virtual radio* - manual configuration.

Depending whether a radio duty cycle is used or not, different definitions have to be configured additionally. Table 5.6 shows the needed configuration when no radio duty cycling is used.

Name	Default	Description
CTC_INTERVAL	1000	Duration between two CTC time slots, in milliseconds.

Table 5.6: Configuration of the *virtual radio* - without RDC.

In the case that radio duty cycling is used, the following definitions of Table 5.7 have to be adjusted.

Name	Default	Description
CTC_POLICY	2	Define the occurrence of the CTC time slots, e.g., 2 means that only each second *off-phase* will be used for CTC.
CTC_MIN_TIME	50	The minimum amount of time that has to be left for starting the process of the *virtual radio*, in milliseconds.

Table 5.7: Configuration of the *virtual radio* - with RDC.

To be able to change the configuration of the *virtual radio* during runtime, additional radio parameters are defined. Radio parameters in Contiki are defined as enumeration (enum), which is a user-defined data type that consists of integral constants. Hence, each parameter is represented by an integer that is automatically incremented if a new parameter is added. The standard parameters are represented by integers from 0 to 16 and the BLE specific parameters from 100 to 123. To avoid conflicts with existing parameters, all CTC related radio parameters are starting from 200. Thus, even new parameters can be added without any issues.

The following CTC-related radio parameters are defined to read or overwrite several of the described configurations above during runtime:

- RADIO_PARAM_CTC_ENABLE_DISABLE
- RADIO_PARAM_CTC_WINDOW
- RADIO_PARAM_CTC_SEND_WHOLE_WINDOW
- RADIO_PARAM_CTC_MAX_RETRANSMISSIONS
- RADIO_PARAM_CTC_RSSI_THRESHOLD
- RADIO_PARAM_CTC_RX_TIMEOUT
- RADIO_PARAM_CTC_PRIORITY
- RADIO_PARAM_CTC_NETWORK_ID
- RADIO_PARAM_CTC_RSSI_THRESHOLD_MIN
- RADIO_PARAM_CTC_RSSI_THRESHOLD_MAX
- RADIO_PARAM_CTC_RSSI_THRESHOLD_STEPS
- RADIO_PARAM_CTC_RSSI_MIN_DURATION
- RADIO_PARAM_CTC_RSSI_DURATION_RANGE
- RADIO_PARAM_CTC_INTERVAL
- RADIO_PARAM_CTC_POLICY

Besides the configuration of the *virtual radio*, some additional radio parameters are needed for *X-Burst*:

- RADIO_PARAM_CTC_TX_DATA
  When this parameter is set, the attached payload is copied to a buffer and a specific flag is set to inform the *virtual radio* that a message has to be transmitted via CTC. This parameter is only writable.
- RADIO_PARAM_CTC_HEADER
  Gets or sets the configuration of the header of the CTC messages.
- RADIO_PARAM_CTC_RECEIVER_LINK_ADDR
  Gets or sets the link address of the receiver of the CTC message. If no address is defined, the message is sent as broadcast to every device in range.
- RADIO_PARAM_CTC_BLE_ON
  Informs the *virtual radio* that the radio was turned on. Only needed when BLE is used. This parameter is only writable.

- RADIO_PARAM_CTC_BLE_OFF
  Informs the *virtual radio* that the radio was turned off. Only needed when BLE is used. This parameter is only writable.

The introduced radio parameters can be read or overwritten at anytime from the application. Therefore, the following (already existing) functions can be used:

- NETSTACK_RADIO.set_value(radio_parameter, value)
- NETSTACK_RADIO.get_value(radio_parameter, *value)
- NETSTACK_RADIO.set_object(radio_parameter, *dest, size)
- NETSTACK_RADIO.get_object(radio_parameter, *src, size)

Depending on the type of data that should be read or overwritten, different functions have to be used. With the *set_value* and *get_value* functions, only integer values can be read or overwritten. For each other data types, e.g., arrays and structs, the *get_object* and *set_object* functions have to be used.

5.3.3 Adaptation to the Duty Cycle

As discussed in Section 4.3, adaptation to an existing radio duty cycle is needed to guarantee a proper coexistence between *X-Burst* and the usual transmissions of the operating system. In the case that no RDC is used, a schedule between *X-Burst* and the OS has to be established. In particular, the adaptation to the RDC mechanisms nullRDC and ContikiMAC, when ZigBee is used, is shown. Furthermore, the adaptation to the duty cycle of BLE is discussed. Thereby, it is distinguished between connection-oriented and connectionless communications.

ZigBee

In the following, the adaptation of the *virtual radio* when using ZigBee is shown. Thereby, it is distinguished between using no radio duty cycling, i.e., using nullRDC, and using the default radio duty cycling mechanism of Contiki, i.e., ContikiMAC.

NullRDC is a RDC implementation of Contiki that works only as a pass-trough layer, i.e., it only transmits a packet and returns the result of such a transmission. Hence, no radio duty cycling is done. Therefore, the integration of *X-Burst* is exactly as described in Section 4.3.2. Hence, a schedule of the radio accesses between the OS and *X-Burst* have to be established.

To create the schedule of the OS and *X-Burst*, Contiki's *rtimer* is used. Therefore, a *rtimer* with the duration of the defined *CTC interval* is set at the start of a device. After the *rtimer* expires, the CTC process of the *virtual radio* is called and a new *rtimer* is set for the next time slot. Thus, a CTC time slot is guaranteed at each *CTC interval* amount of time. The duration of each time slot is defined by the *CTC window*.

ContikiMAC is the default RDC mechanism of Contiki. It uses periodical wake-ups to listen after transmissions from other devices. During these wake-ups, two inexpensive CCA checks are done consecutively, where the RSSI of the channel is measured. If the value is below a specified threshold, the CCA returns positive, i.e., the channel is free. Hence, the receiver can go back to sleep mode, i.e., turn off the radio. Otherwise, a transmission was detected and the receiver has to stay awake to be able to receive the full packet. For transmitting a packet, a sender repeatedly transmits the packet until an acknowledgment was received. In the case of a broadcast, where no acknowledgments are sent, the sender transmits the packet during the full duty cycle to ensure that all devices in its proximity have received it.

As discussed in Section 4.3.1, when RDC is used, *X-Burst* has to adapt to it, i.e., access the radio only in the *off-phases* of the duty cycle. Since ContikiMAC relies on precise timings between transmissions, an accurate adaptation of *X-Burst* is even more important. The working principle of ContikiMAC and the adaptation of *X-Burst* can be seen in Figure 5.7.

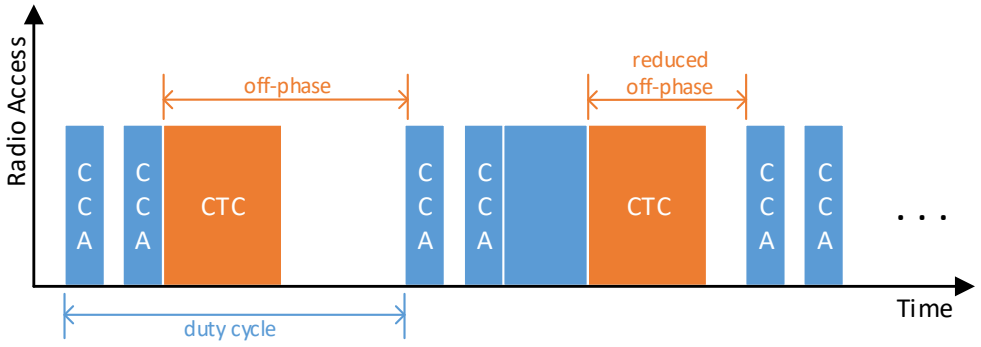


Figure 5.7: Adaptation of *X-Burst* to the ContikiMAC RDC mechanism.

As shown in Figure 5.7, in the first wake-up period, two CCA checks have not detected a transmission and thus, the device normally goes back to sleep mode by turning off its radio. Instead, with *X-Burst*, the radio is kept on and the CTC time slot starts. After a time, defined by the *CTC window*, the CTC time slot ends and the device goes back to sleep mode.
In the second wake-up period, the device has detected a transmission and thus, stays awake to receive the packet, which will shorten the duration of the *off-phase*. Since the usual transmissions have a higher *priority* than CTC-related ones, by default, it is possible that the the CTC time slot is shortened to fit within the remaining time of the current *off-phase*. Hence, not the full duration, as defined by the *CTC window*, will be available

for transmitting or receiving CTC messages during the current *off-phase*.
The maximum duration of a CTC time slot is limited by the maximum time in which the radio is switched off, i.e., until the next CCA checks are done. The amount of checks that are done within one second are defined by the *NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE*, which is set to 8 by default. This results in a duty cycle of about 125 ms. Using the TI CC2650, the two CCA checks, including the time in between, need about 3 ms. Hence, the maximum *off-phase* of the standard configuration of ContikiMAC (and, therefore, the maximum available time for a CTC time slot) is about 122 ms.

Bluetooth Low Energy

In the following, the adaptation of the *virtual radio* when using Bluetooth Low Energy is shown. It is distinguished between connection-oriented and connectionless communications. When a BLE device changes from a connectionless to a connection-oriented communication or vice-versa, the *virtual radio* will not detect it, i.e., it will not adapt to the changed behavior of the radio duty cycle automatically. Since the configuration of the *virtual radio* can be changed at any time, the application is responsible to inform the *virtual radio* of the changed behavior and modify it accordingly.

A **connectionless communication** is the easiest way to exchange data using BLE. It is mostly used for advertising a device's presence to other devices in range or to setup a connection between two BLE nodes. At the beginning of each advertising event (a periodical event defined by the *advertising interval*), an advertiser broadcasts its advertising packets to every device in its surrounding. These packets are sent subsequently on all three advertising channels (37, 38, 39).
Devices that are listening for advertising packets are so called scanners. At the beginning of each scanning event (a periodical event specified by the *scanning interval*), a scanner listens for a time,defined by the *scan window* for advertising packets. Depending on the configuration of a scanner, i.e., active or passive, a scan request is sent or not respectively. The principle of a BLE connectionless communication the adaptation of *X-Burst* can be seen in Figure 5.8.

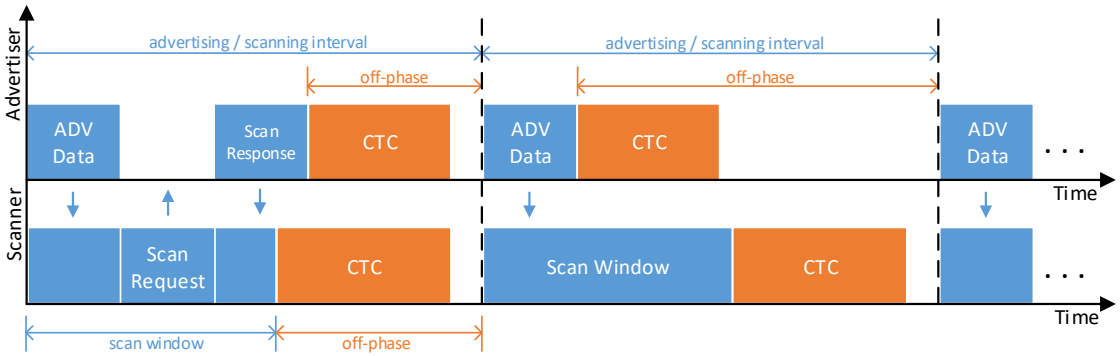


Figure 5.8: Adaptation of *X-Burst* to a connectionless communication of BLE.

As shown in Figure 5.8, at the first event, the scanner is scanning actively, i.e., it sends a scan request back to the advertiser, which responses with a scan response packet. At the second event, the scanner is scanning passively, i.e., no scan request is sent back and thus the advertiser can directly go to sleep mode. Instead of allowing the advertiser to switch off the radio, it is kept on and the CTC time slot starts. The maximum duration of the *off-phase* (and, therefore, the maximum time available for the CTC time slots of an advertiser) is only limited by the *advertising interval* and the amount af advertising packets that are sent. The time between the beginning of two consecutive advertising packets (regardless if a scan request and scan response is sent) shall be less than or equal to 10 ms. Since the maximum amount of advertising packets sent within an event is three, i.e., one packet for each advertising channel, the maximum duration needed for transmissions within an event is 30 ms. Depending on the BLE specification, the *advertising interval* shall be a multiple of 0.625 ms and in the range of 20 milliseconds to 10.24 seconds. Hence, even by using a very short *advertising interval* of 100 ms, 70 ms are still available for *X-Burst*.
A scanner always listens at the start of a scanning event for advertising packets. After the scan time is expired, the device would usually go to sleep mode, but, instead, the radio is kept on and the CTC time slot is started. The maximum *off-phase* and, therefore, the maximum time available for the CTC time slots of a scanner, is only limited by the *scanning interval* and the *scan window*. Depending on the BLE specification, the *scanning interval* and the *scan window* shall be less than or equal than 10.24 seconds. Hence, finding enough time for *X-Burst* between two consecutive scanning events should be possible.

A **connection-oriented communication** is needed to be able to exchange data bidirectionally between two BLE devices. The connection setup is done using the advertising channels. Thereby, an initiator responds with a connection request to a received adverting packet that supports connections. After the connection setup is done, the two devices, i.e., master and slave, are able to exchange data bidirectionally.
At the beginning of each connection event, defined by the *connection interval*, the master always sends a data packet to the slave, which has to be responded. After these first transmissions, further data can be exchanged until the connection event is closed.

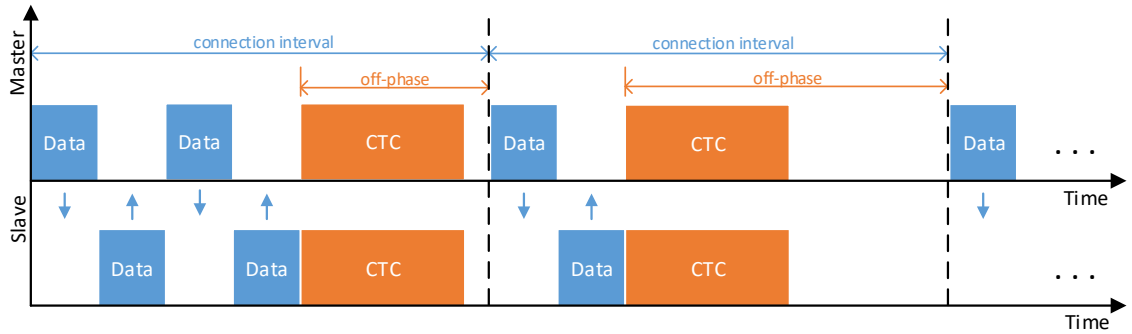


Figure 5.9: Adaptation of *X-Burst* to a connection-oriented communication of BLE.

As can be seen in Figure 5.9, at the first connection event, master and slave are exchanging several BLE data packets. After the connection event ends, i.e., no more data packets are transmitted, instead of turning off the radio, the latter is kept on and the CTC time slot starts. At the second connection event, only the minimum data exchange between master and slave is done. Hence, more time is available for the cross-technology communication. The maximum duration of the *off phase* and therefore, the maximum available time, for the CTC time slots, is only limited by the *connection interval* and the amount of exchanged BLE packets. Depending on the BLE specification, the *connection interval* shall be a multiple of 1.25ms in the range of 7.5 milliseconds and 4 seconds. The number of exchanged BLE packets within a connection event is arbitrary. Nevertheless, due to the flexibility in configuring the duration of the *connection interval*, finding enough time between two consecutive connection events should be feasible.

5.3.4 Implementation

Before the radio is initialized, the configuration of the *virtual radio* is done, i.e., setting the *CTC window*, default *header* of CTC messages, *priority*, *policy*, etc. After the configuration is complete, a new process is started, which is the core element of the *virtual radio*. In the following, this process will be called *vr-process*, i.e., virtual radio process. Immediately after boot, the *vr-process* waits for a specified amount of time to make sure that the device has booted completely and successfully established connections with the devices in its proximity. Depending on the configuration, i.e., with or without *auto configuration*, the *vr-process* will start measuring the duty cycle of the radio or not.

Measuring the Radio Duty Cycle

In case the *auto configuration* was enabled, the *vr-process* autonomously determine the radio duty cycle. Therefore, the *virtual radio* measures each radio duty cycle for a specified time. This is done by calculating the duration between two consecutive calls of the on-function of the *virtual radio*. Each duration is measured in *rtimer* ticks and is saved for further processing. Depending on the configuration, it is possible to turn off the radio during the measurements since only the information from the upper layers is used to determine the radio duty cycle. Hence, measuring false duty cycles, e.g., receiving packets change the on and off times of the radio, can be avoided. Some RDC mechanisms will turn on the radio multiple times within one duty cycle, e.g., the two CCA checks of ContikiMAC. In the following this is called *multiple on-periods*. Since every duration of two consecutive calls of the on-function is measured and used for determining the duty cycle, these additional measurements within a duty cycle have to be filtered out. Otherwise a wrong duty cycle is calculated. This is done by defining a minimum duration that the radio has to be turned off before it is turned on again. Hence, each measured duty cycle that is smaller than the defined duration is discarded and not used for further processing. After the measurement time is expired, the duty cycle is calculated as the average of all measured cycles during the specified time. Additionally, also the amount of *multiple on periods* within a duty cycle is determined. If no duty cycle was detected and the radio was turned off during the measurements, it has to be turned on again. In case the *auto*

configuration was disabled, the duty cycle has to be manually defined in the *ctc.h* file, as described in Section 5.3.2.

After all configurations are complete and the duty cycle was specified, the *vr-process* will enter an endless loop. At the beginning of each iteration, the *vr-process* yields, i.e., it goes to sleep mode. Hence, other processes can run in the meantime. If there is enough time for the cross-technology communication, the *vr-process* gets called, i.e., it gets polled, and the CTC time slot starts. Depending on the used RDC mechanism, this operation si carried out differently.

RDC Adaptation

When no radio duty cycle was detected or defined, a schedule between the OS and *X-Burst* has to be created. This is done by using Contiki's *rtimer* which is set once before the *vr-process* enters the endless loop and every time it gets polled. After the *rtimer* expires, a function is called to pull the *vr-process*. Hence, it is guaranteed that the *vr-process* is always called in the exact same interval as specified.
In case a RDC mechanism is used, the *virtual radio* is called differently, since it has to adapt to the existing mechanism. In Contiki it is not allowed to use multiple *rtimer* simultaneously. Hence, it is not possible to use Contiki's *rtimer* for scheduling the CTC time slots since those are already needed for the periodic wake-ups of the RDC mechanism. To solve this problem, whenever the radio should be turned off, i.e., when the off-function of the *virtual radio* is called, multiple calculations are done to determine if the next *off-phase* of the duty cycle is suitable for the CTC time slot. In particular, every time the off-function of the *virtual radio* is called, it is checked what kind of RDC mechanism is used. Depending on the used mechanism, the *virtual radio* wakes up the *vr-process* only if the next *off-phase* is suitable. For example, ContikiMAC uses two CCA checks within a duty cycle: if the *vr-process* is polled whenever the off-function of the *virtual radio* gets called, the second CCA check would never be performed. To counteract this problem, the duration between two consecutive calls to turn on the radio is measured and compared with a specified threshold. If the duration is shorter than the threshold, the current on-period has to be the second CCA check. Therefore, the next *off-phase* will be suitable. Otherwise, the current on-period is the first CCA check and therefore, the *vr-process* cannot be called.
Another issue is that transmitting and receiving CTC messages takes some time. If the *vr-process* would be called regardless of the remaining time of the *off-phase*, it could happen that a message could not be fully sent or received. Hence, energy is wasted since no useful information could be exchanged and the radio was kept on unnecessarily. Therefore, the remaining duration of the *off-phase* will be checked. If the device has received or sent a packet, the radio was on for a longer duration and thus, the next *off-phase* will be shorter. To avoid waking up the *vr-process* unnecessarily, i.e., the remaining time for the CTC time slot is too short for sending or receiving messages, a duration for the minimum amount of the remaining time of the *off-phase* can be defined. Hence, the *vr-process* will only be called if at least the specified duration is available.

Process Flow

When the *vr-process* gets polled, the CTC time slot starts. First, the configured channel of the radio is changed to the one specified for the cross-technology communication. Afterwards, the remaining duration of the *off-phase* is determined. Therefore, the period in which the radio was turned on within the current duty cycle is measured, and thus, the remaining duration of the *off-phase* can be determined. Depending on the configuration, i.e., the *priority* of *X-Burst*, the *CTC window* may be shortened if it does not fit within the remaining time. Hence, by setting the *priority* of *X-Burst* to low, it is guaranteed that the CTC time slot ends before the usual transmissions of the OS starts again. If the *priority* of *X-Burst* is set to high, the full *CTC window* is provided, regardless if the communication flow of the OS will be violated.
After the available duration for the CTC time slot was determined, it is checked if a CTC message has to be sent. This is done by checking a given flag, i.e., the *tx-flag*, of the *virtual radio*. If the *tx-flag* is set, e.g., by the application, the payload of the message is copied to a specified buffer, i.e., the *tx-buffer*, of the *virtual radio*. Hence, when the *virtual radio* detects that the *tx-flag* was set, the procedure for transmitting messages is called and the data of the *tx-buffer* is sent. In the case that the message could not be sent, i.e., the message was too long to be transmitted within the remaining time, the *tx-flag* and *tx-buffer* will not be cleared. Since it is not possible to overwrite the data of the *tx-buffer* as long as the *tx-flag* is not reset, the same message will be tentatively sent in the next iteration of the *vr-process*. After the message was successfully transmitted or the number of retransmissions exceeded a specified amount, the *tx-flag* and *tx-buffer* are cleared. Hence, a new message can be sent.
The time needed for transmitting the whole message is measured and the remaining time of the CTC slot is calculated. If the remaining duration is longer than a specified threshold, the procedure for scanning other CTC messages is started. Otherwise, the *vr-process* will go back to sleep mode before the CTC time slot ends to save energy. After the duration of the CTC time slot is expired, the channel of the radio is set back to the one used for the usual communications. If RDC is used the radio is turned off to save energy. The *vr-process* goes back to sleep mode and waits until it gets polled again. Afterwards the whole procedure will start again.

Transmitting Messages

Sending messages via cross-technology communication cannot be done at anytime. The instant of time in which a message will be transmitted is decided by the *virtual radio*. To inform the *virtual radio* that a message has to be sent, the *tx-flag* of the *virtual radio* has to be set and the payload of the message has to be copied to the *tx-buffer* of the *virtual radio*. This can all be done directly from the application by using the already existing *netstack_radio.set_value* and *netstack_radio.set_object* function of Contiki.
Depending on the configuration of the header, the CTC message is built differently. The payload and all the additional data needed for the message, e.g., the network ID, addresses, checksum, are copied into a new buffer in its transmitting order. After the message is built, the expected duration of its transmission is estimated. Since each byte is represented by two consecutively sent energy bursts with a fixed duration, the time needed for the full

transmission of the message can be estimated. Therefore, for each byte of the message, the durations of the two corresponding energy bursts are summed up and a specified amount of time, i.e., simulating the time between sending two energy bursts, is added. The estimated transmission time is calculated by summing up the duration of each byte of the message. If the estimated time is longer than the remaining duration of the CTC slot, the message will not be sent and a new attempt is made in the next iteration. Otherwise, the transmission of the message is started, where each byte is sent consecutively by calling the *send_byte* function of the CTC-HAL implementation of the used hardware. Each byte is represented as two hex values and the needed amount of payload bytes for generating the energy burst with the corresponding duration of the hex value is determined. This is done by using a lookup table that maps each hex value to the correct amount of payload bytes needed for generating an energy burst with the required duration. The actual value of the payload bytes can be arbitrary since there is no information encoded within the actual message, only the length of the payload is important. In this implementation, each payload byte has the value 0x00. Table 5.8 shows the actual used lookup table of the implementation of the TI CC2650 Launchpad when ZigBee is used, i.e., in IEEE mode.

Hex value	Energy burst duration	Payload bytes
0x0	192 μs	0
0x1	320 μs	4
0x2	448 μs	8
0x3	576 μs	12
0x4	704 μs	16
0x5	832 μs	20
0x6	960 μs	24
0x7	1088 μs	28
0x8	1216 μs	32
0x9	1344 μs	36
0xA	1472 μs	40
0xB	1600 μs	44
0xC	1728 μs	48
0xD	1856 μs	52
0xE	1984 μs	56
0xF	2112 μs	60

Table 5.8: Amount of payload bytes necessary to achieve the required energy burst durations with ZigBee.

As shown in Table 5.8, the gap between two successive energy burst durations is longer than the optimum durations defined in Table 4.1. This is due to the fact that the TI CC2650 Launchpad in IEEE mode uses an non-instantaneous measurement of the received signal strength, as described in Section 4.2, and the low measurement granularity of the RSSI, i.e., with a rate of 45 kHz. Hence, to be able to decode the received energy bursts correctly, the gap between two successive energy burst durations was extended.

To achieve the same durations when using the TI CC2650 Launchpad in BLE mode, a different mapping is needed due to the higher data rate of BLE compared to ZigBee. The lookup table for BLE is shown in Table 5.9.

Hex value	Energy burst duration	Payload bytes
0x0	192 μs	1*14 (14)
0x1	320 μs	1*30 (30)
0x2	448 μs	2*23 (46)
0x3	576 μs	2*31 (62)
0x4	704 μs	3*26 (78)
0x5	832 μs	3*31 (94)
0x6	960 μs	3*37 (110)
0x7	1088 μs	4*32 (126)
0x8	1216 μs	4*36 (142)
0x9	1344 μs	5*32 (158)
0xA	1472 μs	5*35 (174)
0xB	1600 μs	6*32 (190)
0xC	1728 μs	6*34 (206)
0xD	1856 μs	6*37 (222)
0xE	1984 μs	7*34 (238)
0xF	2112 μs	7*36 (254)

Table 5.9: Amount of payload bytes to achieve the required energy burst durations with BLE.

As discussed in Section 4.1.2, BLE test packets are used for generating the required energy bursts. Depending on the BLE specification, only data packets with a maximum payload of 37 bytes can be sent. Hence, usually multiple packets have to be sent consecutively to generate energy bursts with the required durations. This can be seen in Table 5.9. Nevertheless, the TI CC2650 Launchpad is able to sent BLE test packets with a payload up to 255 bytes. Thus, only one test packet is sent for generating the required energy bursts.

To inform a receiver that the following energy bursts identify a CTC message, a preamble is sent before every message. Further, the energy burst representing the high nibble of a byte, i.e., the four most significant bits, is sent first and only standard compliant functions are used for sending data packets. After every byte was sent, the remaining duration of the CTC time slot is checked and the next byte is only transmitted if enough time is available. Therefore, it is always guaranteed that the given amount of time for the CTC slot is never exceeded. After the message was successfully transmitted, the device can scan for other CTC messages, unless specified differently.
To be sure that a receiver will be awake when a CTC message is sent, it is possible to configure a transmitter to send a message during the whole *CTC window*. To this end, the remaining duration of the CTC slot, after a successful transmission, is calculated.

The message will be transmitted as long as enough time is available within the time slot. To avoid that two consecutively sent messages will be misinterpreted, i.e., merged, by a receiver, the *virtual radio* has to wait some time between sending the messages. if a transmitter is configured to send a message during the whole *CTC window*, it will not be able to receive other CTC messages.

Receiving Messages

After the *virtual radio* has transmitted a message, the remaining duration of the CTC time slot is determined. Depending on the configuration and the remaining time available in this iteration, the *vr-process* will start the receiving procedure or will go back to sleep mode.
In the receiving procedure, the RSSI of the configured channel is measured frequently. Since some radios need to be configured to be able to measure the RSSI (e.g., when using BLE the radio has to be in scanning mode), the *prepare_scanning* function of the CTC-HAL implementation is called. Depending on the configuration of the RSSI parameters shown in Table 5.3, the thresholds needed for determining the duration of an energy burst as discussed in Section 4.2, are determined. Afterwards, the RSSI is measured frequently and compared to the different thresholds. If the RSSI exceeds the minimum threshold, an energy burst was detected and the measurement of its duration started. For each additional threshold, a separate measurement is started if the threshold is exceeded and stopped if the indicator falls back below the threshold again. To determine the duration, a timestamp is saved, in *rtimer* ticks, when a measurement is started or stopped. Afterwards, the duration is determined by calculating the difference between the two timestamps. If a duration is smaller than the minimum duration that was defined, it will be discarded and not used for further processing. Additionally, an outlier detection is done by checking if the measured duration is in a specified range to the previous measured duration of the energy burst. If the duration is outside the specified range, it will be discarded. This is also needed to filter out 'peaks', i.e., interference that occurred during a measurement. After the RSSI falls back below the minimum threshold, the measurement is stopped and the real duration of the energy burst is determined. Therefore, the average of all remaining measured durations, for each threshold that was exceeded within this energy burst, is calculated. This duration is close to the real duration of the detected energy burst as discussed in Section 4.2. Afterwards, the duration is verified and decoded by using a lookup table. It is also possible to use only one threshold for determining the duration of an energy burst, i.e., in the case that an instantaneous measurement of the received signal strength is used. Towards this goal, the *virtual radio* has do be configured accordingly.
The mapping from hex value to *rtimer* ticks can be seen in Table 5.10. Since the mapping is independent of the used technology, only one lookup table is needed among all technologies.

Hex value	Energy burst duration	*rtimer* ticks
0x0	192 μs	13
0x1	320 μs	21
0x2	448 μs	29
0x3	576 μs	38
0x4	704 μs	46
0x5	832 μs	55
0x6	960 μs	63
0x7	1088 μs	71
0x8	1216 μs	80
0x9	1344 μs	88
0xA	1472 μs	96
0xB	1600 μs	105
0xC	1728 μs	113
0xD	1856 μs	122
0xE	1984 μs	130
0xF	2112 μs	138

Table 5.10: Mapping from hex values to energy burst durations and *rtimer* ticks.

If there exists an entry of the measured duration in the lookup table, the measured duration is considered as a valid part of a CTC message. Due to the low granularity of the RSSI measurement, as the fact that the TI CC2650 Launchpad in IEEE mode uses an non-instantaneous measurement of the received signal strength, the duration is always compared to be in a specified range (defined duration $\pm$ ϵ). If a duration is not within the specified range of a defined duration in the lookup table, it is rejected.

To avoid decoding actual noise, i.e., not CTC related messages, the *virtual radio* is looking only for a specified preamble. In particular, the preamble is represented by four consecutive energy bursts, i.e., bursts representing the value 0x1010. After a preamble was detected, each following energy burst that is received will be treated as part of a CTC message. After each second received energy burst, only whole bytes are sent: the hex values represented by both bursts are merged together and saved as a byte to a buffer.
To guarantee that the CTC time slot will not be exceeded, the remaining time within the CTC slot is calculated after each RSSI measurement. In case no time is left, the receiving procedure will be aborted, regardless on whether a message is currently received. If the *priority* of the CTC scheme is set to high and a message needs more time to be successfully received, the *virtual radio* will exceed the granted time slot.
Since the duration between two related energy bursts is not too long, a timeout, i.e., the *rx_timeout*, is specified to detect the end of a message. In particular, if an energy burst was not received for a specified time, the *virtual radio* assumes that the transmission has ended and starts reconstruction the original data. During the measurement of energy bursts, all encoded values were saved as bytes to a buffer. To reconstruct the original data, the received bytes have to be parsed accordingly. The first byte of a CTC message represents the *header*. Depending on the configuration of the *header*, the *virtual radio* will parse the data accordingly. During the reconstructing of each field of the message, the

length of the received data is verified. In case the actual length of the data is different from the expected one, the data is discarded and a corresponding error message is shown. After the original data was reconstructed, it is posted with all additional informations to the application. Therefore, the application has to listen to a specific event, which is posted by the *virtual radio* in case a message was received.
Afterwards, the *virtual radio* will determine the remaining duration of the current iteration. If enough time is available, the receiving procedure is repeated. Otherwise, the *vr-process* will go back to sleep mode.

Usual Communications

The *virtual radio* is also responsible for managing the usual transmissions of the operating system. Therefore, all required functions defined in Section 5.3.1, have to be provided. These functions are needed for a communication between the upper layers and the radio layer, i.e., the actual implementation of the radio. For usual communications between the upper layers and the radio implementation, the *virtual radio* only operates as a forwarder, i.e., the corresponding data will only be looped through the *virtual radio*. Therefore, neither the upper layers nor the radio implementation will notice the existence of the *virtual radio*. Hence, the usual communications, i.e., transmissions, of Contiki will not be affected.

Chapter 6

Evaluation

This chapter shows an experimental evaluation of *X-Burst*. Section 6.1 briefly describes the setup used for the evaluation. In Section 6.2, a validation of *X-Burst* is given, showing a cross-technology communication between a ZigBee and a BLE device. Furthermore, the Packet Reception Rate (PRR) depending on the content of the payload of a CTC message is shown for both communication directions. In Section 6.3, a theoretical and practical evaluation of the actual achieved throughput is given. Moreover, the best possible throughput achievable with *X-Burst* is discussed and compared with the one actually achieved on real hardware. Additionally, the achieved throughput is shown as a function of the payload length. The power consumption and the memory footprint of *X-Burst* are discussed in Section 6.4. Section 6.5 and Section 6.6 show the behavior of *X-Burst* when adapting to different RDC mechanisms, as well the influence of the policy, priority, and other configurations, respectively. Section 6.7 evaluates the robustness of *X-Burst* by computing the PRR for both communication directions in the presence of different kinds of interference. Additionally, the PRR is shown as a function of the amount of payload bytes in a CTC message.

6.1 Experimental Setup

For the evaluation, two TI CC2650 LaunchPads[1] were used: one as CTC transmitter and one as CTC receiver. The devices were placed at a distance of one meter between each other. We evaluated *X-Burst* in an office environment, i.e., we could not explicitly minimize the background traffic from other devices. Hence, multiple WiFi access points, laptops and smartphones were present during the whole evaluation.

[1] more information about the TI CC2650 LaunchPad can be seen in Appendix B

For the configuration of the TI CC2650 LaunchPad in IEEE mode (i.e., the configuration of the *virtual radio*), the standard values shown in Section 5.3.2 were used.
For the configuration of the TI CC2650 LaunchPad in BLE mode, some values had to be modified. Due to the instantaneous measurement of the received signal strength in BLE mode, compared to the non-instantaneous one used in IEEE mode, and the low granularity of Contiki's *rtimer*, the measured durations are more likely to be a bit longer than defined. Hence, the range in which a duration is valid was changed to:

- CTC_DURATION_PLUS: 6
- CTC_DURATION_MINUS: 3

Since the TI CC2650 uses an instantaneous measurement of the RSSI in BLE mode, only one threshold is needed for the measurement of the duration of an energy burst. Hence, the configuration of the thresholds was changed to:

- CTC_RSSI_THRESHOLD_MIN: -75
- CTC_RSSI_THRESHOLD_MAX: -74

Depending on the role of a device, i.e., if it is used as transmitter or receiver, the CTC_WINDOW and the DUTY_CYCLE parameters were changed accordingly.

Due to the common mapping between hex values and energy burst durations among all technologies, *X-Burst* can also be used to send CTC messages between two ZigBee or two BLE devices. Since the focus of this thesis is on the communication between ZigBee and BLE devices, only cross-technology communication between ZigBee and BLE and vice-versa were evaluated. Showing a communication between devices using the same PHY layers using *X-Burst* is left as future work.

6.2 Validation

In this section, a cross-technology communication between a ZigBee device and a BLE device is shown to prove the functionality of *X-Burst*. Furthermore, the PRR depending on the content of the payload of a CTC message is shown.

Figure 6.1 shows a transmission from a ZigBee device to a BLE device. Therefore, the detection of the various energy bursts, representing the message, were recorded on the BLE device. For the transmission of the CTC message, only the necessary parts were sent: the preamble, the header and the data payload.
The full message is shown in the top Figure 6.1. For a better evaluation of the message, the different parts were enlarged and illustrated accordingly. Additionally, the background of the figure was changed for a better separation between the different parts of the message. The preamble and the header of the message can clearly be seen in the middle of the figure and the data payload is shown in the bottom. Additionally, the corresponding hex values of each energy bursts are displayed.

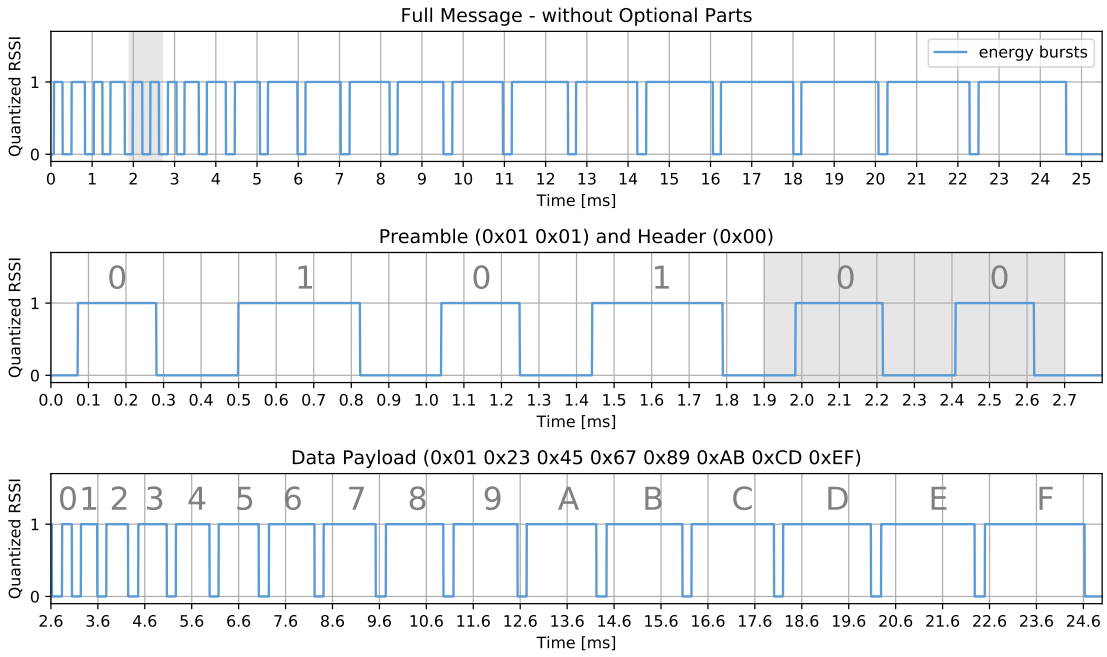


Figure 6.1: Reception of a CTC message on a BLE node.

In Figure 6.1, the different durations of the various energy bursts, representing the hex values in the range of 0x0 - 0xF, can clearly be distinguished.

Figure 6.2 shows the transmission of the same data payload the one seen previously, but including all optional parts of a CTC message: the preamble, the header, the network ID, the length byte, the receiver address, the transmitter address, the data payload and the checksum.
The message was sent again from a ZigBee device to a BLE device and the detection of the energy bursts were recorded on the BLE device. The full message is shown in the top of the figure. For a better evaluation of the message, the different parts were enlarged and illustrated accordingly. Additionally, the background of the figure was changed for a better separation between the different parts of the message.

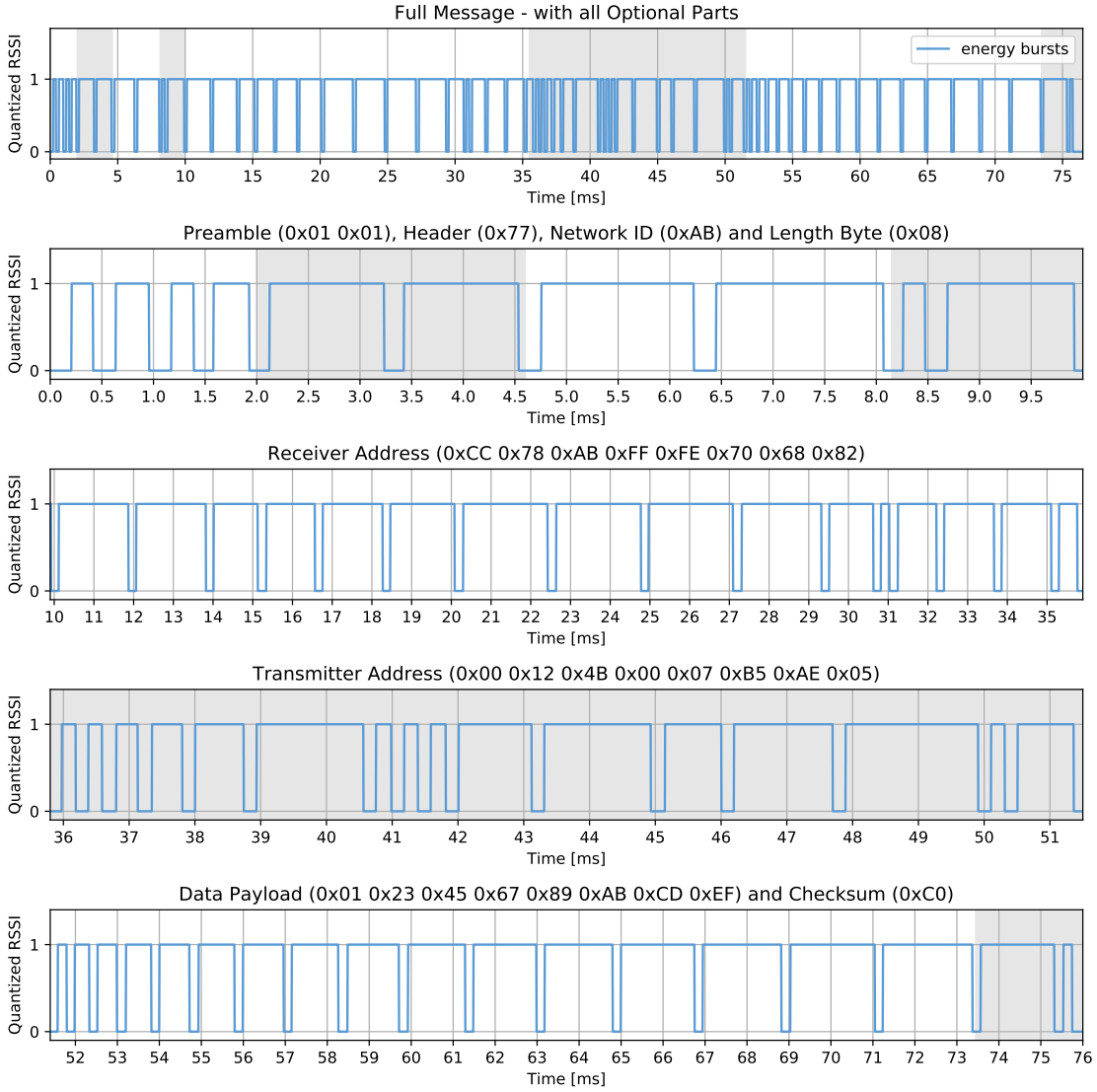


Figure 6.2: Reception of a CTC message on a BLE node including all optional parts.

It can clearly be seen in Figure 6.2 that the transmission time is significantly longer compared to when only the necessary parts were sent (Figure 6.1). In particular, if all optional parts are included, 19 bytes were additionally transmitted. Hence, the transmission time is about 52 ms longer as if only the necessary parts of a CTC message were sent for the same data payload.

Figures 6.3 and 6.4 show the PRR depending on the content of the data payload of a CTC message. In particular, 1500 messages including four identical payload bytes representing the corresponding hex value were sent for each hex value in the range [0x0 - 0xF]. Each CTC message includes the preamble, the header, the four payload bytes with the same, identical hex value, and the checksum for detecting transmission errors. Both figures evaluate the correctly received, corrupted and lost messages. Figure 6.3 shows the transmission from a BLE device to a ZigBee device and Figure 6.4 the reverse communication.

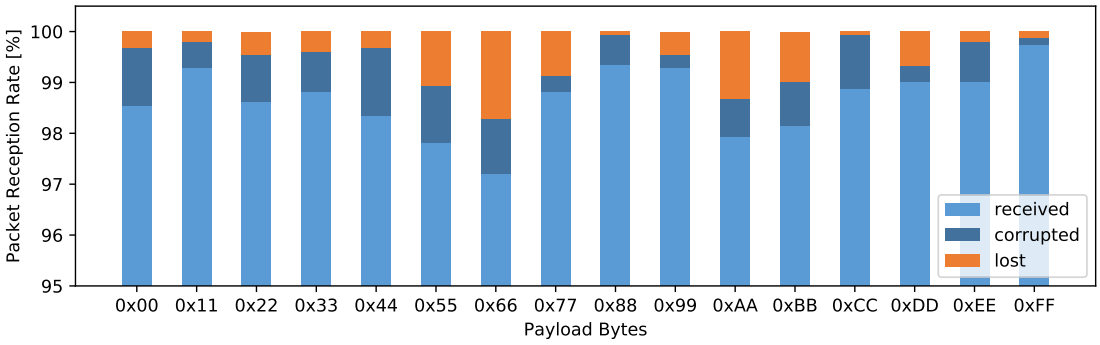


Figure 6.3: Packet reception rate when sending four bytes with identical hex value from a BLE to a ZigBee device.

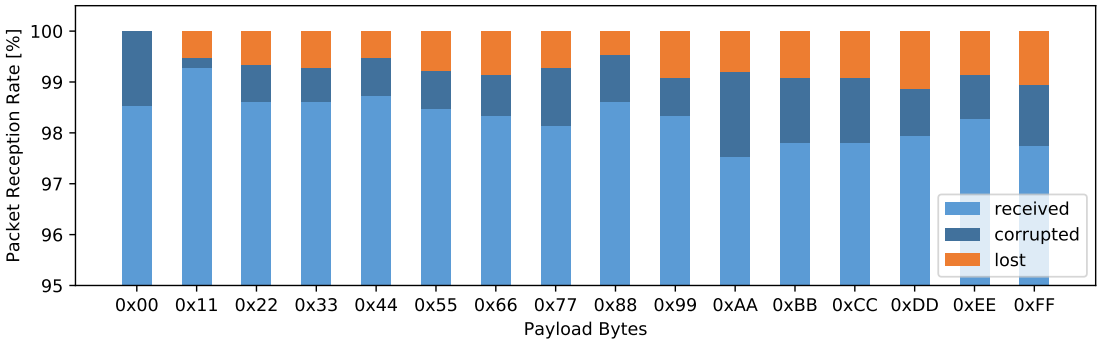


Figure 6.4: Packet reception rate when sending four bytes with identical hex value from a ZigBee to a BLE device.

As shown in Figures 6.3 and 6.4, a very high PRR for both communication directions was achieved. In particular, the correct reception of a CTC message never falls below 97 %. It can also be seen that the content of the data payload of a message has no influence on the reception rate for messages with a payload length of four bytes. The small variations of the PRR are due to the low measurement granularity of the RSSI and to the interference created by surrounding wireless devices during the evaluation.

6.3 Throughput

In this section, the throughput of *X-Burst* is evaluated. In particular, we did a theoretical and practical computation of the achievable throughput. Furthermore, the theoretical limit of *X-Burst* is discussed.

6.3.1 Theoretical Evaluation

The throughput of *X-Burst* strongly depends on the transmitted data, i.e., the higher the hex value, the longer the duration of the energy burst. Hence, a concrete throughput can not be determined. Nevertheless, an upper and lower bound for the achievable throughput can be calculated.
Besides the used mapping between hex values and durations, the throughput also depends on the time needed by the radio between two consecutive data transmissions. Since the used durations are fixed, the throughput only depends on the *preparation time* of the radio. The *preparation time* of the TI CC2650 LaunchPad for both modes, i.e., IEEE and BLE mode, was measured using a mixed signal oscilloscope.

For the following calculations, the overhead of sending the necessary parts needed for a transmission, i.e., the preamble, the header and the checksum, are not taken into account. Assuming that only large CTC messages are sent, those parts can be neglected. For the calculations, the durations from Table 5.10 were used.

When using the TI CC2650 LaunchPad, the *preparation time* is about 220 μs if the device is in IEEE mode and about 400 μs in BLE mode. The best data rate is achieved when only bytes with value 0x00 are transmitted, as those have the shortest durations. The time needed for transmitting one byte of value 0x00 can be calculated as:

$$0x00_{ieee} : (192 * 2 + 220) = 604\ \mu s$$

$$0x00_{ble} : (192 * 2 + 400) = 784\ \mu s$$

where 192 is the duration of the energy burst representing the hex value 0x0 and where 220 or 400 represents the *preparation time* of the TI CC2650 LaunchPad. All values are in microseconds.

Hence, sending one byte of value 0x00 takes 604 μs for ZigBee and 784 μs for BLE.

The upper bound of the achievable throughput can be calculated by determining how many bytes can be sent within one second. We must not forget to take again the *preparation time* of the radio into account:

$$upper\ bound_{ieee} : \frac{1000000}{604 + 220} = 1213.59\ B/s$$

$$upper\ bound_{ble} : \frac{1000000}{784 + 400} = 844.59\ B/s$$

Therefore, the **upper bound** is 1213.59 B/s or **9.71 kbit/s** for ZigBee and 844,59 B/s or **6.76 kbit/s** for BLE.

The lower bound can be calculated exactly as the upper bound, but instead of sending only bytes of value 0x00, bytes of value 0xFF are sent. Those bytes have the longest durations and thus, the lowest data rate is achieved as follows:

$$0xFF_{ieee} : (2112 * 2 + 220) = 4444\ \mu s$$

$$0xFF_{ble} : (2112 * 2 + 400) = 4626\ \mu s$$

$$lower\ bound_{ieee} : \frac{1000000}{4444 + 220} = 214.41\ B/s$$

$$lower\ bound_{ble} : \frac{1000000}{4626 + 400} = 198.97\ B/s$$

Therefore, the **lower bound** is 214.41 B/s or **1.7 kbit/s** for ZigBee and 198.97 B/s or **1.59 kbit/s** for BLE.

To have a more meaningful value of the throughput, the average achievable throughput, when the values of the transmitted data are equally distributed, is calculated. The average time necessary to transmit one byte can be calculated as:

$$average\ duration\ (hex) : \frac{\sum durations}{16} = \frac{18432}{16} = 1152\ \mu s$$

$$average\ duration\ (byte)_{ieee} : (1152 * 2 + 220) = 2524\ \mu s$$

$$average\ duration\ (byte)_{ble} : (1152 * 2 + 400) = 2704\ \mu s$$

Hence, transmitting one byte takes on average about 2.524 ms for ZigBee and about 2.704 ms for BLE. The average throughput is determined by:

$$average_{ieee} : \frac{1000000}{2524 + 220} = 364.43\ B/s$$

$$average_{ble} : \frac{1000000}{2704 + 400} = 322.16\ B/s$$

The **average throughput** is hence 364.43 B/s or **2.9 kbit/s** for ZigBee and 322.16 B/s or **2.58 kbit/s** for BLE.

Apart from the *preparation time*, the **best achievable throughput of *X-Burst*** is limited by the technology with the lowest data rate that has to be supported, i.e., ZigBee. The best possible mapping from energy burst durations to payload bytes for ZigBee, when only standard-complaint packets are used, is shown in Table 4.1. Table 4.2 shows the corresponding mapping for BLE. If the *preparation time* of the radio would be reduced, e.g., through improvements or by using a different hardware, the achievable throughput would be increased.
Ideally, the *preparation time* would be reduced to zero. This, however, would be a problem as a receiver would not be able to distinguish between two consecutive data transmissions anymore. Hence, some time between sending two consecutive packets is need. Assuming a receiver with a sampling rate of the RSSI of 1 MHz and the use of an instantaneous measurement of the received signal strength would allow reducing the *preparation time* to a minimum of about 10 μs.
Reducing the *preparation time* to 10 μs and using the mapping from Table 4.1 would lead to the following throughput:

- upper bound: **19.8 kbit/s**
- average: **9.05 kbit/s**
- lower bound: **5.87 kbit/s**

6.3.2 Practical Evaluation

To assess the throughput of the actual *X-Burst* implementation, a practical evaluation on the TI CC2650 was carried out. In particular, 1500 messages with a variable payload length were sent for three different types of payload: bytes with value 0x00 or 0xFF only, or bytes equally distributed among all possible hex values. Each CTC message includes the preamble, the header, the payload bytes and the checksum for detecting transmission errors. Additionally, the time needed for transmitting the 1500 messages was measured and only the correctly received messages were used for determining the throughput. Furthermore, also the overhead of each message, i.e., sending the preamble, the header and the checksum were also taken into account.

The following two figures show the achieved throughput on real hardware as a function of the payload length for different kinds of payload. Figure 6.5 shows the throughput of a transmission from a BLE device to a ZigBee device and Figure 6.5 shows the throughput for the reverse communication.

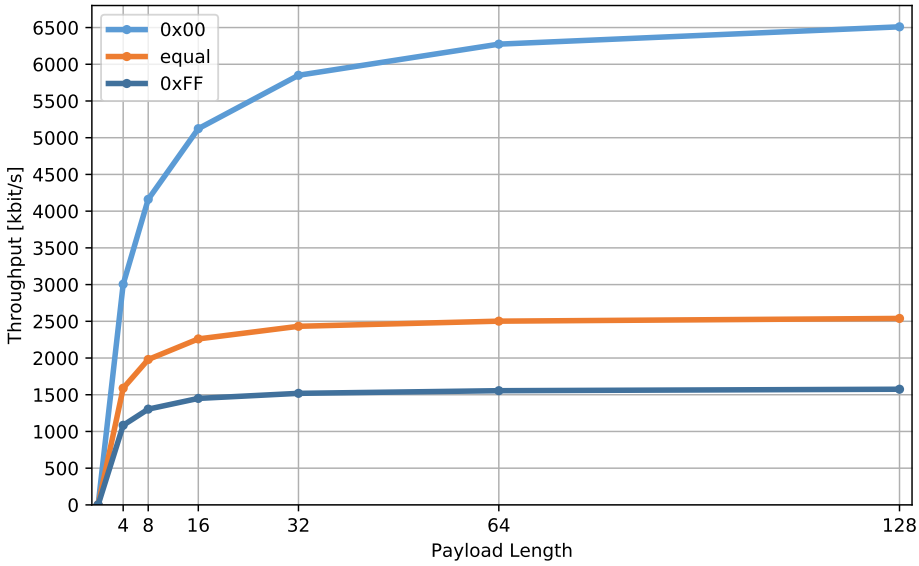


Figure 6.5: Throughput of a transmission from a BLE to a ZigBee device depending on the payload length for different kinds of payload.

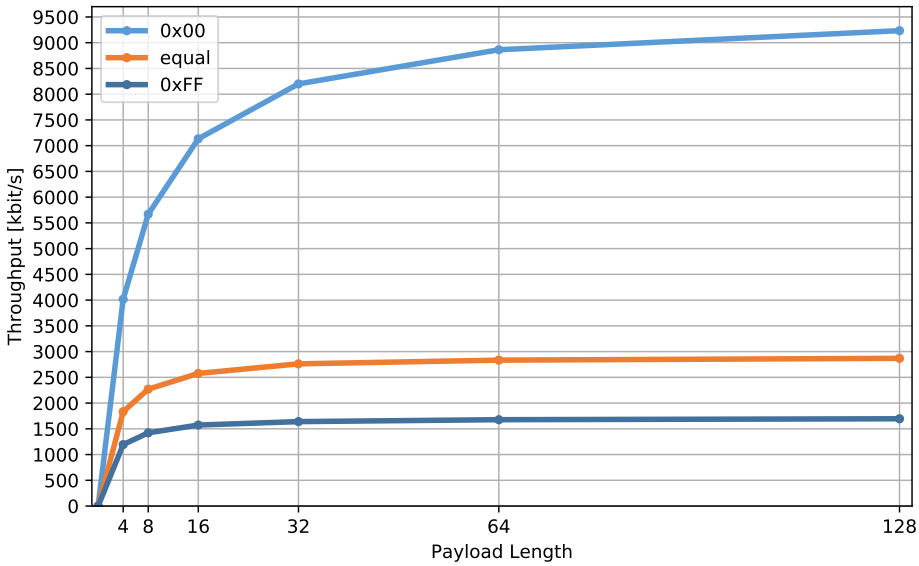


Figure 6.6: Throughput of a transmission from a ZigBee to a BLE device depending on the payload length for different kinds of payload.

As shown in the Figures 6.5 and 6.6, the achieved throughput increases with the payload length. This is due to the included overhead of each message, i.e., the preamble, the header and the checksum. The more bytes are sent, the less influence has the overhead of the message to the achieved throughput. This behavior is only true until the payload length reaches a specified value, i.e., 64 bytes. Increasing the payload even further would not increase the achieved throughput significantly. On the contrary, the throughput could even get worse again. This behavior is due to the decreasing PRR of correctly received CTC messages when more bytes are sent within one message. As a result, the time needed for transmitting the message will be increased and thus, the probability of a transmission error, e.g., through the occurrence of an interference during transmission, is increased.

The following values were achieved by evaluating the throughput of *X-Burst* on real hardware:

BLE → ZigBee:
upper bound: **6.51 kbit/s**, average: **2.54 kbit/s**, lower bound: **1.58 kbit/s**

ZigBee → BLE:
upper bound: **9.23 kbit/s**, average: **2.87 kbit/s**, lower bound: **1.7 kbit/s**

The achieved throughput when BLE is used as transmitter is significantly smaller as in the case of ZigBee due to the larger *preparation time* of the radio when the TI CC2650 is used in BLE mode.

The practical evaluation of the throughput on real hardware confirms the theoretical calculated throughput in Section 6.3.1.

6.3.3 Summary

The following table compares the theoretically calculated throughput in Section 6.3.1 with the one evaluated on real hardware in Section 6.3.2. Additionally, the limit of *X-Burst* is shown.

	Upper Bound	Average	Lower Bound
Limit	19.8 kbit/s	9.05 kbit/s	5.87 kbit/s
Theoretical$_{ieee}$	9.71 kbit/s	2.9 kbit/s	1.7 kbit/s
Practical$_{ieee}$	9.23 kbit/s	2.87 kbit/s	1.7 kbit/s
Theoretical$_{ble}$	6.76 kbit/s	2.58 kbit/s	1.59 kbit/s
Practical$_{ble}$	6.51 kbit/s	2.54 kbit/s	1.58 kbit/s

Table 6.1: Comparison of the theoretically and practically evaluated throughput achieved by *X-Burst*.

6.4 Energy Consumption and Memory Footprint

In this section, the energy consumption and the memory footprint of *X-Burst* are evaluated. In particular, the energy consumption of the CC2650 Microcontroller Unit (MCU) is shown for different modes of operations. Furthermore, the energy consumption of some specific scenarios are discussed. Additionally, the memory footprint of *X-Burst* in terms of RAM and ROM usage is analyzed.

X-Burst only affects the energy consumption when a RDC mechanism is used. In particular, only the duty cycle of the MCU is modified by *X-Burst*, i.e., the MCU is kept on for transmitting or receiving CTC messages. Hence, only the energy consumption of the MCU is discussed.
The energy consumption of the CC2650 MCU was calculated for different modes of operations by using the values of the datasheet[2]. Those values match with the energy consumption determined experimentally on the CC2650 MCU from [21]. The voltage supply of the TI CC2650 LaunchPad for normal operations is about 3.3 V. Table 6.2 shows the energy and power consumption for the CC2650 wireless MCU of different modes of operations.

[2]http://www.ti.com/lit/ds/symlink/cc2650.pdf

Mode	Energy Consumption	Power Consumption
Active	2.93 mA	9.66 mW
TX at 0 dBm	6.1 mA	20.13 mW
Active & TX	9.03 mA	29.79 mW
RX	5.9 mA	19.47 mW
Active & RX	8.83 mA	29.13 mW
Standby (low power)	1 μA	3.3 μW

Table 6.2: Energy and power consumption of different modes of operations for the CC2650 MCU.

As shown in Table 6.2, it is not distinguished if the TI CC2650 LaunchPad is in IEEE or BLE mode. This is because the energy consumption is independent of the used technology.

The energy consumption of *X-Burst* strongly depends on the used configuration. Hence, a concrete value about the additional energy consumption of a device using *X-Burst* cannot be given. The energy consumption strongly depends on the following values:

- CTC_WINDOW
- CTC_POLICY
- DUTY_CYCLE

Hence, the energy consumption will change depending on the configuration of the values above.

To give a better understanding of the additional energy consumption caused by *X-Burst*, the consumption of some specific operations are shown. Table 6.3 shows the energy consumption of transmitting different CTC messages depending on the payload length. Therefore, the time needed for transmitting the different CTC messages was measured and the energy consumption was calculated depending on the values in the datasheet of the CC2650. For calculating the minimum and maximum energy consumption, messages with payload bytes of value 0x00 and 0xFF only were sent, respectively. To calculate the average consumption, messages with payload bytes equally distributed among all possible hex values were sent.
Each message was sent in IEEE mode and includes the preamble, the header, the payload bytes and the checksum. The only difference between sending in IEEE and BLE mode is the *preparation time* of the radio. In case the device is in BLE mode, the time between two consecutively data transmissions is about 180 μs longer than in IEEE mode, as shown in Section 6.3.1. This does not significantly change the energy consumption of transmitting CTC messages. Therefore, it was not distinguished if the device is in IEEE or BLE mode.

Payload Length	Minimum [nAh]	Average [nAh]	Maximum [nAh]
4 Bytes	17.29	41.5	64.83
8 Bytes	26.14	68.25	110.13
16 Bytes	42.71	121.76	200.72
32 Bytes	75.77	228.84	386.7
64 Bytes	141.9	448.03	758.58
128 Bytes	274.22	886.8	1502.8

Table 6.3: Energy consumption of transmitting different CTC messages depending on the payload length.

Table 6.4 shows the energy consumption of measuring the RSSI frequently, depending on the duration, i.e., the configuration of the CTC_WINDOW. Since there is no difference if the device is in IEEE or BLE mode, the energy consumption of measuring the RSSI frequently is independent of the used technology.

RSSI measurement [ms]	Energy Consumption [nAh]
50	125.42
80	200.67
100	250.83
500	1254.17

Table 6.4: Energy consumption of measuring the RSSI frequently depending on the duration.

Assuming a continuous use of *X-Burst*, i.e., measuring the RSSI frequently without using duty cycling, the TI CC2650 LaunchPad would run for a full day by using an ordinary coin cell battery (220 mAh).

The memory footprint of *X-Burst* was analyzed in terms of RAM and ROM usage. Table 6.5 shows the memory footprint of the TI CC2650 LaunchPad in IEEE and BLE mode with and without *X-Burst*. The same application was used among all different modes.

	RAM used / free [kB]	ROM used / free [kB]
BLE mode	14.28 / 1.30	49.58 / 73.30
BLE mode with *X-Burst*	15.40 / 0.47	58.56 / 64.32
IEEE mode	11.67 / 4.21	45.55 / 77.33
IEEE mode with *X-Burst*	12.21 / 3.66	56.35 / 66.35

Table 6.5: Memory usage of the TI CC2650 LaunchPad in different modes.

As shown in Table 6.5, the memory usage of RAM and ROM is higher when the LaunchPad is in BLE mode. Moreover, the additional memory usage caused by *X-Burst* is not very high. Table 6.6 shows the additional usage of RAM and ROM caused by *X-Burst*.

	RAM usage [kB]	ROM usage [kB]
X-Burst only (BLE)	1.12	8.98
X-Burst only (IEEE)	0.54	10.8

Table 6.6: Memory footprint of *X-Burst*.

As shown in Table 6.6, the additional usage of RAM and ROM due to *X-Burst* is minimal. Hence, *X-Burst* suits well memory-constrained devices.

6.5 Adaptation to Different RDC Mechanisms

In this section, the adaptation of *X-Burst* to different RDC mechanisms is shown. We depict the seamless coexistence of *X-Burst* besides normal operations of the operating system by showing the time slots for accessing the radio, of the operating system, and of *X-Burst*. Additionally, the time for transmitting and receiving CTC messages is shown.

Figure 6.7 shows the adaptation of *X-Burst* when no RDC is used. In particular, a schedule of the accesses of the radio between *X-Burst* and the operating system is created. The TI CC2650 LaunchPad was used in IEEE mode and **nullRDC** was used as RDC mechanism.

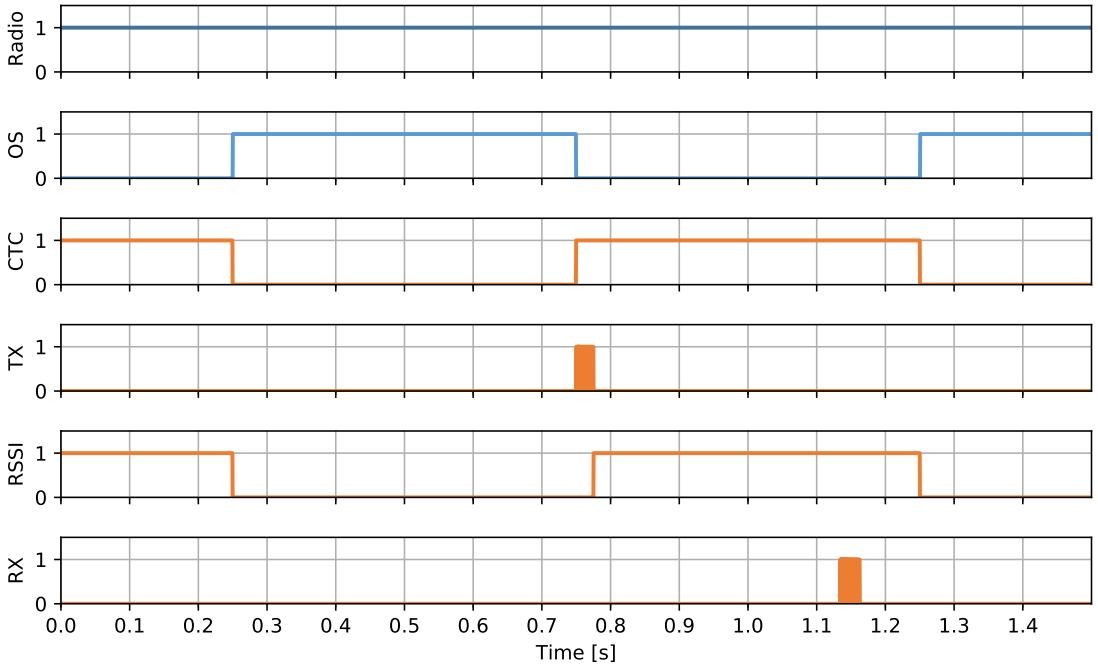


Figure 6.7: Adaptation of *X-Burst* - without RDC (nullRDC).

As can be seen in Figure 6.7, the radio is kept on for the whole time since no RDC is used. Furthermore, the different time slots for accessing the radio, of the OS and *X-Burst* (CTC) can be seen. In particular, the OS and *X-Burst* have both an assigned time slot of 500 ms. If the assigned time slot of one ends, the slot of the other starts immediately. At the beginning of the CTC time slot, one message was sent (TX) at time 0.78. Afterwards, the RSSI was measured frequently to look after other CTC messages until the assigned time slot has ended. As shown in Figure 6.7, one message was received (RX) at time 1.15.

In case a RDC mechanism is used, *X-Burst* adapts to the existing duty cycle in an unobtrusive way. To show the adaptation for ZigBee, the standard RDC mechanism of Contiki, i.e., ContikiMAC, was used. In the case of BLE, the adaptation to a connectionless communication is shown. Since the only difference between a connection-oriented and a connectionless communication is in the duration of the interval and the amount of data being exchanged, only the adaptation to a connectionless communication is shown.
Figure 6.8 shows the adaptation of *X-Burst* to the RDC mechanism **ContikiMAC**. The adaptation to a **BLE connectionless communication** is shown in Figure 6.9.

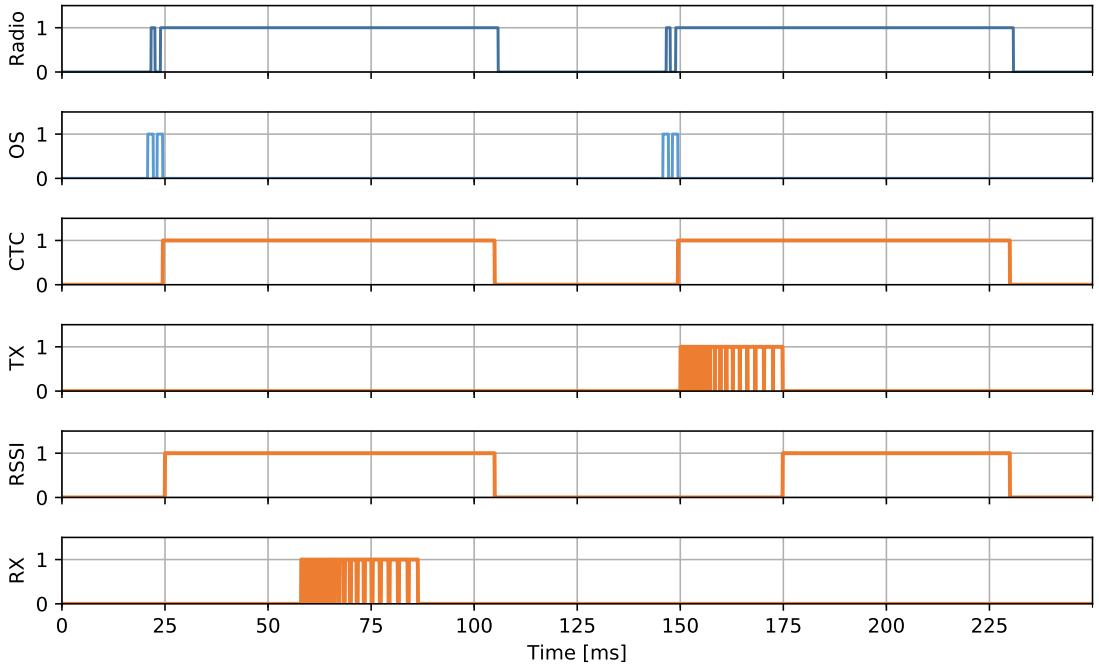


Figure 6.8: Adaptation of *X-Burst* - ContikiMAC.

The two CCA checks of ContikiMAC can clearly be seen in Figure 6.8 (OS). Since no transmissions were detected, the radio would normally go back to sleep mode but it is kept on so that the CTC time slot starts. In this example, the CTC_WINDOW was set to 80 ms. In the first time slot, no message was sent and the whole time slot was used to look after other messages, i.e., measure the RSSI frequently. Within this time, one message was received (RX). In the second CTC time slot, a message was transmitted (TX) and the time for measuring the RSSI was reduced accordingly. After each CTC time slot ends, the radio is turned off until the next two CCA checks of ContikiMAC.

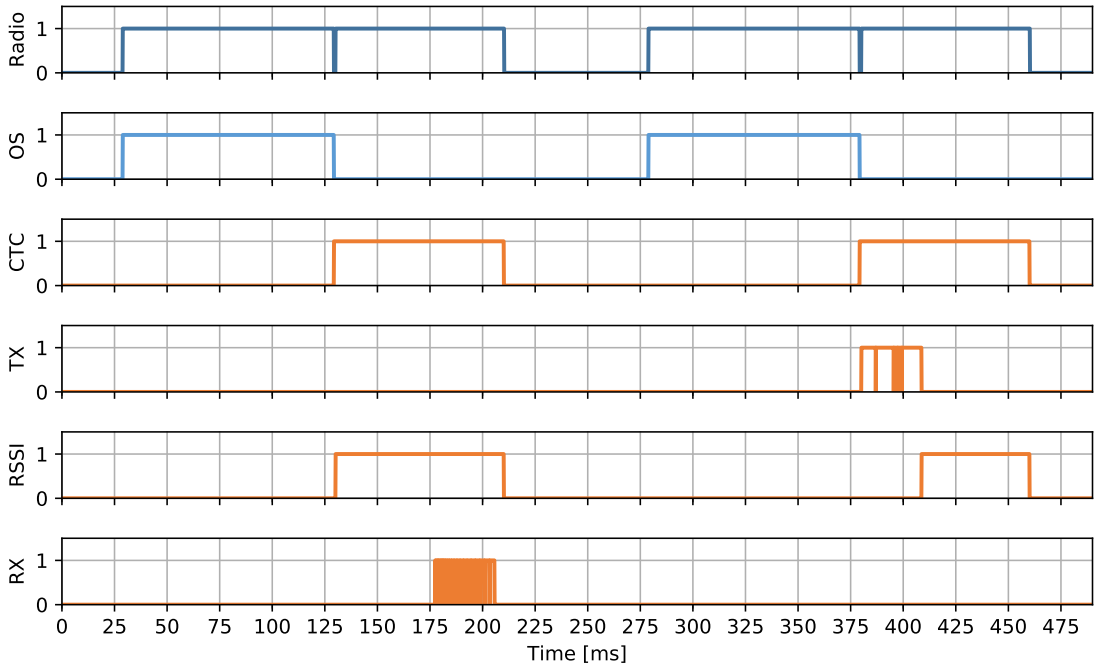


Figure 6.9: Adaptation of *X-Burst* - BLE connectionless communication.

Figure 6.9 shows a BLE connectionless communication with a *scanning interval* of 250 ms and a *scan window* of 100 ms. The CTC_WINDOW was set to 80 ms. As can be seen in Figure 6.9, the radio is turned off for a few milliseconds before each start of a CTC time slot. Since in BLE the RDC is done directly in the radio driver, *X-Burst* has no control over it. After the time of the *scan window* has passed, the BLE radio driver turns the radio off. Hence, *X-Burst* has to turn on the radio at the beginning of each CTC time slot. Besides that, the reception (RX) and the transmission (TX) of a CTC message is shown. After each CTC time slot ends, the radio is turned off until the next BLE scanning event.

6.6 Changing Configurations

In this section, the influence of changing different settings of *X-Burst* to its behavior is shown by altering its policy and priority.
Figure 6.10 and Figure 6.11 show the impact to the behavior of X-Burst when the **policy** and the **priority** are altered, respectively.

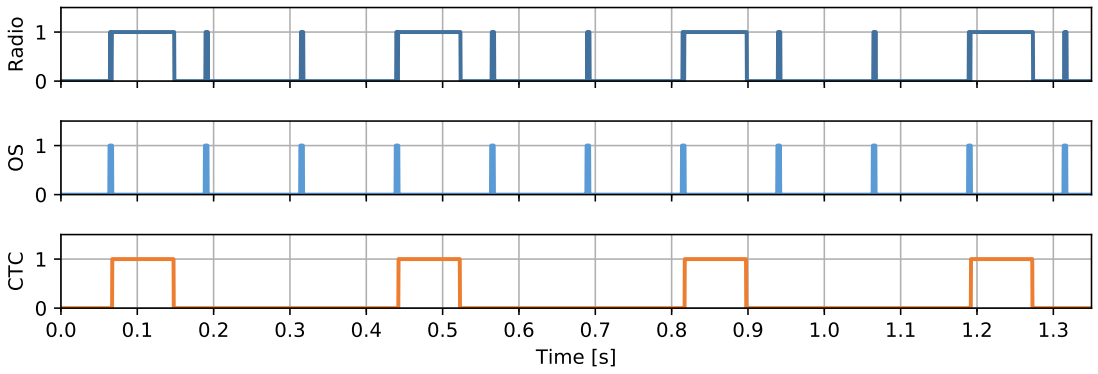


Figure 6.10: Influence of the policy to the behavior of *X-Burst*.

As shown in Figure 6.10, not every *off-phase* is used by *X-Burst*. In particular, the policy was set to 3, i.e., only each third *off-phase* is used. ContikiMAC was used as RDC mechanism.

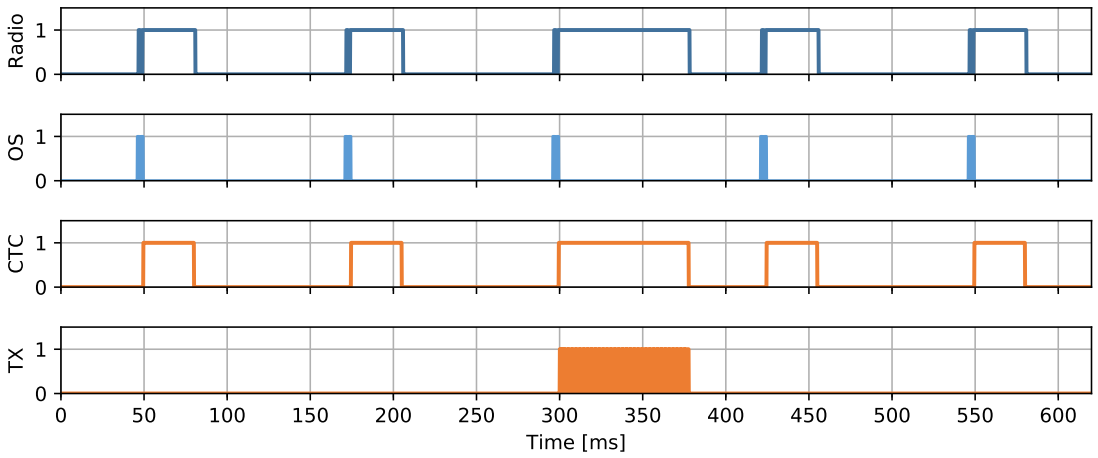


Figure 6.11: Influence of the priority to the behavior of *X-Burst*.

As shown in Figure 6.11, a CTC message larger as the assigned time slot was transmitted. ContikiMAC was used as RDC mechanism and the CTC_WINDOW was set to 30 ms. Usually, only messages with a transmission time lower than the remaining time of a CTC time slot can be transmitted. Nevertheless, setting the *priority* of *X-Burst* to

high, allows exceeding the assigned time slot regardless if the communication flow of the OS will be violated. In the Figure 6.11, the *priority* was set to high and thus, a message with a transmission time of about 77 ms could be sent. Afterwards, the priority was set back to low again and the normal behavior of *X-Burst* is restored. Setting the *priority* to high also allows the reception of messages with a needed reception time larger than the remaining time of a CTC slot.

Figure 6.12 shows the possibility of sending one CTC message during a full CTC time slot. Therefore, the message is sent repeatedly until no time within the current time slot is left.

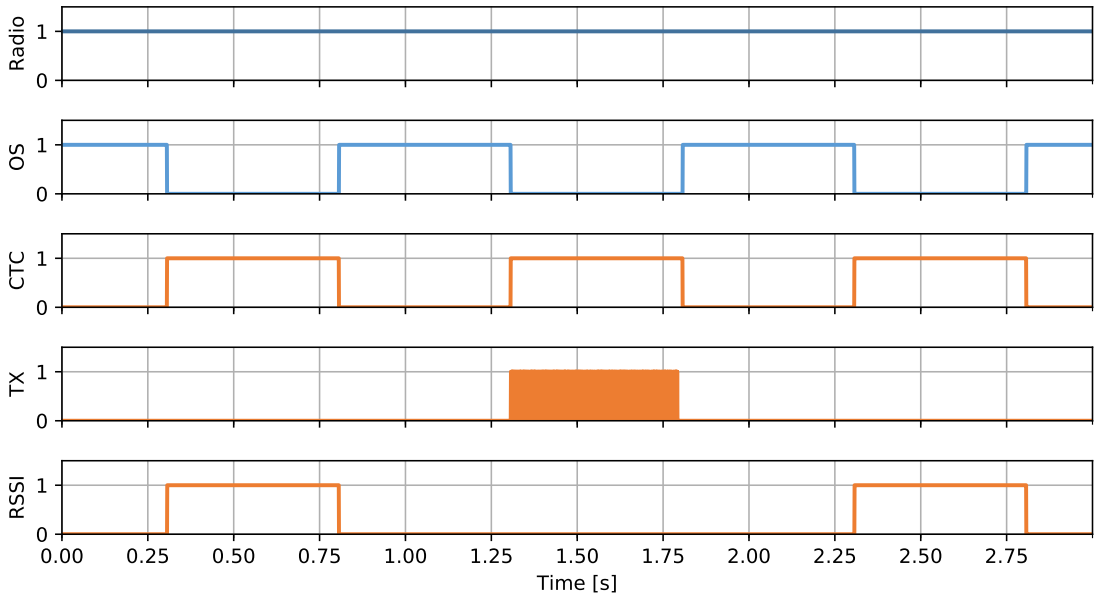


Figure 6.12: Sending one CTC message during the full CTC_WINDOW.

As shown in Figure 6.12, a message is sent repeatedly during the whole time slot. In particular, a message with a transmission time of about 32 ms was sent 15 times consecutively. As a consequence, scanning after other CTC messages is not possible within this time slot. The same settings as the ones used to derive Figure 6.7 were used, i.e., nullRDC and a CTC_WINDOW of 500 ms.

6.7 Robustness to External Interference

In this section, the robustness of *X-Burst* is evaluated. In particular, we evaluate the PRR in the presence of different kinds of interference. Furthermore, we show the PRR as a function of the payload length of a CTC message.

The PRR was measured for CTC messages with different types of payload and in the presence of various types of interference. In particular, we distinguished between correctly received, corrupted and lost messages when different types of interference were present during the measurements. In particular, 1500 messages with a fixed payload of eight bytes were sent for three different types of payload: bytes with value 0x00 or 0xFF only, or bytes equally distributed among all possible hex values (0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF), labelled as *EQUAL*. Each CTC message includes the preamble, the header, the eight payload bytes and the checksum for detection transmission errors.
We measured the PRR for four different scenarios. In the first one, no interference was explicitly generated to have a reference for the other measurements. The second one evaluated the PRR under the influence of other BLE communications. To this end, a BLE headset playing music was placed between the two TI CC2650 LaunchPads. For the third and fourth scenario, the influence of the PRR under ongoing WiFi communications was evaluated. Towards this goal, the used channels of the two Launchpads were changed to fully overlap with the WiFi channel 6, i.e., ZigBee to channel 18 and BLE to channel 17. In particular, the influence of audio and full HD video streaming were analyzed. Towards this goal, a smartphone was placed between the two devices to generate the desired interference. Figure 6.13 shows the communication from BLE to ZigBee and Figure 6.14 the reverse communication.

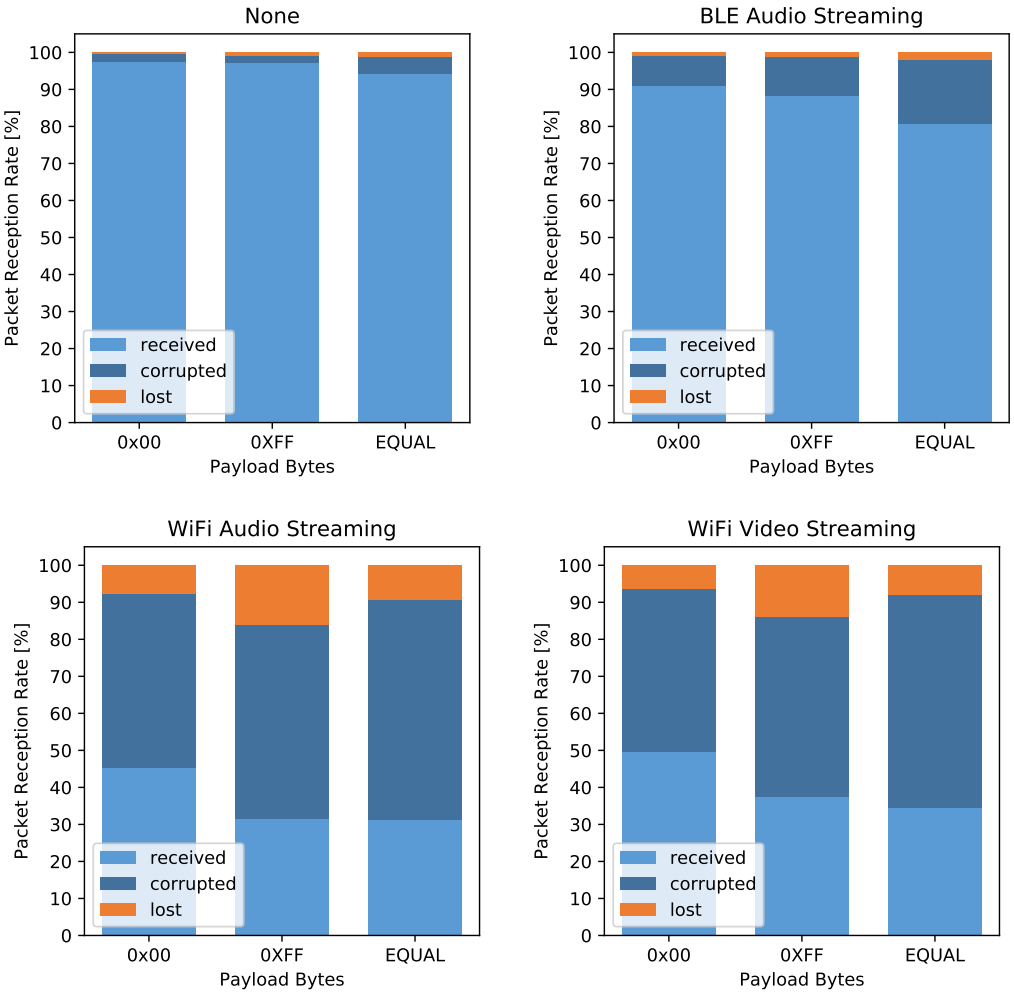


Figure 6.13: Packet reception rate when transmitting from BLE to ZigBee in the presence of different kinds of interference.

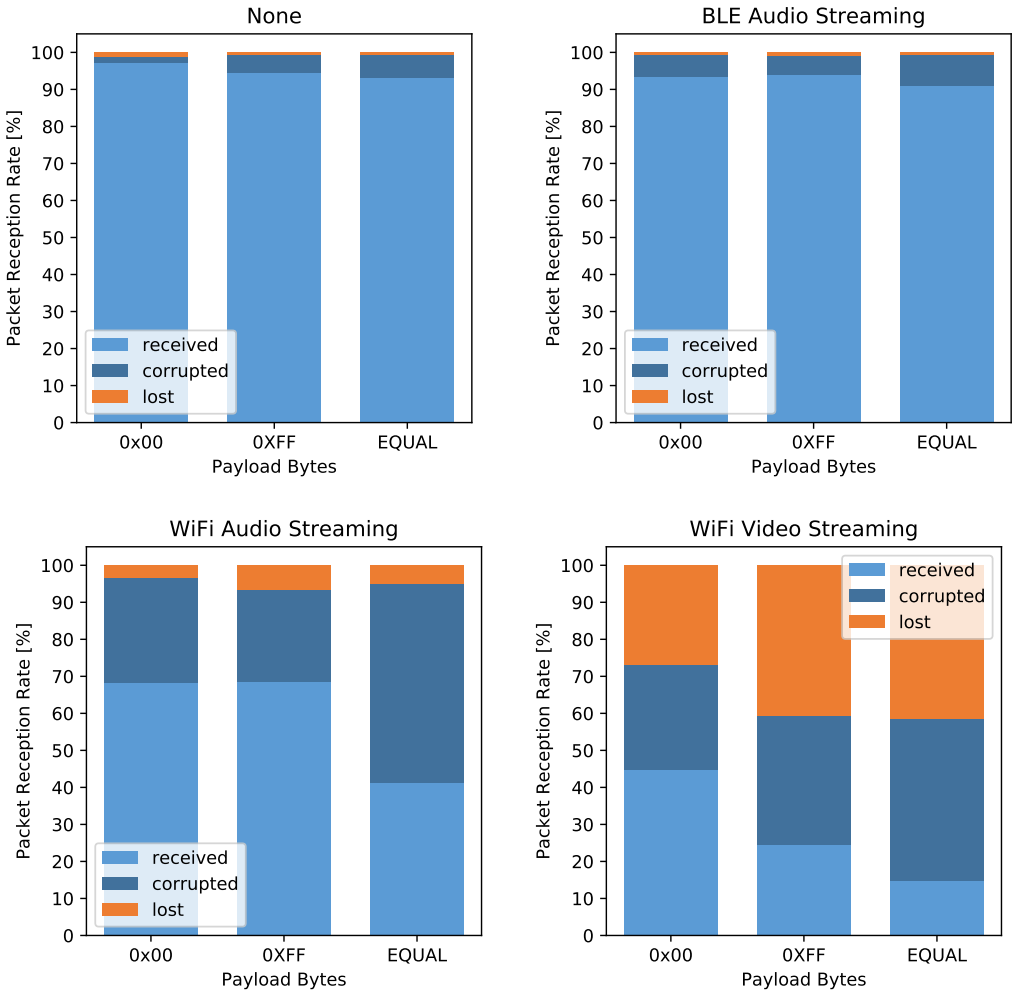


Figure 6.14: Packet reception rate when transmitting from ZigBee to BLE in the presence of different kinds of interference.

The behavior of the PRR under the influence of different types of interference was as expected. In case no interference was explicitly generated, the PRR for the correctly received messages was always above 90 % for each type of payload.
The presence of other BLE communications had the least impact to the PRR of *X-Burst*. BLE uses channel hopping to avoid interference with other communications. Hence, a channel is only used for a very short time, which is the reason for the low influence to the PRR of *X-Burst*.
In the presence of WiFi communications, the PRR drops significantly. The reasons for that are the very high transmission power of WiFi (compared to BLE or ZigBee), and the fact that all data is sent over the specified channel, i.e., WiFi does not use channel hopping. For full HD video streaming, the PRR is even worse compared to audio streaming. This is because of the higher amount of transmitted data. As can be also noticed in

the figures, the PRR in the presence of WiFi communications is quite different depending on the communication direction. This is due to the uncontrollable behavior of the used smartphone as the different data being exchanged. A more reproducible and meaningful test case, determining the robustness of *X-Burst* in presence of WiFi communications, will be done as future work.
In Figure 6.13 and 6.14, it can also be seen that the PRR also depends on the type of transmitted data. In case bytes with value 0x00 only were transmitted, the highest PRR was always achieved. The time needed for transmitting such kind of a message is very short, since the energy burst representing the hex value 0x0 has the shortest duration. Hence, the probability of a collision with other transmissions is very low. In the case of sending bytes with value 0xFF only, the lowest PRR is achieved, since this kind of message has the longest transmission time. However, as shown in the figures, the PRR when the bytes of the payload are equally distributed among all possible hex values is always a little bit lower compared to the case were bytes with value 0xFF only were sent. Due to the various energy bursts, a decoding error because of a low measurement granularity of the RSSI, is more likely to occur as when only bytes of the same value are transmitted.
The detection of a CTC message was always above 80 % (except in the case of WiFi video streaming in Figure 6.14 where the decetion of a message was only above 58 %). Hence, implementing an error correction could significantly improve the PRR and, hence, the robustness of *X-Burst*.

The following two figures show the PRR as a function of the payload length for both communication directions. In particular, 1500 messages with a variable payload length, were sent for three different types of payload: bytes with value 0x00 or 0xFF only, or bytes equally distributed among all possible hex values. Each CTC message includes the preamble, the header, the payload bytes and the checksum for detecting transmission errors. Figure 6.15 shows the communication from BLE to ZigBee and Figure 6.16 the reverse direction.

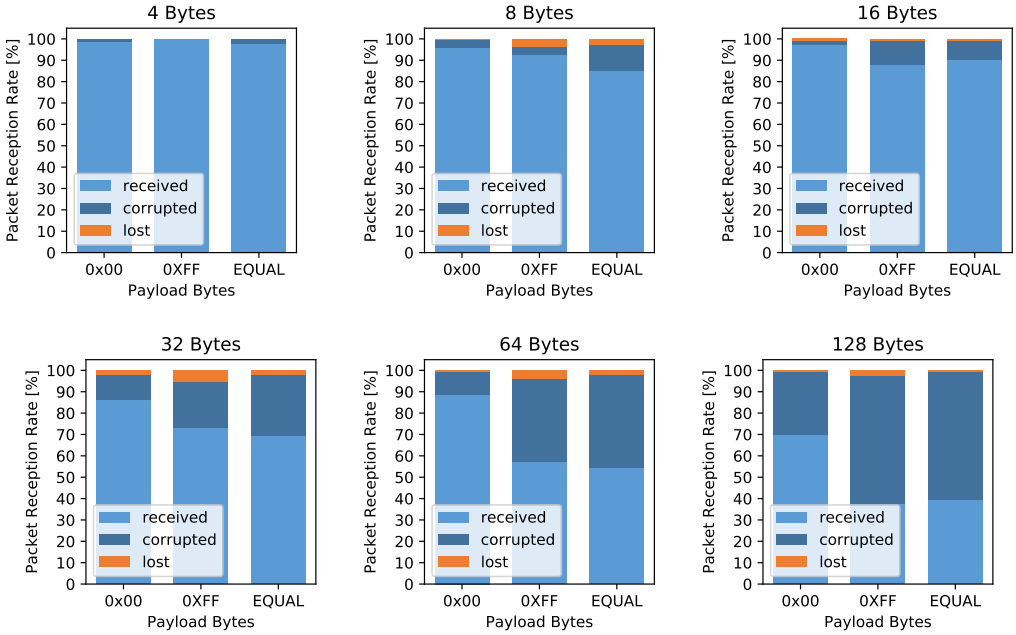


Figure 6.15: Packet reception rate when transmitting from BLE to ZigBee depending on the payload length.

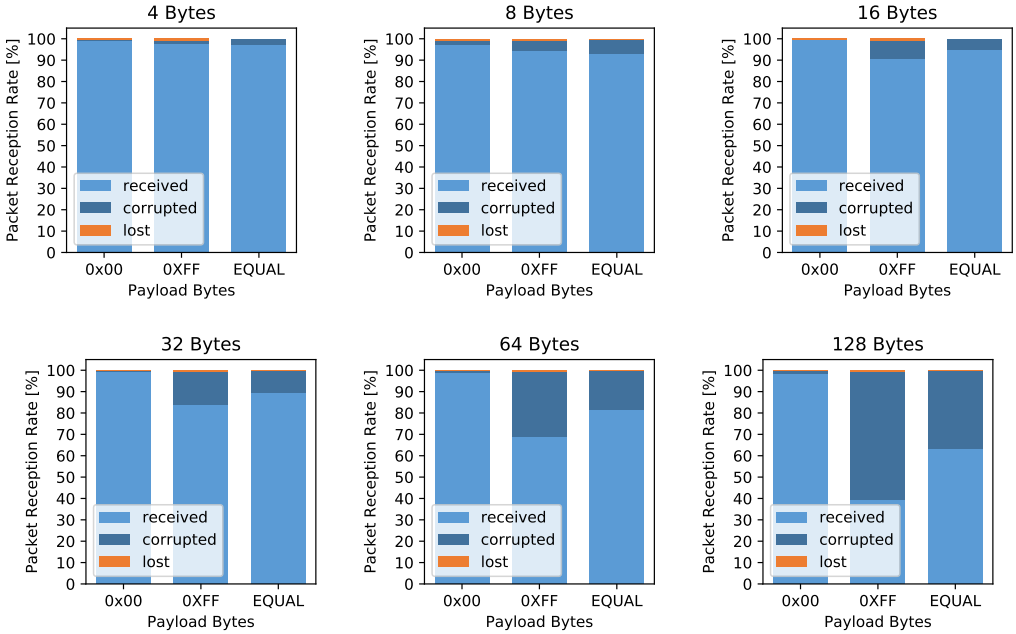


Figure 6.16: Packet reception rate when transmitting from ZigBee to BLE depending on the payload length.

As shown Figures 6.15 and 6.16, the PRR is inversely proportional to the payload length, i.e., the more bytes are transmitted, the lower is the PRR of a CTC message. The probability of a transmission error, e.g., through the occurrence of interference during the transmission, increases with the transmission time of a CTC message. Hence, the PRR of a message where bytes with value 0x00 only were transmitted was higher than the one of a message with bytes of value 0xFF. Usually, the PRR when sending a payload consisting of bytes equally distributed is lower than the one measured when sending only bytes of value 0xFF. This is not true for some measurements of Figure 6.16. The reason for that was a problem with the test case. It was not possible to let the receiver frequently measure the RSSI. In particular, the receiver had to pause the measurements for a few milliseconds each second. As result, the longer the transmission time of a message, the more messages were corrupted due to the breaks in the RSSI measurements. Since transmitting a message with payload bytes of value 0xFF only takes the longest time, those messages were affected the most by the problem with the use case.

Chapter 7

Conclusion & Future Work

This chapter concludes the thesis with a summary of the contributions of this thesis in Section 7.1 and an outlook about the future development of *X-Burst* in Section 7.2.

7.1 Conclusion

In this thesis, we presented *X-Burst*, a novel cross-technology communication approach for off-the-shelf IoT devices operating in the 2.4 GHz ISM band. Compared to most of the other works in the field of CTC, *X-Burst* enables a bidirectional communication between ZigBee (IEEE 802.15.4) and BLE devices with an average throughput of about 2.9 kbit/s. This is achieved by using precisely-timed energy bursts to convey information among devices with incompatible physical layer. The data is encoded as the duration of different energy bursts: in particular, 16 different energy bursts have been defined, where four bits of information is encoded in the duration of each burst. The data can be decoded by measuring the RSSI, which is typically a feature offered by all IoT devices.
X-Burst was integrated into the open source operating system Contiki, such that no changes to existing implementations are needed. Furthermore, a new radio driver for Contiki was written, i.e., the *virtual radio*, which manages the coexistence between the operating system and *X-Burst* in an unobtrusive way. Hence, the normal communication flow of the operating system is not affected by *X-Burst*.
X-Burst was evaluated on real hardware, showing a working cross-technology communication between a ZigBee and a BLE device. The evaluation showed that a PRR of more than 97 % for a payload length of four bytes could be achieved in an office environment with background interference. Furthermore, the robustness of *X-Burst* was evaluated in the presence of different kinds of interference, i.e., other BLE and WiFi communications. As a result, the PRR drops up to 80 % in the presence of other BLE communications and up to 15 % in the case of WiFi. Additionally, a correlation between the PRR and the payload length of a CTC message could be noticed.
Since no special hardware is required, *X-Burst* can easily be ported to other hardware platforms, enabling CTC among various IoT devices.

7.2 Future Work

In the following, an outlook about the future development of *X-Burst* is given.

Adding WiFi support. *X-Burst* was designed to be independent of the used technology or hardware platform. Hence, it will work on every device that fulfills the general requirements shown in Section 3.1.1. In the future, we also want to enable a bidirectional communication with WiFi devices, to obtain a CTC scheme allowing a bidirectional communication among the three most used wireless technologies: ZigBee, BLE and WiFi. The main challenge of using *X-Burst* on WiFi devices is the ability of reading the RSSI of a channel. Furthermore, the bandwidth of a WiFi channel is 22 MHz, i.e., it is relatively large compared to ZigBee's and BLE's 2 MHz channels. Hence, the WiFi channels have to be divided into smaller subcarriers to enable a proper communication.

Clock synchronization. We are planning to use *X-Burst* for synchronizing the clocks of heterogeneous devices. In industrial measurements and data acquisition systems, it is necessary to use heterogeneous devices for observing the same event. Hence, a proper synchronization among these devices is required to give sense to the measured data. Otherwise, a rational analysis of the collected data would not be feasible due to different timestamps referring to the same event.

Channel management. To reduce cross-technology interference between devices operating in the same ISM band, a proper channel management among heterogeneous devices is necessary. If the used channels are communicated among all devices in close proximity, the channels could be adjusted to reduce cross-technology interference. Hence, we are also planning to build a proper channel management that uses *X-Burst* to communicate the used channels among different technologies.

Porting *X-Burst* to another hardware platform. To show the portability of *X-Burst*, we will port it to another hardware platform supported by the Contiki OS, e.g., to the Tmote Sky which uses an IEEE 802.15.4 capable radio (TI CC2420).

Optimizing the throughput. Due to the low measurement granularity of Contiki's *rtimer* and the inaccurate duration measurement, the achieved throughout of *X-Burst* is not optimal. A better throughput could be achieved by implementing a better measurement of energy burst durations. Furthermore, reducing the time between transmitting two successive data packets would also increase the throughput of *X-Burst*. Additionally, developing an encoding scheme tailored to *X-Burst* could also increase the data rate. Since the duration of bursts increases with the value of the transmitted data, i.e., the higher the value, the longer the duration of the burst will be, the throughput strongly depends on the transmitted data. Hence, developing an encoding scheme that minimizes the value of a byte, i.e., that minimizes the bits which are one, would have a positive effect on the data rate.

Splitting CTC messages. In case the assigned CTC time slots are very short, larger messages can only be sent or received by changing the *priority* of *X-Burst*, which will violate the normal behavior of the operating system. Another possibility would be to allow fragmentation of messages, so that each fragment can be sent in a very short time. A receiver has to reassemble each received part correctly and reconstruct the original message.

Appendices

Appendix A

Wireless Technologies

The used wireless technologies ZigBee (IEEE 802.15.4) and BLE are described in more detail below.

A.0.1 ZigBee

ZigBee is a standard for low-rate WPANs that builds up on the IEEE 802.15.4 physical radio specification that defines the MAC and physical layer of the network stack. ZigBee is responsible for the higher two layers, the network and application layer. It is an open standard and was defined by the ZigBee Alliance. ZigBee is designed for low-cost, low-power battery-operated devices that does not require a high bandwidth. Compared to other WPANs, e.g., BLE or WiFi, it has a very low complexity. The standard supports different network topologies such as mesh, tree, star or peer-to-peer networks, whereby usually a multi-hop mesh network topology is used.
ZigBee operates in the unlicensed ISM bands including the 2.4 GHz (global), 868 MHz (Europe) and 915 MHz (USA & Australia) bands. Usually the 2.4 GHz band is used which is separated into 16 channels (11-26) where each channel has a bandwidth of 2 MHz and is 5 Mhz apart of the next channel. This can be seen in figure A.1.
The standard delivers low latency communication and achieves data rates between 20 kbit/s (868 MHz band) up to 250 kbit/s (2.4 GHz band). It can handle networks with thousands of nodes that are distributed over a large area. Since each node can be separately addressed from the Internet, ZigBee can also participate in the IoT. It also supports low duty cycle, which allows even battery operated devices to have a very long lifetime. The communication range is restricted to the environment characteristics and power output. Typical distances are between 10 and 100 meter.
Uses for ZigBee are low-power low-bandwidth monitoring and controlling applications such as home automation or medical device data collection.

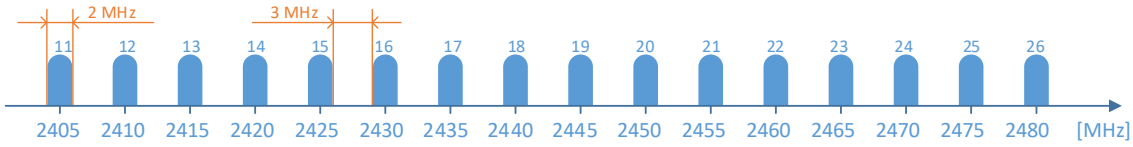


Figure A.1: Channel separation of ZigBee.

A.0.2 Bluetooth Low Energy

BLE, or sometimes also marketed as Bluetooth Smart, is a standardized ultra-low-power wireless technology for short-range WPANs. Compared to Classic Bluetooth, BLE is designed for low-power battery operated devices with limited hardware resources. BLE and Classic Bluetooth are not interoperable. It supports star (piconet) and mesh network topologies, thus it can also be used for multi-hop communications. BLE operates in the unlicensed 2.4 GHz ISM band which is separated into 40 channels (0-39) with a bandwidth of 2 Mhz. The channel separation can be seen in figure A.2. Three out of those 40 channels are so called advertisement channels (37, 38, 39) and are used for device discovery, connection establishment or data broadcasting. Therefore a very fast connection setup is possible compared to Classic Bluetooth. The remaining 37 channels are data channels used for bidirectional communication between two already connected devices. An adaptive frequency hopping scheme is used to counteract interference problems. BLE achieves data rates up to 1 Mbit/s by a maximum transmission power of 10 mW. Since BLE was designed with focus on a very low power consumption, BLE devices can operate for years by only using a coin cell battery. This is achieved by switching the transceiver off as long as possible. The widespread use of BLE and the fact that it can handle networks with a huge number of devices makes it also a good participant in the IoT domain. The communication range is in the scope of a few tens of meters, which strongly depends on the power output and the environmental characteristics.
Typical applications for BLE are low-power monitoring and controlling within a short communication range. It is mostly used in the area of healthcare, automotive, sport and home automation.

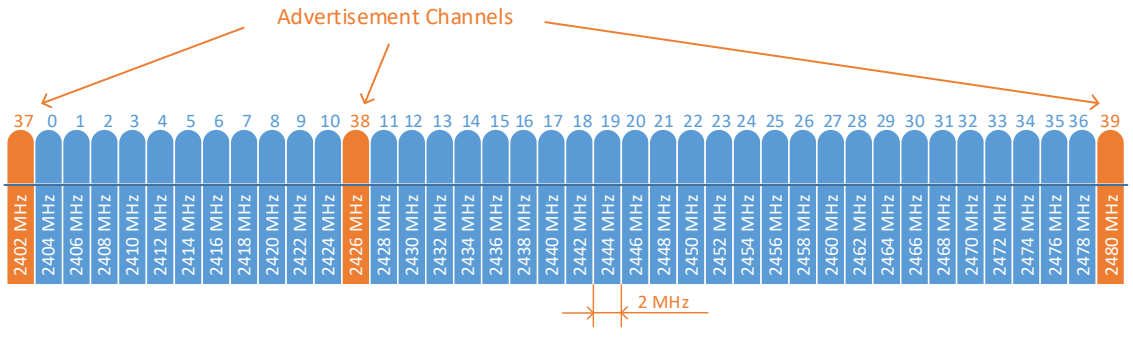


Figure A.2: Channel separation of BLE.

Appendix B

Hardware

The hardware used for this thesis is described in more detail below.

B.0.1 Texas Instrument multi-standard CC2650 LaunchPad

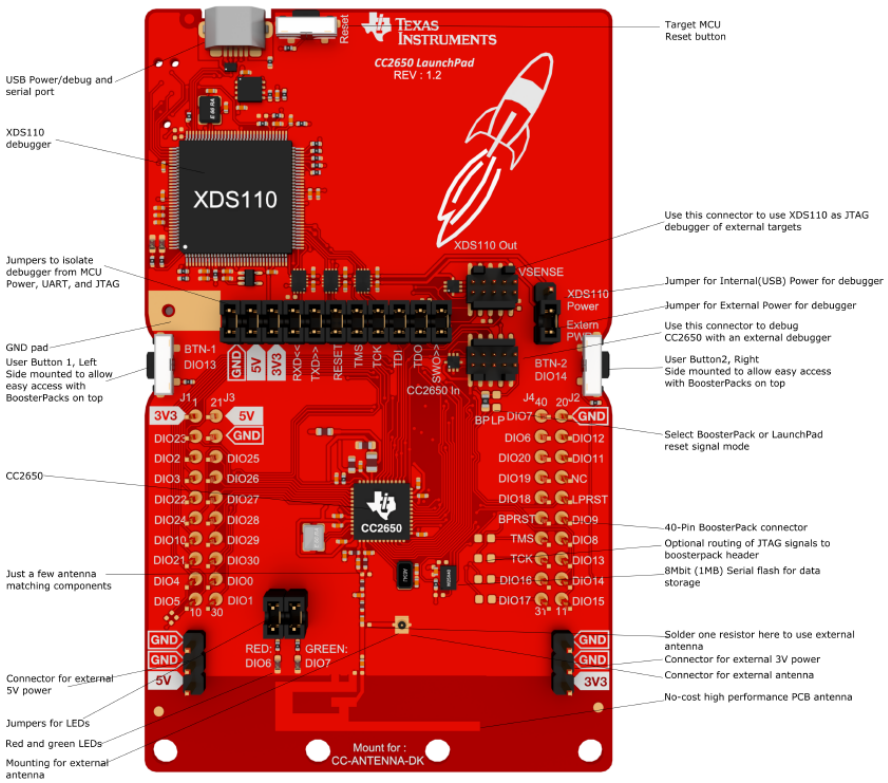


Figure B.1: Texas Instrument LaunchPad[1].

[1]Figure taken from: `http://pablocorbalan.com/img/cc2650_lp_explained.png`

The Texas Instrument LaunchPad (figure B.1) is an ultra-low-power development platform. It uses the CC2650 wireless microcontroller unit that operates in the 2.4 GHz ISM band. Due to its very low current consumption, the CC2650 provides excellent battery lifetime even on a small coin cell battery. As wireless technologies BLE and IEEE 802.15.4 (ZigBee) are supported. The main processor is a 32-bit ARM Cortex-M3 which runs at 48 MHz and provides 128KB of flash memory and 20KB of RAM. The IEEE 802.15.4 MAC and the BLE controller are embedded into a ROM and partly running on a separate ARM Cortex-M0 processor. This design allows a very effective power management as the packet reception and transmission can run autonomously from the rest of the system. Additionally this also improves the overall system performance and frees up flash memory for the application.
The Launchpad provides a good range of peripherals including a 12-bit analog to digital converter, four general purpose timer modules and many more. Over the air firmware updates are also supported. It is closely related to the Texas Instrument Sensortag which is one of the three primary supported platforms of the Contiki OS. Thus the Lauchpad also came with a good support of the Contiki OS.
Because of the support of both wireless technologies BLE and ZigBee and the good integration into Contiki, the Lauchpad was chosen for this thesis.

Appendix C

Additional Definitions of Energy Bursts

In the following, alternative mappings from hex values to energy burst durations for *X-Burst* are shown. Furthermore, the throughput that would be achieved by using the respective mapping is shown. In particular, the upper and lower bound as the average throughput are determined. For the calculations, the *preparation time* of the TI CC2650 Launchpad was used (IEEE mode: 220 μs, BLE mode: 400 μs).

Table C.1 shows the fastest possible mapping for ZigBee. Unfortunately this mapping can not be used with the TI CC2650 LaunchPad because of a to low measurement granularity of the RSSI and the duration of energy burst. Nevertheless, it could be used with a different hardware.

Hex value	Duration	*rtimer* ticks	Payload bytes$_{ieee}$	Payload bytes$_{ble}$
0x0	192 μs	13	0	1*14 (14)
0x1	224 μs	15	1	1*18 (18)
0x2	256 μs	17	2	1*22 (22)
0x3	288 μs	19	3	1*26 (26)
0x4	320 μs	21	4	1*30 (30)
0x5	352 μs	23	5	1*34 (34)
0x6	384 μs	25	6	2*19 (38)
0x7	416 μs	27	7	2*21 (42)
0x8	448 μs	29	8	2*23 (46)
0x9	480 μs	31	9	2*25 (50)
0xA	512 μs	34	10	2*27 (54)
0xB	544 μs	36	11	2*29 (58)
0xC	576 μs	38	12	2*31 (62)
0xD	608 μs	40	13	2*33 (66)
0xE	640 μs	42	14	2*35 (70)
0xF	672 μs	44	15	2*37 (74)

Table C.1: Alternative mapping for *X-Burst* - Mapping A.

IEEE: upper bound: 9.71 kbit/s, average: 6.13 kbit/s, lower bound: 4.48 kbit/s
BLE: upper bound: 6.76 kbit/s, average: 4.81 kbit/s, lower bound: 3.73 kbit/s

The mapping of Table C.2 could be used for the communication from ZigBee to BLE. Since the TI CC2650 LaunchPad in IEEE mode uses a non-instantaneous measurement of the received signal strength, the measured durations will always vary a little bit. Hence, using this mapping would result in a lot of decoding errors due to the small gap between the different durations. Nevertheless, in the case that the TI CC2650 Launchpad will only act as a receiver in BLE mode, this mapping could be used because of the more accurate measurement of the received signal strength when using the LaunchPad in BLE mode. We do not chose this mapping because we wanted a common mapping among both technologies.

Hex value	Duration	*rtimer* ticks	Payload bytes$_{ieee}$	Payload bytes$_{ble}$
0x0	192 μs	13	0	1*14 (14)
0x1	256 μs	17	2	1*22 (22)
0x2	320 μs	21	4	1*30 (30)
0x3	384 μs	25	6	2*19 (38)
0x4	448 μs	29	8	2*23 (46)
0x5	512 μs	34	10	2*27 (54)
0x6	576 μs	38	12	2*31 (62)
0x7	640 μs	42	14	2*35 (70)
0x8	704 μs	46	16	3*26 (78)
0x9	768 μs	50	18	3*29 (86)
0xA	832 μs	55	20	3*31 (94)
0xB	896 μs	59	22	3*34 (102)
0xC	960 μs	63	24	3*37 (110)
0xD	1024 μs	67	26	4*30 (118)
0xE	1088 μs	71	28	4*32 (126)
0xF	1152 μs	75	30	4*34 (134)

Table C.2: Alternative mapping for *X-Burst* - Mapping B.

IEEE: upper bound: 9.71 kbit/s, average: 4.48 kbit/s, lower bound: 2.92 kbit/s
BLE: upper bound: 6.76 kbit/s, average: 3.73 kbit/s, lower bound: 2.58 kbit/s

Table C.3 shows a possible mapping that achieves, in theory, a slightly higher throughput compared to the actual one used in this thesis (Table 5.10). It can be used for the TI CC2650 LaunchPad in IEEE and in BLE mode.
We do not chose this mapping because it is not as robust as the actual one used in this thesis due to the smaller gaps between the durations.

Hex value	Duration	*rtimer* ticks	Payload bytes$_{ieee}$	Payload bytes$_{ble}$
0x0	192 μs	13	0	1*14 (14)
0x1	288 μs	19	3	1*26 (26)
0x2	384 μs	25	6	2*19 (38)
0x3	480 μs	31	9	2*25 (50)
0x4	576 μs	38	12	2*31 (62)
0x5	672 μs	44	15	2*37 (74)
0x6	768 μs	50	18	3*29 (86)
0x7	864 μs	57	21	3*33 (98)
0x8	960 μs	63	24	3*37 (110)
0x9	1056 μs	69	27	4*31 (122)
0xA	1152 μs	75	30	4*34 (134)
0xB	1248 μs	82	33	4*37 (146)
0xC	1344 μs	88	36	5*32 (158)
0xD	1440 μs	94	39	5*34 (170)
0xE	1536 μs	101	42	5*36 (182)
0xF	1632 μs	107	45	6*32 (194)

Table C.3: Alternative mapping for *X-Burst* - Mapping C.

IEEE: upper bound: 9.71 kbit/s, average: 3.53 kbit/s, lower bound: 2.16 kbit/s
BLE: upper bound: 6.76 kbit/s, average: 3.05 kbit/s, lower bound: 1.97 kbit/s

Bibliography

[1] Nest Labs, "Nest." `https://nest.com/`, April 2018.

[2] Nuki Home Solutions, "NUKI." `https://nuki.io/en/`, 2017.

[3] H. N. Saha, S. Auddy, S. Pal, S. Kumar, S. Pandey, R. Singh, A. K. Singh, P. Sharan, D. Ghosh, and S. Saha, "Health monitoring using Internet of things (IoT)," in *2017 8th Annual Industrial Automation and Electromechanical Engineering Conference (IEMECON)*, pp. 69–73, Aug 2017.

[4] SAMSUNG, "Samsung health." `https://health.apps.samsung.com/`, 2015-2017.

[5] Waymo, "The Google self-driving car project." `https://waymo.com/`, 2015-2017.

[6] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, pp. 22–32, Feb 2014.

[7] M. Erol-Kantarci and H. T. Mouftah, "Energy-efficient information and communication infrastructures in the smart grid: A survey on interactions and open issues," *IEEE Communications Surveys Tutorials*, vol. 17, pp. 179–197, Firstquarter 2015.

[8] LineMetrics GmbH, "LineMetrics - Asset monitoring." `https://www.linemetrics.com/en/`, 2018.

[9] Gartner Inc., "Gartner report." `https://www.gartner.com/newsroom/id/3598917`, 2017.

[10] G. Zhou, J. A. Stankovic, and S. H. Son, "Crowded spectrum in wireless sensor networks," in *Proc. of the 3rd Workshop on Embedded Networked Sensors (EmNets)*, 2006.

[11] U. Wetzker, I. Splitt, M. Zimmerling, C. A. Boano, and K. Rmer, "Troubleshooting wireless coexistence problems in the industrial Internet of things," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, pp. 98–98, Aug 2016.

[12] Z. Chi, Y. Li, H. Sun, Y. Yao, Z. Lu, and T. Zhu, "B2W2: N-way concurrent communication for IoT devices," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, pp. 245–258, ACM, 2016.

[13] K. Chebrolu and A. Dhekne, "Esense: Communication through energy sensing," in *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking*, MobiCom '09, pp. 85–96, ACM, September 2009.

[14] S. M. Kim and T. He, "FreeBee: Cross-technology communication via free side-channel," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pp. 317–330, ACM, 2015.

[15] W. Jiang, R. Liu, L. Liu, Z. Li, and T. He, "BlueBee: 10,000x faster cross-technology communication from Bluetooth to ZigBee," Novmeber 2017.

[16] "IEEE standard for low-rate wireless networks," *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pp. 1–709, April 2016.

[17] C. A. Boano, T. Voigt, C. Noda, K. Römer, and M. Zúniga, "JamLab: Augmenting sensornet testbeds with realistic and controlled interference generation," in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pp. 175–186, April 2011.

[18] SIG Bluetooth, "Specification of the Bluetooth system - covered core package version: 4.1.." `https://www.bluetooth.com/specifications/bluetooth-core-specification/legacy-specifications`, December 2013.

[19] A. Dunkels, "The ContikiMAC radio duty cycling protocol," tech. rep., SICS, December 2011.

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

[21] M. Spörk, C. A. Boano, M. Zimmerling, and K. Römer, "BLEach: Exploiting the full potential of IPv6 over BLE in constrained embedded IoT devices," November 2017.