Defining a minimal BLE stack

A Bluetooth Low Energy implementation in Rust

Master's thesis in Computer Systems and Networks

Francine Mäkelä
Johan Lindskogen

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Defining a minimal BLE stack

A Bluetooth Low Energy implementation in Rust

Francine Mäkelä
Johan Lindskogen

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Defining a minimal BLE stack
A Bluetooth Low Energy implementation in Rust
Francine Mäkelä
Johan Lindskogen

Supervisor: Olaf Landsiedel
Examiner: Philippas Tsigas

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An illustrative representation of a Bluetooth stack.

Typeset in LaTeX
Gothenburg, Sweden 2018

Abstract

Today, Internet of Things (IoT) has spread to many everyday situations. The smart devices constituting IoT can be everything from your smartwatch, to components of your car or nodes collecting environmental data in a building. It is not uncommon for these devices to be powered using limited sources, such as batteries. This means that they have to be conserved with their energy.

One way for these devices to communicate is via Bluetooth Low Energy (BLE), a wireless protocol specifically designed to consume less energy than the classic Bluetooth protocol.

In this master's thesis, we aim to find the minimal BLE stack required for a device to advertise its existence and for it to enter a connection with another device and keep that connection alive. To check whether our definition holds we present a design and implementation of it in Tock, an operating system for embedded devices. As Tock is written in the programming language Rust, so is also our implementation.

The evaluation of the implementation includes two parts. The first part is a validation of the behaviour of a device running our code. We perform different tests, each focusing on a particular behaviour that is required from the device. Next, we conduct performance tests to measure the reliability, power consumption and timing of the device.

Our evaluation shows that the implementation fulfils the requirements, even though the performance tests reveal that it is not optimised. As the implementation is a mirror of the design, which in turn is a possible description of the definition of the minimal stack, we conclude that our definition states precisely what is required to fulfil the goal of "establish and keep a connection".

Keywords: BLE, Bluetooth low energy, IoT, Rust, Tock, Embedded systems.

Acknowledgements

We want to thank our supervisor Olaf Landsiedel for excellent mentoring and support during the project. Amit Levy, Niklas Adolfsson and the other people from the Tock community for answering our Tock-specific questions. We also want to thank our examiner Philippas Tsigas for his support. Lastly, we want to thank the people who act as peer reviews of this report and opponents at our presentation.

Francine Mäkelä & Johan Lindskogen, Gothenburg, June 2018

Contents

List of Figures

List of Tables

1
Introduction

Internet of Things (IoT) take more and more place in our lives, and not many can doubt its usefulness. Connecting smart devices, i.e., devices with the capability to collect data, making calculations with the data and communicate with other devices, opens up the possibility to simplify our lives in many ways. These devices can be used in critical applications, such as collision avoidance systems in vehicles, thereby increasing the safety in our everyday lives. Other applications are non-critical and only exists due to their convenience or the entertainment they provide. An example of this could be a refrigerator that informs us when we are about to run out of milk. For this connection to be possible the connected devices needs a way to communicate, via Wi-Fi or Bluetooth Classic for example. Both Wi-Fi and Bluetooth Classic have their advantages, but none of them is optimised for applications with tight constraints on energy usage [3, 4]. For example, devices that run on a coin cell battery cannot afford the relatively high energy usage of Bluetooth Classic [3]. A more suitable option for this setting is to use Bluetooth Low Energy (BLE). BLE is the result of an initiative by Nokia, which was adopted by the Bluetooth Special Interest Group, and eventually included in the Bluetooth specification in late 2009 [5]. This new communication protocol can run for years on devices powered by coin cell batteries [5], which makes the technology an appealing choice for devices with constraints on energy consumption.

The limited amount of energy available is not the only thing that makes embedded devices tricky to work with; memory capacity can also be an issue. This, of course, puts pressure on the programming language to either be efficient in this regard or to grant the programmer more control. For this reason, C is still a widely used programming language among embedded systems [6], but there are new competitors; one of these is Rust. The language is developed with systems programming in mind and therefore tries to handle issues that often arise from these kinds of systems. Rust promises to challenge the efficiency of C but also gives the programmer guarantees of memory safety and no race conditions. Furthermore, Rust is designed to catch errors at compile-time rather than at run-time, which makes debugging more manageable, particularly since it might not be easy to do in embedded systems.

These attractive attributes of Rust interested researches at Stanford University when they were to start their implementation of an operating system for embedded systems that would aim for stability and security [1]. The project, named Tock, is described in Section 2.3.

1.1 Problem Statement

Until last year, Tock did not have a BLE stack. During the spring term of last year, Nilsson and Adolfsson started implementing a BLE stack as a part of their master's thesis [7]. As a result, Tock got support for advertising and passive scanning. For applications that have more advanced use-cases, this will not always be enough, as these applications need to keep track of with whom they are communicating.

This would prompt the need for implementing support for connections, but we find the definition provided in the Bluetooth specification too broad for simple applications. Therefore, we provide a new definition with no more than what is needed to establish and keep a connection.

1.2 Contribution

In the process of solving what is defined in the problem statement (Section 1.1), it is relevant for us to validate that our definition covers enough functionality, and therefore we choose to implement it for Tock.

As performance is of great importance in embedded systems, we also describe our experience of using Rust for the implementation and argue whether the programming language is a reasonable choice for implementing BLE. Furthermore, we compare the performance and energy consumption of our final implementation to that of existing BLE implementations in C.

This master's thesis contributes with the following:

- A description of a "minimal BLE stack", containing the parts necessary for establishing a connection.
- A design and implementation of the "minimal BLE stack" for a device with limitations on the available energy.
- A discussion whether Rust is suitable for implementing BLE.
- A comparison of both energy efficiency and performance to that of an implementation in C.

1.3 Delimitations

The goal of this thesis is to present the minimal stack; therefore we will exclude multiple types of advertising that do the same thing and are therefore redundant. This is the motivation for why we chose only to take the simpler of the flow of establishing a connection into consideration.

Our minimal stack also only cover allowing incoming connections, since this is the role with lesser responsibilities and that is more commonly assumed by the type of device supported by Tock. The communication is not encrypted, as this adds additional complexity.

1.4 Key Results

This thesis produces a definition of a minimal stack. We argue that it only needs the two lower layers of the standard BLE stack to establish an keep a connection. We provide a Rust implementation that demonstrates this for a fact and evaluate the power consumption and other measurements. By choosing Rust we get the opportunity to explore this promising young language in the context of embedded programming, and in particular, using it for building a BLE stack.

A summary of our evaluation follows:

- The reliability of the BLE stack is similar to that of Mynewt.
- Measurements of the power consumption shows that our implementation in Tock has an overall lower energy consumption during advertising, even if it is higher during peaks. Over a connection interval, the power consumption is slightly higher than that of Mynewt.
- In a connection, the time from when the device receives a packet to when it replies is within the range of $\pm$ 1 µs from what is stated in the Bluetooth specification.

1.5 Thesis Outline

This thesis is divided into eight chapters that are structured to successively give the reader a deeper understanding of the project. Chapter 2 starts by describing what hardware we use, the Rust programming language, Tock and Bluetooth Low Energy. In Chapter 3, we present two competing operating systems for embedded devices, both implemented in C, and we make a short comparison between them and Tock. We define our interpretation of a minimal BLE stack and present our design in Chapter 4, while Chapter 5 describes how it is implemented. This chapter also covers a discussion of Rust's suitability for a Bluetooth stack. After that, we present what method we use for evaluation of the system and our results in Chapter 6. The thesis ends with a summary and conclusions of some key parts of the thesis in Chapter 7.

2
Background

This chapter introduces elements that are needed to get a clear picture of our project, including the hardware that is used during this thesis and the advantages of using this particular hardware. It introduces Rust and its features so that the reader can understand how they are used in Tock when the operating system is presented in the following section. Lastly, we describe the parts of BLE needed to follow along when reading about the design in Chapter 4.

2.1 The Hardware Platform

This thesis uses a nRF52 development kit from Nordic Semiconductor as target development hardware. The development kit supports a few different hardware chips, but the one we are using is the nRF52832 System-on-a-Chip.

The nRF52832 is a chip with a modern 32-bit ARM Cortex-M4F processor [8]. Together with the CPU, the chip also contains 64 kB of RAM and a flash memory of 512 kB and a 2.4 GHz Bluetooth Low Energy antenna.

It compares well to other chips with regards to energy efficiency [9]. Energy efficiency is of great importance for embedded devices as they often require long times of unattended operation.

Compared to the previous version of the development kit, nRF51, the nRF52 has considerably improved the start-up time, or "ramp-up time", of the Bluetooth antenna and the radio chip that is controlling it. The ramp-up time has gone from 140 µs down to 40 µs [2], which is a substantial improvement and it is especially useful in Bluetooth when the replies sometimes need to be sent within short windows of time, see Section 2.4.

To further help with these timing constraints, the nRF51 and nRF52 provide the developer with a set of hardware shortcuts. These shortcuts can be used to bind events and tasks together in the hardware by programming what decision should be taken beforehand, at an instant during non-timing-critical periods, and thereby avoiding a potentially slow decision to be made during periods when a reaction is expected within microseconds. Both boards also have support for calculating the checksum (CRC) of every packet sent or received at the hardware-level which also eliminates a slow operation during critical periods.

In Table 2.1, the nRF52 is compared to other devices to give a better understanding of its computing power.

Device	CPU speed	RAM
nRF51	32 MHz	32 kB
nRF52	32 MHz	64 kB
OnePlus 5	2.45 GHz	6 GB
MacBook Pro 2017	2.9 GHz	16 GB

Table 2.1: *Comparison of nRF hardware with modern, everyday devices.*

2.2 Rust

We will do the programming part of this thesis in the relatively young language, Rust. The language is only three years old, version 1.0 was released 2015, but has gained a following for its unique approach to memory management and type safety.

2.2.1 Embedded Rust

Rust in the context of embedded programming is still in its early stages, but this modern language has many advantages to older ones traditionally used for embedded programming, like C.

The Rust language and its compiler is more powerful, catches more errors at compile-time, and thereby prevents run-time errors. This is especially convenient for embedded programming since run-time errors are hard to debug without designated debugging equipment.

Rust also promises zero-cost abstractions, that allows the programmer to add type-safety checks and structure the code in a more straight-forward way with no additional run-time cost. Most of these abstractions will be flattened into highly optimised code with high performance at compile-time.

2.2.2 Ownership

Rust has a strong concept of ownership; there can only be one owner of a particular variable at a time. Every time a variable is assigned to another, or when a function gets called with a variable, the ownership of that value is handed over. Changing ownership means that all previous references get invalidated.

In particular, the rules are that there can either be many immutable references, i.e. with read access only, to a variable, or one mutable reference, i.e. with read and write access. The borrow checker is the feature responsible for enforcing these restrictions.

This feature prevents race conditions and is one of the more powerful features in Rust since race conditions are generally hard to track down in other programming languages.

2.2.3 Enums and Associated Values

Another useful language feature of Rust are enums and associated values in enums. An enum represents a discrete type, a type with only a finite set of values.

An example of this is the `Option` type in Rust. It is used to represent the presence or absence of a value, by its variants `Some(value)` or `None`. `Option` accomplishes this by using associated values, i.e. the value that is present in the `Some(value)` variant. A variable of type `Option` can only unwrap its associated value if it has a value. With this construct, Rust circumvents references to null values, or null pointers, a common problem for many other languages.

2.3 Tock

Tock is a new operating system that is designed to provide a safe multiprogramming environment for software development on embedded devices [1]. In particular, the problems Tock is trying to solve can be summarised as follows:

Dependability - Embedded system might need to run for a long time without interference from a human user as they might be placed in locations where they are not easily reached, or that their user interface is limited.

Concurrency - If I/O tasks are scheduled concurrently on an operating system the microcontroller can spend more time in its sleeping state. Less energy is needed this way [10].

Efficiency - Embedded devices have much lower memory capacity than a personal computer. Therefore caution must be taken on how the memory is shared between different parts of the system, such as between the kernel and user-space applications.

Fault isolation - Failure in one part of a system should not cause other parts to fail as well.

Loadable application - Users can install applications with no need to reprogram the entire kernel.

Through a combination of the type- and memory-safety features of Rust, modern hardware safety features, and the system architecture of Tock, the operating system manages to provide all five of these items.

2.3.1 System Architecture

The architecture of Tock separates the code of the operating system into capsules and processes, depending on whether they are a part of the kernel or user applications, respectively. The capsules lie in the kernel and are written in Rust to take advantage of the memory- and type-safety features of the programming language. Each capsule consists of a Rust module, which includes an instance of a struct, its associated methods and static variables that all make up the module itself.

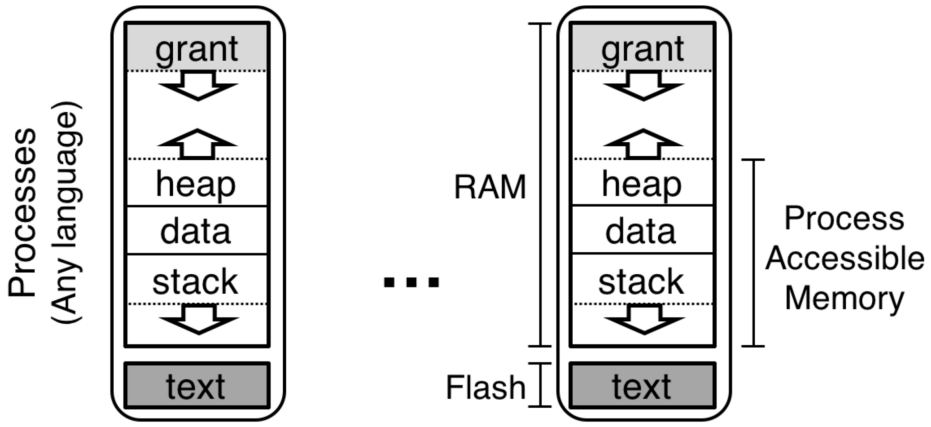

Figure 2.1: *The memory of a process in Tock. Figure is from the paper "Multiprogramming a 64 kB Computer Safely and Efficiently" [1].*

Capsules A capsule can either be considered trusted or untrusted, depending on what type of capsule it is and what its responsibilities are. Most of the capsules are considered untrusted and are therefore not allowed to read from or write directly to the memory of another capsule or process. This constraint is enforced by Rust's inherent isolation of modules, which prevents external modules from accessing private methods, as explained in Section 2.2. Only capsules that need direct access to the hardware are considered trusted. These capsules make use of Rust's keyword `unsafe` to perform necessary operations that are usually prohibited by the memory-safety model, such as allocating memory at specific addresses.

Processes For user programs to be able to be loaded dynamically at run-time, an alternative separation than capsules is needed. In Tock, this separation is called a process. The processes keep user programs physically separated in the hardware rather than by language features like the capsules do. As a benefit, the programmer can implement applications for Tock in any language of their choosing since the processes do not rely on Rust's module separation as a safety feature.

A process can communicate with the kernel via a system call interface, and other processes via inter-process communication (IPC), in which a process can share a memory region with other processes.

Each of the processes is assigned a region of memory which is separated from the memory of the kernel and the other processes. As in other operating systems, the process keeps its stack, heap and other process-related information in this memory region (see Figure 2.1). However, unlike other operating systems, the memory contains an additional component known as a grant. The grant is used by a capsule that needs to allocate memory during run-time to serve a request from a process.

Even though the grant resides in the address space of a process, the process cannot access this memory region. If this would be possible, the process could tamper with the capsule's data, which would break the security model. The capsules, on the other hand, can read and modify data inside the grant through a limited

System call
`command`
`allow`
`subscribe`
`memop`
`yield`

Table 2.2: *The five types of system calls supported by Tock*

programming interface. By leveraging Rust's type-system, Tock can ensure that references created inside of a grant cannot be moved out of the grant.

2.3.2 Scheduling

Tock uses an event-driven kernel scheduler where the events originate from either system calls, which are sent from the processes, or from interrupts, which are sent asynchronously by the hardware. A capsule communicates with another capsule by directly calling its public functions or by sharing memory. All capsules share a common stack and are scheduled cooperatively by the kernel, which means that every task runs until its completion. As a consequence of this, long-running capsules will degrade the performance of other capsules.

The scheduling of processes differs from the scheduling of capsules. Since the processes have separate stacks they can be scheduled preemptively, that is, running concurrently. Tock is using the simple but effective round-robin scheduling algorithm. Preemptive scheduling allows a process to perform lengthy executions without causing any adverse effects for other processes.

2.3.3 System Calls

There are five types of system calls in Tock which provides an interface for processes to communicate with the kernel. The system calls are listed in Table 2.2.

The `command` system call is used by a process to request the kernel to perform a particular task. It takes an integer as a parameter which decides what task the kernel should perform.

The system call `allow` is used to ask the kernel to perform a task which needs a more complex parameter than just an integer. In a sentence, it allows the kernel access to a part of the process's memory space.

If a process wants to be notified when a specific event occurs in one of the capsules, it can use the `subscribe` system call. Upon calling `subscribe` the process passes along a callback function as a parameter, which will be called when the event in question occurs.

The `yield` system call is required to serve queued callbacks. If the callback queue is empty when `yield` is called the calling process is blocked. As soon as a callback is placed on the queue, control is returned to the registered callback function. After the callback function returns, the process resumes execution at the

place where the `yield` call was issued.

There is also a system call known as `memop` which handles memory boundaries for processes.

2.4 Bluetooth Low Energy

Bluetooth Low Energy (BLE) supports wireless communication in the 2.4 GHz ISM (Industrial, Scientific and Medical) band, which is divided into 40 channels [11]. Three of these channels (37, 38 and 39) are the primary advertising channels. They are used by devices that have not established a connection with another device. The primary advertising channels are spread over the band and are separated by several other channels. The reason for this is to minimise interference from one advertising channel to another. The remaining 37 channels are used by devices as secondary advertising channels, and for communicating data after a connection has been established, they are therefore called data channels. By applying frequency hopping, BLE reduces the risk that congestion on one channel might cause packets to get lost. Reducing the risk is necessary since BLE shares the band with other protocols, for example, classic Bluetooth and Wifi.

The BLE stack is divided into three parts: application, controller and host. The latter two are parts of the Bluetooth core system. Both the controller and the host block is further divided into smaller blocks, or layers, with different responsibilities. The controller communicates directly with the hardware and has more critical timing constrains that it needs to fulfil, while the host does not have such constraints. Optionally, the two layers can be separated by a layer known as host controller interface (HCI) which passes information between the upper and lower parts of the stack.

The controller contains two layers: the link layer and the physical layer. The following two subsections focus on describing the role of the link layer; different roles a device can assume; and how a connection is established.

2.4.1 The Link Layer

The link layer is responsible for keeping track of the current operation state of the BLE host; it also keeps track of what actions are allowed in each state. Essentially, the link layer state machine has the following five states: [12]:

Standby state - the device is idle, i.e. it is not sending, nor receiving, any packets. This state can be reached from any of the other four states, and a device in the standby state can move to all other states, except the connection state.

Advertising state - the device is sending packets to advertise its existence. It might also listen for and reply to requests from other devices. A device in this state is called an advertiser.

Scanning state - the device is listening for packets sent by advertisers, and might also reply to them. A device in this state is called a scanner.

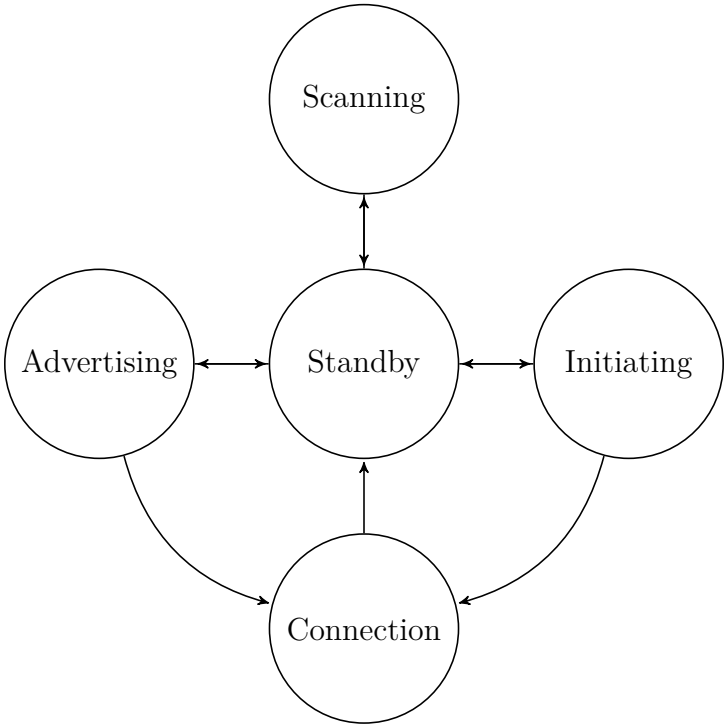

Figure 2.2: *Bluetooth Low Energy Link Layer state machine and all the permitted transitions.*

Initiating state - the device wants to create a connection with another device, and are therefore listening for packets sent by this/these device/devices. A device in the initiating state is referred to as an initiator.

Connection state - the device has been connected to another device. This state can be reached from both the advertising and the initiating state. Within this state, a device can have one of two roles: if entered from the initiating state the device will operate as a master, else it will operate as a slave. The device which assumed the master role is responsible for selecting the different parameters used in the connection, and to communicate them to the slave.

The link layer can have several state machines to keep track of multiple parallel modes of operation, such as connections with multiple peers at the same time. Each of the state machines can, in turn, only be in one of the states. At least one of the state machines must support either the advertising state or the scanning state. Figure 2.2 shows the different states of the link layer state machine, and how it can move between them.

Packet Format

Depending on which state the link layer is in it will send either advertising channel packets or data channel packets. Devices that have established a connection communicates via data channel packets, but up till that point they are using advertising channel packets. Both of these packets have the same format and, as can be seen

Figure 2.3: *The link layer packet format.*

in Figure 2.3 has four mandatory fields. The fields we will focus on are the access address field and the PDU (Protocol Data Unit) field.

The access address field contains a four-byte long address. In the advertising state, when advertising channel packets are used, all packets carry a predefined advertising access. After two devices have established a connection, they change their access address to identify what packets belong to their connection and use that address in the access address field. It is the responsibility of the initiating device to generate a random access address that is only used within that specific connection, and communicate this address to the peer device before the connection is established in a designated connection request packet.

What the PDU field of a link layer packet contains depends on if the packet was transmitted on an advertising channel or a data channel. Both of the PDU types carries a header and a payload, but what these fields contain differs. While the advertising channel PDU holds information about the advertiser, the contents of the data channel PDU can be both empty, to serve as an acknowledgement of receiving another packet, or contain requests or replies to requests of data during the connection.

Advertising State

As an advertiser, the device follows a sequence of sending advertising packets on all or a subset of the three advertising channels before it becomes idle for a predefined amount of time. If the device broadcasts advertisements of the scannable type, it will listen for and respond to scan requests from other devices before changing to the next channel. Each sequence of sending advertisements is called an advertising event. The time from the start of one advertisement event to the beginning of its following event is called advertising interval and is a period of between 20 ms and about 10 000 s.

An advertiser does not necessarily have to advertise on all of the three primary channels, but each advertising event has to be sent on the channels in ascending order, i.e. if channel 37 and 38 are used the device shall advertise on channel 37 before channel 38. Furthermore, if the advertiser receives a connection request or a scan request, the advertising event might end earlier.

An advertising packet can be of one of several types and indicates what kind of requests the advertiser will respond to if any. The advertising events can be one of the following types: scannable; connectable; connectable and scannable; and non-scannable and non-connectable. The event type which is both connectable and scannable can only be undirected, but the remaining three exists as both directed and undirected events. If undirected events are used, anyone is allowed to reply to the advertisements, while directed adds a restriction on who is allowed to send a

Advertising event type	Directed	Undirected
Scannable	x	x
Connectable	x	x
Scannable & connectable	-	x
Non-scannable & non-connectable	x	x

Table 2.3: *The different advertising event types, and whether they can be directed and/or undirected.*

request to the advertiser.

Most of the advertising types require the time spent on each channel to be no longer than 10 ms. For some of the advertising types, it is also required that the advertiser can start to receive packets 150 µs after the last advertisement was sent. If the received packet was a request, the advertiser must also be able to start transmitting a response within 150 µs from the time when the reception of the request was finished.

Scanning State

Just like the advertiser, a scanning device jumps between the primary advertising channels, but its timing requirements differ from that of the advertiser. Two parameters have to be specified to the scanner: a scan window which tells how long the scanner will listen on a channel, and a scan interval which is the time from the start of one scan window to the start of its succeeding scan window.

A scanner can either be passive and only listen for advertising packets, or be active and respond to scannable advertising PDUs. The use-case decides which of the modes should be used. If the scanner wants the extra information that the advertiser offers it should send a scan request back. The response to the scan request will be sent in a scan response PDU. Until the scanner receives a scan response from the advertiser, it will assume the packet was lost and will continue to send scan requests as replies to the subsequently received advertisements.

As an active scanner contributes to congestion on the channels, it is required to use a backoff procedure to reduce the risk of collisions between packets. The Bluetooth specification does not specify an exact algorithm for the procedure, but the algorithm has to respect that the advertising channels are a shared medium.

Initiating State

An initiator acts very much like a scanner, but with the intention of connecting to another device. To create a connection, an initiator sends a connection request as a response to a connectable advertising PDU.

The connection request sent by the initiator contains parameters that will be used by the two devices to keep the connection alive, such as information about what channels they are going to communicate over and in what order. When the advertiser receives a connection request, and the connection is created, the initiator assumes the master role and the advertiser the slave role detailed in the next section.

2.4.2 Establishing a Connection

A connection can either be considered created or established. Directly after two devices have entered the connection state as described in Section 2.4.1, the connection is said to be created. Not until a device receives a packet from its peer device is the connection said to be established. As previously mentioned, the initiator of the connection has the master role and is responsible for timing in the connection. The other device is known as a slave.

The connection is driven by so-called connection events. In a connection, data PDUs are sent instead of advertising PDUs. The devices jump between channels from the channel map field specified in the connection request PDU sent by the initiator. How this changing of channels is done are specified by a channel selection algorithm [12]. Within a connection event, the master and the slave are taking turns of who is sending and who is receiving. The exchange of packets within a connection event takes place on the same channel. If any of the two devices want to send more than one packet during a connection event, it indicates so by setting a bit called MD (more data). Not until after the connection event is finished, the master and the slave moves to the next channel.

Every connection event starts with the master transmitting a packet, and ends when neither of the two devices wants to send anything more, but no later than 150 µs before the next connection interval [12]. The length of the connection interval is specified in the connection request sent by the master and is used to synchronise the master and the slave after they have changed channel between two connection events. As the master initiates each connection event, the least that can happen during a connection event is the transmission of that single packet. The slave is required to transmit a packet whenever the master sends something, except if the CRC match fails two times in a row within the same connection event. If that happens, the event should be closed. The master is always allowed to send another packet, no matter whether the CRC is correct or not, as long as the slave replies.

A data PDU header contains two bits which are used for acknowledgement of data packets. One bit is used as the sequence number (SN) of the packet. The other bit indicates what SN a device expects to get next from its peer, and is therefore called next expected sequence number (NESN) [12]. As the NESN bit can represent no more than a single packet, only one packet at the time can be acknowledged, and therefore a device has to continue to resend a packet until the peer acknowledges it.

Several things affect the performance of a connection, and it might get lost without any warning. To avoid being trapped in an already lost connection, both the master and the slave keeps track of how long time has passed since they last received a packet from their peer. If that number gets to large, the device will end the connection and enter the standby state.

3
Related Work

Tock has many competitors in the field of operating systems aimed for embedded devices [13, 14, 15]. For Tock to compete with these, it should have advantages in at least some areas when it is compared to other operating systems. In this chapter, we briefly present two other operating systems and compare their BLE implementation with that of Tock.

3.1 Apache Mynewt

Apache Mynewt is an open source operating system for IoT devices which, along with other protocols, supports communication via BLE [16]. The BLE implementation, named NimBLE [17], supports the full BLE stack. The implementation even includes an HCI layer, which allows the user to exchange the host or controller part of NimBLE with hardware or software provided by another vendor [18].

Even though Mynewt as an operating system supports several different boards [19], the BLE controller supports only Nordic Semiconductor's nRF51 and nRF52 [20], as only drivers for these has been implemented. On the other hand, the host implementation can run on any of the boards supported by Mynewt.

The link layer in NimBLE can adopt all of the five states specified in the Bluetooth Specification [18]. An advertiser can be either connectable or non-connectable, and in the same way, a scanner can either have the intention of connecting to an advertising device (act as initiator) or just listen for packets. Further, NimBLE allows a device to act in several roles concurrently, and a connectable advertiser can be in several connections at the same time.

The controller of a BLE stack contains, as mentioned in Section 2.4, a link layer and a physical layer. In NimBLE, the code for the link layer is grouped into several files, as what should be done depends on the current state of the link layer state machine [21]. For example, functions that are unique for an advertiser is grouped together in its own file. One example of how this is used can be seen in Listing 1. Here, a function that is common between the link layer states is called when the radio has started to receive a packet. This function uses a switch statement to determine in what state the link layer is in and then calls its counterpart in the corresponding link layer.

Apache Mynewt also provides some example projects, which allow a user to play and learn how to use NimBLE [17]. In the evaluation of our implementation in Section 6, we compare against the example project called 'bleprph', which is a basic implementation of a BLE peripheral.

```
void rx_start() {
    switch (link_layer_state) {
        case STATE_CONNECTION:
            conn_rx_start();
            break;
        case STATE_ADVERTISING:
            adv_rx_start();
            break;
        case STATE_INITIATING:
            init_rx_start();
            break;
        case STATE_SCANNING:
            scan_rx_start();
            break;
    //Code has been removed
}
```

Listing 1: *When the radio has started to receive a packet, the link layer in NimBLE switches on the link layer state to see where the packet should be forwarded. This code is a simplified version of the real NimBLE code [22].*

Discussion

NimBLE is a complete implementation of the full BLE stack, which means that all of the host layer and the controller layer is supported. In contrast, our implementation of BLE in Tock provides only a controller layer, as Section 4 discusses. As a result, NimBLE has a wider range of use cases. For example; a layer called L2CAP is responsible for the fragmentation of packets, meaning that NimBLE can send larges packets than our implementation supports.

One similarity between the two BLE implementations is that both of them opted to write their own BLE stack for nRF52 to fit the operating system, rather than using the driver provided by Nordic Semiconductor.

The NimBLE controller supports both nRF51 and nRF52, while we choose to only focus on the nRF52 board. This choice comes partly from an advise we got from one of the researches at Stanford, who told us that they are going to phase out the older version of the board.

Another difference between NimBLE and our BLE stack is of course that while NimBLE is implemented in standard C, we are using Rust as the programming language.

NimBLE has yet another advantage over our BLE implementation: it supports concurrent connections. Even though we are not actively working to include support for this, we still strive to structure the code in such a way that it could be relatively easy included in future versions of the BLE stack.

3.2 Contiki

Contiki is an open source operating system for embedded devices which was developed to meet the hard constraints of sensor networks [23]. Sensor nodes need to exchange and forward data within the network, which Contiki enables by supporting communication protocol such as IPv4, IPv6 and 6LoWPAN. The operating system allows for downloading and removing applications at run-time, both manually and over the air [23], and due to the low power requirements, it can run wireless relay nodes on batteries [13].

Like Tock, Contiki supports several hardware devices, among them the nRF52 [24].

Contiki does not have its own full BLE implementation for nRF52; instead, the operating system relies on the SDK and a "softdevice" provided by Nordic Semiconductor.

Michael Spörk [25] presents the work of creating a stack which supports IPv6 over BLE. This would enable devices with tight energy restrictions to communicate over the internet via just a BLE interface. The idea takes advantage of the fact that IPv6, and IP in general, is not dependent on the upper or lower layers of the protocol stack [26]. By substituting the IPv6 link layer with the BLE link layer, also two devices using different link layer protocol can communicate using IPv6.

The idea of having IPv6 running on top of BLE is not new, it is based on an RFC [27, RFC7668], and there exist other implementations of it already. Spörk suggests that the importance of his work lies, partly, in that it is open source, in contrast to other existing implementations. Allowing anyone to read and use the code is valuable for further research, as one does not have to start from scratch if one wishes to continue the work within the area.

In his thesis, Spörk presents a general design for the stack which fits the architecture of Contiki, but that can be used together with any hardware running the operating system. The design by Spörk introduces an additional layer BLE-HAL, that is not directly a part of the network stack of Contiki. This layer is responsible for communicating with the BLE controller provided by the hardware, or if it is not fully implemented, the BLE-HAL adds the needed functionality. This means that the layer is hardware dependent and that what exactly is included in the BLE-HAL depends on the hardware used and must be adjusted when porting to new hardware. To the rest of the network stack, the layer provides functions similar to the HCI layer in a BLE stack.

Discussion

Our BLE implementation in Tock and the one in Contiki differs in the way that they approached supporting different parts of the BLE stack. We have aimed at implementing the BLE controller, i.e., the lower layers, while Contiki relies on this part to be handled by the softdevice provided by Nordic Semiconductor. Therefore a direct comparison between the two is challenging to make, as their BLE implementation for nRF52 is not open source.

In relation to our work, the interesting part of Spörk's implementation is the

BLE-HAL layer. Even though this layer might not add a massive amount of functionality for the nRF52, as the board can be loaded with a BLE controller, it is interesting in other aspects. His work shows us how a fully implemented BLE controller can be used, both by other BLE host implementations as well as for other purposes than was originally thought of.

Spörk also mentions in his master's thesis that one reason why the functions of the BLE-HAL mimics those of a BLE HCI is that it will make the integration of the BLE-HAL easier the if the BLE controller of the hardware comes with an HCI layer. This layer is not mandatory in a BLE stack, and our implementation does not provide such a layer, but with this reasoning in mind, we can see the benefits of adding such a layer in the future.

4
Design

This chapter presents a high-level overview of our design, suited for the system architecture of Tock. It is further described as a possible implementation in the next chapter, Chapter 5. First, we describe our minimal stack, which goes into detail about what our minimal stack contains, what parts we chose to include or exclude and why we chose to do so. The system architecture section follows with an overview of how the different modules in our BLE stack relate to each other and how they communicate with each other. This chapter presents an answer to the first goal of this thesis, as Section 1.1 specifies, namely the goal to provide "A definition of a minimal BLE stack". How we choose to realise this definition into an actual system design is also presented in this chapter.

4.1 Basic Flow

In this master's thesis, we focus on enabling data exchange between two devices using Bluetooth Low Energy. The minimal stack is derived from what is needed to create, establish and keep a connection alive. For this task, a device has to perform the following steps:

1. Advertise its existence
2. Receive a connection request from another device
3. Receive and reply to data packets from that device

The first step is required as a device must receive an advertisement for a device before it is allowed to send a connection request to it. The second is what is required to create a connection, and the third is what is needed to establish and keep the connection.

4.2 Definition of a Minimal Stack

The Bluetooth specification states that the following is required from a minimal BLE stack:

> "A minimal implementation of a Bluetooth LE only core system covers the four lowest layers and associated protocols defined by the Bluetooth

> specification as well as two common service layer protocols; the Security Manager (SM) and Attribute Protocol (ATT) and the overall profile requirements are specified in the Generic Attribute Profile (GATT) and Generic Access Profile (GAP)." [28, p. 179]

To fulfil the basic flow in Section 4.1, we require no more than the two lower layers, i.e. the link layer and the physical layer. The physical layer is needed to send and receive data by communicating with the hardware. The link layer ensures that the device behaves according to the Bluetooth specification, as there are no guarantees that communication with other devices is possible otherwise. This includes making sure that the device holds the timing constraints, switches between transmitting and listening for packets at a proper time, to correctly interpret data received and also to have knowledge of what should be done depending on what packet was received.

The rest of the protocols mentioned in the minimal implementation from the specification are services that operate on top of an existing connection. The Security Manager is a service for secure pairing and encrypted connections [29]; both are superfluous for the most basic device communication. Attribute Protocol and Generic Attribute Profile are service discovery protocols used to identify what services a device provides [29]. Service discovery is less critical if it is known beforehand with what device the connection will be established.

From this argumentation it is clear that the scope of this master's thesis does not cover everything in the minimal stack outlined in the Bluetooth specification. Instead, our definition is the following:

> "The minimal Bluetooth Low Energy stack is defined as an implementation in which only a Link Layer and a Physical Layer are present."

Discussion

The definition differs quite a bit from the Bluetooth specification, as Section 4.2 describes. Fundamentally, the difference is that the purpose of the BLE stack in the two definitions is not the same. The Bluetooth specification presents what has to be included in the stack to enable meaningful communication over BLE. Our definition, on the other hand, describes a BLE stack which will only be able to create and establish a link layer connection.

In real-world use cases, the described BLE stack is not enough to satisfy the requirements for communication with other devices. A specific example of this is that most real-world data transfers would need some encryption, which is implemented by the security manager. We exclude parts like ATT, GATT and GAP as we see them as higher-level protocols that are more suitably implemented as a user-space library, and is therefore outside the scope of this thesis.

From this, we conclude that our definition of a BLE stack indeed is a minimal one for its use case.

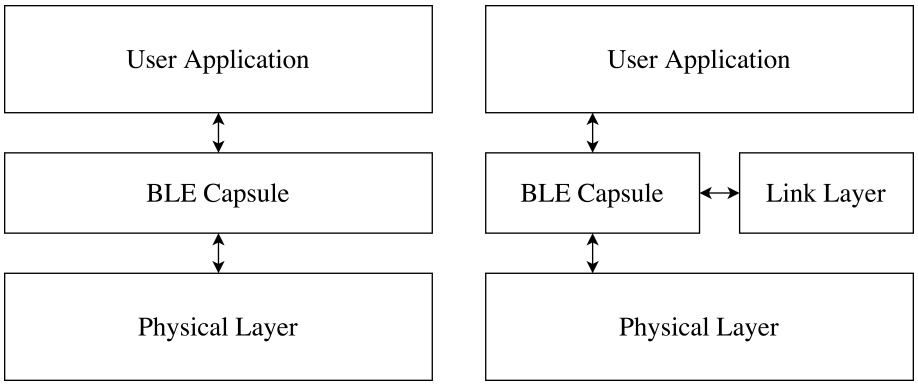

Figure 4.1: *Comparison of Tock BLE stack before (left) and after (right) our design was implemented.*

4.3 Overview of System Architecture

Tock already had some support for BLE at the start of this thesis, even though it was very limited. A device could either act as an advertiser that sends an advertisement on each of the advertising channels or as a passive scanner that listens for advertisements, i.e., without responding to any of them. The implementation consisted of three main components: BLE user library, BLE capsule and BLE hardware module [7]. Together these three parts constitute the BLE device driver in Tock. The user library hides the actual Bluetooth implementation from the user application, and provide an easy way to start and stop advertising or scanning. The hardware module is designed towards a particular piece of hardware and is responsible for tasks like instructing the radio when it should transmit or receive, and to set hardware timers, among others. The BLE capsule contains the logic that ties the other two parts together and makes sure that the device acts according to the Bluetooth specification.

With the new set of features that comes with a Bluetooth stack that supports connections, the previous design was no longer flexible enough. As a result, we have to refactor and extend the previous design to make it more flexible and to meet the tight timing requirements of BLE.

Expanding upon the stack-like design, we decide to add a layer to closely match the Bluetooth stack, and also further divide different parts of the BLE device driver's responsibilities into different modules. By keeping all changes inside the BLE kernel code, the external interface is not affected. This means that the BLE capsule's system call interface towards the user application stays unchanged.

4.4 Layer Responsibilities and Communication

The BLE capsule acted previously as a sort of coordinator; it was the entering point for a user application, and it communicated with the hardware module. In the new design, these responsibilities remain, with an addition: the BLE capsule is also responsible for the communication with a new layer called link layer, as Figure 4.1

shows.

Even though the work of this thesis has come to increase the amount of logic in the hardware module, most of the functionality related to the exchange of packets is hardware independent, and so we have chosen to put this general logic in the BLE capsule and the link layer. It is the link layer's responsibility to make decisions on the protocol level, to instruct the radio to transmit or receive or when it should change the channel.

4.4.1 Link Layer

The idea of introducing a link layer is to encapsulate new, link layer specific, functionality within it. Further, some decisions that currently is a part of the BLE capsule are moved into this layer as well. This way, the BLE capsule have less knowledge of how data should be interpreted and moves towards having an even more coordinator-like role.

The link layer knows how the system should act as a response to all the different events from the hardware, as it knows the link layer state machine, and what is an acceptable response depending on the current state. As the hardware module only communicates with the BLE capsule, the BLE capsule has to forward any request or decisions aimed at the link layer. These requests are often about whether or not a packet received by the radio shall be read, or how and when the radio shall transition between sending and listening. The latter is an example of a task that was previously enclosed by the BLE capsule. Further, the link layer knows how to translate data in a connection request into meaningful information.

4.4.2 Hardware Module

The hardware module is what typically is referred to as the physical layer in Bluetooth. Just as before, this is the only part of the BLE device driver that has to be implemented differently depending on what hardware is used. The hardware module is the layer that is responsible for reading and writing data to the hardware registers. This layer avoids making decisions unrelated to the hardware, rather it forwards them to the BLE capsule and awaits further instructions. This communication is further described in Section 4.5.

4.4.3 BLE capsule

The BLE capsule is as mentioned above a coordinator which forwards data between the link layer and the hardware module, as well as it is the entering point for user programs. Apart from this, the BLE capsule also holds other logic, such as handling when an application wakes up after it has been idle, prepares packets and forwards these to the radio when they should be sent.

It also keeps an internal representation of user programs, and it is also used for storing the state related to individual user applications, as Tock requires them to be able to be scheduled preemptively. Some examples of this state are the advertising data, current radio channel and current link layer state.

The information in the representation of a user program is sometimes required for specific decisions, and as it could be somewhat unpractical to always forward this information to the link layer, there are occasions when the BLE capsule makes decisions in matters that could be seen as more link layer related.

Further, this part of the BLE stack also handles callbacks to the user program when a subscribed event occurs.

4.4.4 Discussion

To realise our definition of a minimal stack; it must be designed in a way which matches how capsules are usually implemented in Tock. Therefore only minor changes are done to the design of the already existing stack.

When implementing the hardware module, the task is to enable the hardware and software to communicate with each other through a hardware-independent interface. The feature set and hardware specific details can be very different between hardware manufacturers, and even between different models of hardware from the same manufacturer. As an effect, all hardware specific code fits best in its own layer close to the actual hardware.

In Figure 4.1 we see that the link layer and the BLE capsule is placed at the same level of abstraction. The reason for this is that the BLE capsule is responsible for making decisions for the hardware module, while also facilitating the communication between the hardware module and the user application. Instead of putting more of the Bluetooth specific knowledge into the BLE capsule, which would break the single responsibility principle, we opted to extract the Bluetooth specific decision-making to another module at the same level of abstraction.

Our design comes with one disadvantage: the decision-making logic in the link layer requires information stored in the internal representation of an app. Due to this, the app-specific information has to be passed down by the BLE capsule to the link layer every time a decision has to be made. An alternative solution would be to duplicate the data, i.e., store the information in both the representation of the app and the link layer. We dismiss this solution as it introduces a risk of inconsistency between the values, no single source of truth.

4.5 Hardware Module Events

In our design, we want to keep the Bluetooth-specific knowledge separated from the hardware controlling logic, which is why we have different layers in the stack presented above. Two problems that arise when doing it this way are how to forward the required parameters to other layers and how to get the result back.

Before the start of our thesis there was already a bit of functionality like this, a `TransmitClient` and a `ReceiveClient`. A different module could register as one or both of these clients to receive events when an advertisement was transmitted or received. It is a common pattern in Tock to use a capsule that registers as a listener to and commands the hardware component.

In our design, we extend this idea with more events (see Figure 4.2) to allow the

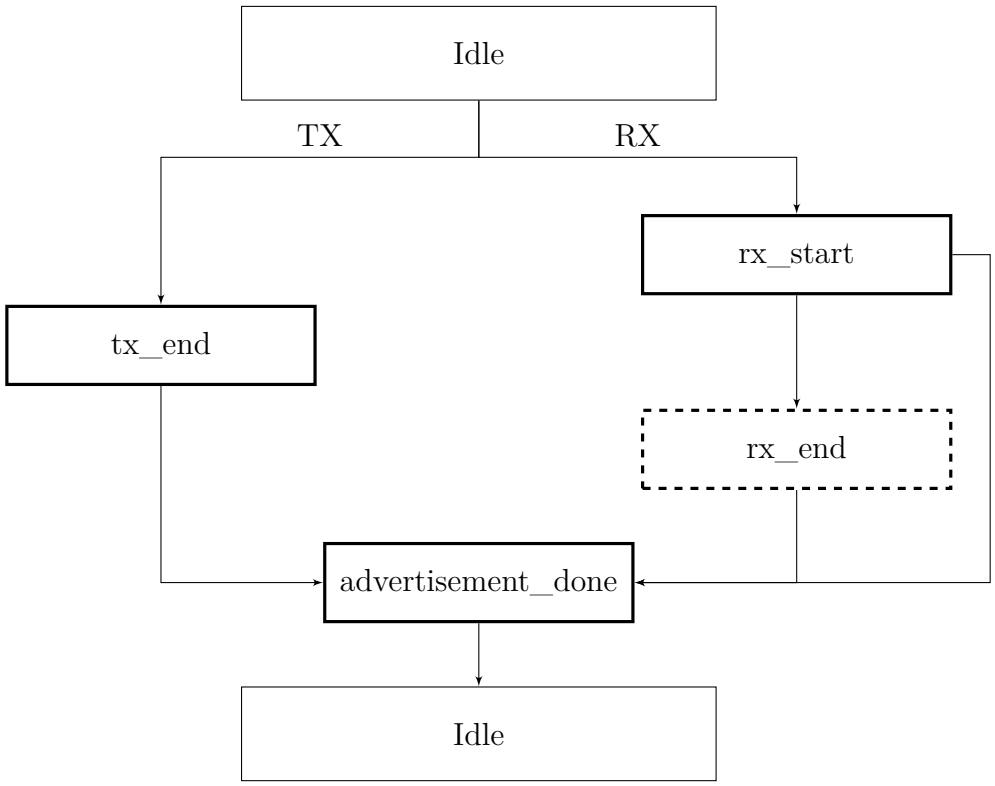

Figure 4.2: *A flow chart showing the order of different events emitted by the hardware module.*

BLE capsule and link layer to control the hardware module more effectively. These events differ from those generated by the hardware. To emphasise this difference in this text our events are written using lower case, while those generated by the hardware are written in upper case. On each of these events, a method call is issued to the `TransmitClient` or `ReceiveClient`, which passes along information about the event. At the end of the method, the next action to be taken is returned depending on the input parameters. This means that the whole BLE device driver is driven forward using hardware interrupts.

4.5.1 tx_end

This event fires just after the radio has transmitted a packet. The BLE capsule can now decide if it should listen for responses and for how long.

4.5.2 rx_start

When the radio has received the header of a packet, it needs to decide if it should bother with reading the rest of the packet or not. The radio will therefore trigger this event and pass along the packet header to the BLE capsule for it to validate the packet and decide if it should be read.

4.5.3 rx_end

If a packet is decided to be read, this is the next event to be fired. Now the radio has read the whole packet into a buffer and passes it upwards in the stack for it to be processed further.

4.5.4 advertisement_done

A timeout in RX, skipping a packet or finishing processing a packet - these all lead here, where the radio needs to know what to do after the TX/RX is complete on a single channel. Often a TX is scheduled on the following channel, or, if the advertisement interval is complete, the radio is put to sleep, and the advertisement interval timer is started and set to continue advertising again after some long period.

4.5.5 Discussion

The BLE device driver is designed as a state machine driven by hardware interrupts. This creates a rather simple way of making sure that the device behaves as expected as only one interrupt is triggered at the time and also that the behaviour of the device follows a is more or less predefined path.

When the device is sending no interrupt is needed after the TX has started, as the device already has prepared the packet to be sent. For RX, having an interrupt after the header of the packet is received could save some valuable time, as packets that are not of interest are quickly discarded. This behaviour allows the device to move to the next state of the state machine quickly.

The radio has to be disabled every time we want to switch between TX and RX. Therefore, the two paths are merged at the end of the state machine, to then have the radio become idle.

5
Implementation

This chapter starts by briefly presenting an overview of the implementation, and continues with what is new in the hardware module and how this differs from the old implementation. Next, the chapter presents the implementation of the link layer, which is followed by describing the communication between the different layers. Further, the chapter specifies a timeout mechanism used to avoid that a device is waiting in vain. It also presents the *connection driver*, a module that is part of the BLE capsule and responsible for handle different aspects of a connection. The chapter continues with explaining how transmission and reception of a packet are done, focusing on the different hardware interrupts that the hardware module has to handle during these periods. Lastly, the chapter discusses useful parts of Rust that we are using in the implementation.

5.1 Overview

The system described in the previous chapter is realised with Rust modules, at least one for each of the layers in Figure 4.1, except for the user application. Additionally, in some cases where some values and functionality are tightly coupled, they can be grouped into their own module. These modules are then used to hold information that is related to each other, to make calculations on specific types of values, or to extract some piece of information from a larger buffer. This allows for other parts of the system to focus on the control flow of the system instead of knowing how data shall be read. By making this kind of separation, we also split the implementation into smaller modules that are easier to manage.

One example of when this is done is for connection-related data; it is link layer specific, but we see a clear separation of the connection specific logic and the rest. We feel that this motivates moving it to its own module, which is further described in Section 5.6. Another example is the PDU parser which, as the name suggests, encapsulates the knowledge about PDUs from the rest of the modules. Some examples are: translating the buffer sent over the air into a PDU struct, how to validate a PDU, and translation from protocol-specific details; like numbers and type ids, into our representation of a PDU type.

The communication between modules is done in different ways depending on what responsibilities the module has. Some modules communicate through an interface to decouple the layers from each other, some modules provide static functions, and some provide instance methods. The latter case requires that the calling code has access to an instance of the struct.

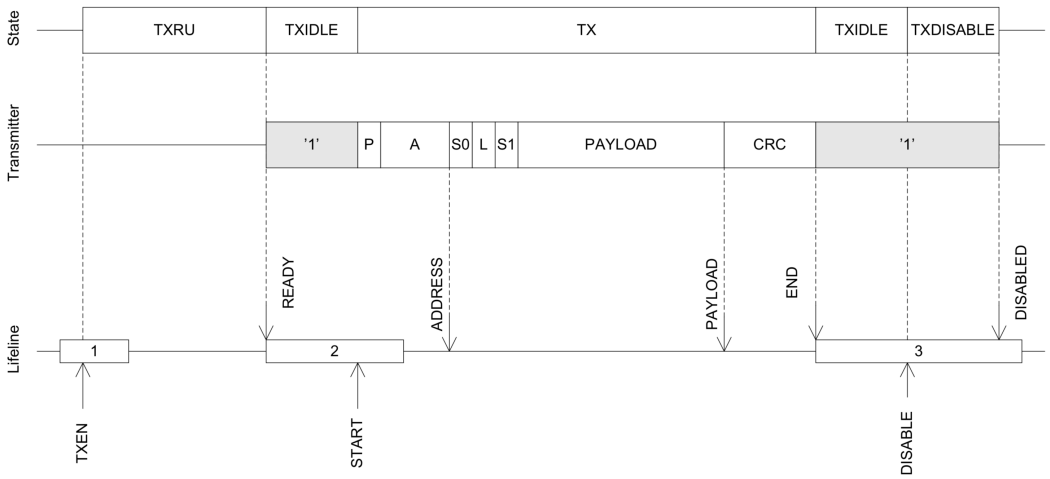

Figure 5.1: *The different events generated when the radio transmits a packet. When a packet is received, the same events are generated but with RX instead of TX. Figure is from "nRF52832 Objective Product Specification" [2].*

5.2 Hardware Module

The hardware module contains a function called `handle_interrupt`, which is the entry point of the BLE device driver after a hardware interrupt has occurred. In the implementation from last year by Nilsson and Adolfsson [7], this function called the BLE capsule if the radio was done either transmitting (TX) or receiving (RX) a packet. The implementation did not take into consideration what had caused the interrupt, only if it was related to transmission or reception of a packet. Further, the implementation also assumed that a device was either an advertiser which were neither scannable or connectable or that it was a passive scanner. This meant that a device was only sending or listening, not both, and the radio did not have to bother with timing issues or switching between the two states.

After an interrupt, the hardware module handed over the to the BLE capsule. From here, the BLE capsule could do one out of two things: either it could schedule another TX/RX on the next channel with a delay of 10 ms; or if it was on channel 39, it scheduled the radio to sleep for 300 ms.

Inspired by Apache Mynewt (see Section 3.1), we choose to look more into different kinds of events that can cause the radio to interrupt, to see how the hardware can be better utilised.

During the transmission and reception of a packet, the radio generates four events: READY, ADDRESS, PAYLOAD and END, as shown in Figure 5.1. The developer can choose to configure the hardware to generate an interrupt or not, for each of these events independently. In addition, since we cannot switch the radio from RX to TX mode directly when the antenna is active, we use the DISABLED event to perform the transition between the modes. This event is generated when the radio is no longer used and is a marker that the radio once again can be set up

```
1     pub fn handle_interrupt(&self) {
2         if regs.event_address.get() == 1 {
3             self.handle_address_event();
4         }
5         if regs.event_disabled.get() == 1 {
6             if self.state.get() == RadioState::RX {
7                 self.advertisement_client.timer_expired();
8             } else {
9                 self.handle_tx_end_event()
10            }
11        }
12        if regs.event_end.get() == 1 {
13            self.handle_rx_end_event();
14        }
15    }
```

Listing 2: *Part of the hardware module, simplified. This is the interrupt handler, which calls other parts of the hardware module that knows how to handle the events.*

for either TX or RX.

The hardware shortcuts, mentioned in Section 2.1, provide the developer with a way to chain these events. For example to utilise this information to make decisions whether the radio should read a packet or not.

5.2.1 Redesigning the Hardware Module

In the redesign, there are four different cases where the program enters the interrupt handler, with different behaviour for each of them. The first reason for the program to enter the function is due to a timeout, which is further described in Section 5.5. Next, two of the reasons are, as before, after the radio is done transmitting or has fully received a packet. The function which handles this interrupts is shown in Listing 2.

In addition to these three, if the radio is receiving a packet, it causes an interrupt as a response to an ADDRESS event, i.e. after it has finished reading the address field of the packet. This type of interrupt causes the hardware module to wait until the radio has read another byte, which is the header of the packet. The header contains the advertising type, and with this, the link layer has enough information to decide whether the hardware module shall continue to read the packet or not. The decision is based on whether or not the type of packet is valid in the current link layer state. If the packet proves to be of interest, the link layer instructs the hardware module to interrupt again when the packet has been read to the end; otherwise it ignores the remaining part.

Along with the change described above, the scheduling of packets has been moved into the hardware module. Originally, this used a timer in the BLE capsule but to optimising the response time of the board we want to delegate as many tasks

```
1  pub struct LinkLayer;
2
3  impl LinkLayer {
4      pub fn handle_rx_start() -> ReadAction {
5          match app.process_status {
6              Some(AppBLEState::Advertising) => match pdu_type {
7                  Some(ScanRequest) => ReadAction::ReadFrame,
8                  Some(ConnectRequest) => ReadAction::ReadFrame,
9                  _ => ReadAction::SkipFrame,
10             },
11             Some(AppBLEState::Connection) => ReadAction::ReadFrame,
12             _ => ReadAction::SkipFrame,
13         }
14     }
15     pub fn handle_rx_end() {}       // Handling end of RX event
16     pub fn handle_tx_end() {}       // Handling end of TX event
17     pub fn handle_timer_expire() {} // Handling on timeout
18 }
```

Listing 3: *Part of the link layer module, simplified. Line one shows the unit-like struct. After that are the functions implemented on the link layer struct listed. The first function gives an example of how the link layer uses the Rust match statement to figure out whether the radio should continue reading a packet or ignore the rest of it.*

as possible to the hardware.

The only timer that is not a responsibility of the hardware module is the interval timer between advertising events, i.e. after advertising on channel 39, that part is left to the BLE capsule to allow preemption of other running apps.

5.3 Introducing a Link Layer

One reason why we choose to introduce a link layer is to isolate the BLE protocol-specific logic from the rest of the BLE device driver. This is especially important as the code grows and would become unfeasible to work with otherwise.

The link layer's responsibilities are not exhaustive but still stretches over a wide range of functionality. It is the only part of the system that has a concept of how a BLE link layer should work, how a device should respond to different events and how the information sent in a connection request should be interpreted. To accomplish this, the link layer provides a function to be called after one of the following events has occurred in the hardware module: ADDRESS, RX_END, TX_END and timeout. The first three events are presented in Section 5.2, while the details around the third are explained in Section 5.5. A shortened and simplified version of the link layer module is presented in Listing 3. All four functions utilise Rust's `match` statement as an easy way of deciding what to do depending on some

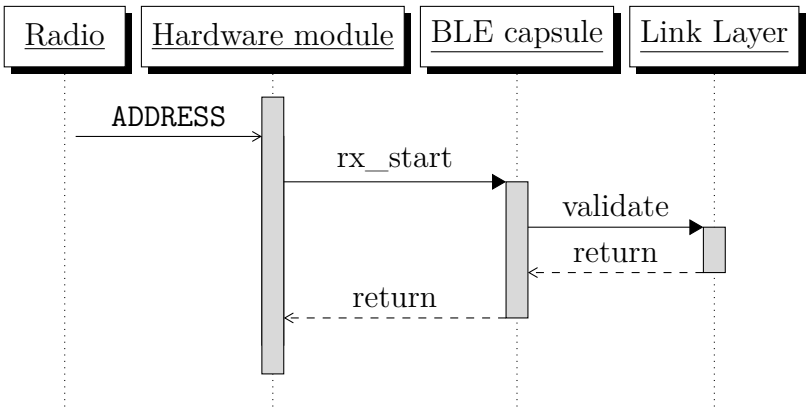

Figure 5.2: *An `ADDRESS` event causes the radio to generate an interrupt. This in turn triggers the process of validating the received packet.*

parameter.

Different packet types are of interest depending on the state of the device and therefore marked as valid in this particular state. It is the link layer's responsibility to make sure that the device handles only the valid packets. Therefore, after an ADDRESS event has caused an interrupt while the radio is in RX, a function called `handle_rx_start` is called (Line 4). At line 5, the function checks whether the device acts as an advertiser or is in a connection. In the first case, the link layer will only accept scan requests and connection requests (Line 7 and 8). After a connection is established, the slave is interested in every packet from the master, and therefore all packets are accepted. In the default case (Line 12), marked with an underscore, we ignore all packets.

When an END event is generated, the link layer is consulted of what should be done next. Two functions are provided for this (Line 14 and 15), as the systems should act in different ways when the radio is in RX and TX. They are called by the radio to decide what to do after a packet has been received or transmitted, respectively. Typically, the reception of a packet means that we need to respond in some way, while the transmission of a packet often requires switching to the next channel and scheduling the subsequent transmission.

After a timeout triggers an interrupt a particular function is called in the link layer (Line 16). From this function, the link layer instructs the hardware layer to reschedule after some time, where the time differs depending on whether the device acts as an advertiser or is in a connection.

As the link layer does not need to keep track of information such as the state of the device, it is implemented as an unit-like struct, i.e. a struct with no fields. Also, all of its functions are public to enable them to be called from the outside, which is the intended purpose of this module.

5.4 Transmit/Receive Flow

This section explains what happens during the transmission and reception of a packet. In particular, what shortcuts and interrupts the radio uses to perform a TX

and RX is explained.

5.4.1 Transmit Advertisement

Every time the advertising interval timer fires, a new advertising interval is started. During an advertising interval, the radio transmits and listens on each of the three primary advertising channels, 37, 38, and 39 in sequence.

Transceiving on one channel starts with that the BLE driver instructs the radio to schedule a TX. The radio reads the information passed down from the BLE driver and sets a shortcut between the timer and a shortcut called `TX_END`. This shortcut will start the radio in TX mode, ready to transmit. At this point, we also set two another shortcuts, `READY_START` and `END_DISABLE`. The first one will cause the radio to start transmitting the packet as soon as it is able to, i.e., when the radio has entered TX mode. The second will disable the radio as soon as the packet is transmitted. We also enable the `DISABLED` interrupt. What this all means is that the flow of a transmission goes from a `DISABLED` state, which is the idle state of the radio, through `READY`, which will start TX by the hardware shortcut and finally we will regain control again when we return to the `DISABLED` state, which now indicates that the TX is completed. We define this as the tx_end event, where the radio calls the BLE capsule to get further instructions. The BLE capsule will, in turn, call the link layer to ask for the appropriate action as defined by the BLE standard.

5.4.2 Receive Advertisement

The radio needs two parameters to start RX mode: delay and timeout. The delay defines when the radio should start to RX. The timeout will be relative to the delay point and is used as the maximum time the radio will listen for responses before moving on.

When the radio schedules a RX, just like for TX, we set `READY_START` and `END_DISABLE`, but for RX we also set `ADDRESS_BCSTART`. This shortcut will cause the bit counter module to start counting bits received by the radio, beginning from the `ADDRESS` event. This time, we also start the timeout and set it to force the radio into the `DISABLED` event. We can from here end up in two possible scenarios, the first case it that the timeout fires the `DISABLED` interrupt, causing the radio to stop listening for packets. The second case is that the radio starts receiving a packet and the `ADDRESS` interrupt fires. In the second case, we stop listening for `DISABLED` interrupts, as we do not want it to fire during the reception of a packet, causing the radio to ignore that packet.

If we get an `ADDRESS` event, we have to decide if the packet is valid in the current state. This is not something the radio can decide, so the decision is deferred to the link layer via the BLE capsule. However, the radio first needs to wait until it has read eight more bytes, the size of the packet header. This is where the bit counter module comes into play. Since the timing between the `ADDRESS` event and reading the full packet header is small enough, we busy-wait until the header is fully read into the memory buffer. This is where we emit our rx_start event. The packet header is also parsed to find out the type of the packet, which will decide if it is

valid in this state. If the packet is valid, the radio enables the `END` interrupt, wherein the full packet is finally read and passed up to the capsule for further processing in the rx_end event. Invalid packets will immediately disable the radio and trigger our advertisement_done event.

In the advertisement_done event, the BLE capsule decides what to do if no response was received to the last advertisement, often this results in the radio switching channel and a new TX to be scheduled. Although, it acts quite differently during a connection. In this case, as the slave role, the only thing to do is to RX at a specified time instant to read a packet from the master device. If a timeout is fired in this state, the next RX will not occur until the next connection interval. After it is scheduled, the radio is put into idle mode to save a bit of power.

5.5 Scheduling and Timeouts

The previous implementation had only one timer, which was used for scheduling communication events (as described in Section 5.2). In our implementation this timer is only used for a single purpose: to schedule the idle time, i.e. how long an advertiser should sleep after an advertising event.

Within an advertising event and after a connection is established, the scheduling is done by the hardware. The hardware module writes to a register `R1` the time at which the communication event should start. This register is then continuously compared with the current time by the hardware. As soon as `R1` matches the current time value, an interrupt is triggered.

A new feature in the implementation is the use of timeouts. Without this functionality, a device would listen forever if no packets are received. This is of great importance in a connection. If a slave fails to receive a packet from master it should handle this by retrying on the next channel. Without the timeout, the slave will stay on the same channel, which would cause the connection to terminate.

Scheduling is always done relative to a packet: either from the start or the end of a received packet or even the previously received packet. Between connection intervals, it is more convenient to set an absolute clock value to keep the hardware module unaware of the specific Bluetooth logic used to calculate this value.

We modelled these cases with an enum named `DelayStartPoint` (see Listing 4), which has variants that describe each of these different cases. Most of the variants hold an associated value that defines the delay value for when the next communication event should be scheduled. One exception is the `PacketEndBLEStandardDelay` variant, which does not hold a value. It represents a value of 150 µs, which is frequently used and we decided to make it a separate value. This is further detailed in Section 2.4.1 and Section 2.4.2.

An instance of `DelayStartPoint` is delivered to the radio when the next communication event should be scheduled. By using a `match` statement, the radio can find what variant of the enum it received and can calculate from what point in time it should schedule the start of the next event.

```
pub enum DelayStartPoint {
    PacketEndBLEStandardDelay,
    PacketStartUsecDelay(u32),
    PacketEndUsecDelay(u32),
    AbsoluteTimestamp(u32),
    PreviousPacketStartUsecDelay(u32),
}
```

Listing 4: *Definition of the `DelayStartPoint` enum, which is used for scheduling.*

5.6 Connection Driver

After a connection is created, the system defers to the connection driver to keep track of connection-related information. The module contains two structs: `DataHeader` and `ConnectionData`.

`DataHeader` represents the header of a data PDU in the connection, which means that it has the same fields as the data PDU. In particular, it has one field for the packet's sequence number and another field for the next expected sequence number, as Section 2.4.2 explains.

`ConnectionData` holds the state of the active connection and provides all functionality for keeping a connection alive. For example, it holds an implementation of the channel selection algorithm mentioned in Section 2.4.2. This algorithm is used to calculate the next channel after each connection event.

Another example is a function for updating the channel map after a channel map change event from the master. There is also a helper method for calculating whether or not the connection interval has ended. The slave should switch to the next channel if the master has nothing more to send on that channel, in this case, the interval has ended. As soon as this happens, the device can stop listening for packets and disable the radio after scheduling the next connection event, where it should start listening again.

5.7 Developing with Rust

During the development of the BLE device driver, we take inspiration from the Apache Mynewt project and the Contiki project. These are both implementations of BLE in C, which means that there is still much room for us to utilise the features and expressiveness of the Rust language to improve upon these previous solutions. In this section, we highlight some parts of the Rust programming language which has a significant influence on our choice of implementation and discuss how these are used.

```c
// (1) C
int handle_rx_end(int pdu_type)

// (2) Rust
fn handle_rx_end(&self, pdu: BLEPduType) -> Option<ResponseAction>
```

Listing 5: *Comparison between the signature of a* ***handle_rx_end*** *function in C and Rust*

5.7.1 Expressiveness

In C, the only types available to the programmer are different variations of integers or floating point numbers: `int`, `char`, `float`, `double`, `short` and `long`. The size of these types in bytes varies depending on the target architecture. The language has structs which are used to group values together but no way of defining custom algebraic data types, also known in some languages as enums. Without this feature, the expressiveness of the language is limited. An example of where this is very apparent is return values, which always requires the programmer to keep track of the implicit agreements between callee and caller, that is the function `T` being called and the function calling `T`.

From just reading function signature 1 in Listing 5 there are not many clues provided to the programmer of what values they might receive as the return value when calling this function, outside of it being an `int`. The compiler will not even prevent them from calling the function with an invalid value, simply because the type system in C is not expressive enough for the programmer to set proper constraints on the values passed to the function.

On the other side of the spectrum, Rust provides the programmer with types like `Option`, which signals that the value returned might be empty, and enums with associated values that lessens the cognitive load instead of requiring more.

5.7.2 Encapsulation

A feature of Rust that is very common in programming languages nowadays is encapsulation. Firstly, a struct can contain public and private fields. Public fields are accessible from the outside of the struct, and private are not. Member functions can either mutate the struct or get data derived from the struct, and Rust groups these together by the struct. All fields and methods are also private by default, which isolates data and behaviour from modules that do not and therefore should not know about them. Above this, every file in Rust is a separate module that encapsulates its fields, structs and functions.

In contrast to this, the encapsulation of C is less fine-grained. C has header files; wherein the programmer declares what will be exported from the file. But no member functions, which means that every function is in global scope as soon as it is included.

5.7.3 Mutability

Rust also has implicit immutability, which in simple terms means that we are not able to change the value of a variable by default. With the keyword `mut`, the variable is changed to be mutable and can be reassigned as many times as needed.

This concept also extends to references. To change a variable the current function needs to have a mutable reference to that variable. This means that there is a clear distinction between functions that are safe, in regards to mutability, to call and functions that are not. The programmer can easily know by just looking at the function signature if the function will be able to mutate its parameters or not.

5.7.4 Discussion

With these highlighted features, our design has a modern approach to the structure of modules and structs. For example, only making fields public when they need to and keep them private by default.

Exclusively at times when a variable needs to be mutable, we declare it to be mutable. This helps with reading and understanding the code since the programmer can be sure that the first assignment will be the only one. An example of this is if a private field of a struct is changed unexpectedly. The only way to change a private field of a struct is in a member function with a `mut` reference. Therefore the number of functions the programmer needs to check for erroneous code is significantly limited down to the number of functions with a mutable reference.

The improvements are not limited to developer experience, but also gives the compiler further information and context. The more the compiler knows about the program flow, the more optimisations it can perform to speed up the program or make it more memory efficient.

One could argue that C allows the programmer to perform these optimisations by hand, but this requires a massive amount of platform-specific knowledge that not every programmer is ready to delve into. We would also argue that it is hard, almost impossible for a programmer to rival the performance of code generated by a compiler.

6
Evaluation

In this chapter, we present how we evaluate our implementation. First, we present the tools which are used in the process. The chapter then continues with describing the tests we performed to verify that our implementation works according to the Bluetooth specification, as well as describing the results of the tests. We continue to describe how well our implementation performs, and detail what measurements we evaluated. Lastly, we present the results of the performance testing, compare it to Mynewt's BLE implementation and discuss the results.

6.1 Test Setup

Testing of embedded devices is non-trivial since one usually does not have an abundance of resources left over on the chip to run debugging code. The nRF52 development kit does have some extra debugging features on the board, but not for testing wireless communication. Instead, Nordic Semiconductor provides a special software [30] that can be loaded onto a nRF52 board and turns it into a Bluetooth sniffer that listens for and collects wireless traffic. The sniffer supports feeding this data into Wireshark [31], a well-known protocol and traffic analyser and debugging tool.

We use a BLE debugging software called LightBlue [32] as a peer device for testing of BLE connections. It listens for nearby devices in advertising state and starts a connection with our development board acting as a slave. This allows us to start a connection with our board and to repeat the process in a deterministic way.

In summary, we have two nRF52 boards: one with TockOS and our driver loaded onto it, the other one runs the Nordic BLE sniffer and is connected to a laptop running Wireshark. That same laptop starts a connection via the LightBlue software.

6.2 Validation

This section describes the task of verifying that our implementation meets what is stated in the Bluetooth specification. As such, we state six requirements that the implementation must meet and design tests with these requirements in mind. The behaviour of the device running our BLE stack is compared against these requirements.

Printing the current state and value of variables at different stages during program execution is a straightforward debugging technique. This is a time-expensive

operation, especially on hardware with slower IO, like the nRF52 board. Particularly during time-critical periods, a print operation can delay program execution to such an extent that we end up well outside the timing constraints required by the Bluetooth specification. Outside of the timing critical sections of execution, printing is very useful and is well-utilised by us.

There are also examples of when printing can be useful even in time-critical periods. One such example is when we want to debug why the device does not establish a connection with another device. In this case, a first step could be to print a message when the device receives a connection request. Since we are only interested in whether or not the connection request is received correctly and do not care for the establishing of a connection, we print upon reception of a connection request and ignore the deadline.

When debugging program flows in situations when the device must hold its deadlines, we use other means to monitor its behaviour. This is where we use the Nordic nRF Sniffer together with Wireshark, which parses and interprets the data sent from the sniffer and displays it in a human-friendly format. The sniffer also provides an option to follow a selected device, i.e. it shows only the traffic to and from the device, and tries to switch channel in the same way as the device it is following should do. However, in the same way as our device has the possibility of missing a packet due to wireless interference, the sniffer will inevitably miss some packets. This causes nothing but minor issues for us, but it has to be taken into account when analysing the traffic.

The requirements that our implementation must meet to fulfil our goals are the following:

1. The device advertises correctly.
2. The device replies to scan requests after 150 µs.
3. The device establishes a connection when a connection request is received.
4. The device replies to the master on the correct channel.
5. The device should change channel map if the master updates it.
6. The device listens for packets on a channel for no longer than the timeout.

All of the requirements stated above are further elaborated upon below.

The device advertises correctly

Our definition of advertising correctly is that an advertising device transmits advertising packets on all the primary advertising channels in increasing channel index order, i.e. 37, 38, 39. We verify this requirement by running the device as an advertiser and check that it indeed transmits on each channel in sequence. For this task, we use both the sniffer and printing, as we cannot trust any of them on their own: the sniffer might miss packets, while printing does not actually ensure that a packet was sent.

1400	0.000635	428	f0:00:00:0f:0f:f0	Broadcast	LE LL	38	0	TockOS	33	ADV_IND
1401	0.000492	428	f0:00:00:0f:0f:f0	Broadcast	LE LL	39	0	TockOS	33	ADV_IND
1402	0.313429	284459	f0:00:00:0f:0f:f0	Broadcast	LE LL	37	0	TockOS	33	ADV_IND
1403	0.000508	428	f0:00:00:0f:0f:f0	Broadcast	LE LL	38	0	TockOS	33	ADV_IND
1404	0.000407	428	f0:00:00:0f:0f:f0	Broadcast	LE LL	39	0	TockOS	33	ADV_IND
1405	0.309129	280461	f0:00:00:0f:0f:f0	Broadcast	LE LL	37	0	TockOS	33	ADV_IND
1406	0.000664	429	f0:00:00:0f:0f:f0	Broadcast	LE LL	38	0	TockOS	33	ADV_IND
1407	0.000618	428	f0:00:00:0f:0f:f0	Broadcast	LE LL	39	0	TockOS	33	ADV_IND
1408	0.211139	287450	f0:00:00:0f:0f:f0	Broadcast	LE LL	37	0	TockOS	33	ADV_IND
1409	0.000961	428	f0:00:00:0f:0f:f0	Broadcast	LE LL	38	0	TockOS	33	ADV_IND
1410	0.000752	427	f0:00:00:0f:0f:f0	Broadcast	LE LL	39	0	TockOS	33	ADV_IND
1411	0.316097	282446	f0:00:00:0f:0f:f0	Broadcast	LE LL	37	0	TockOS	33	ADV_IND
1412	0.000633	428	f0:00:00:0f:0f:f0	Broadcast	LE LL	38	0	TockOS	33	ADV_IND
1413	0.000746	428	f0:00:00:0f:0f:f0	Broadcast	LE LL	39	0	TockOS	33	ADV_IND
1414	0.000494	151	Apple_21:4a:67	f0:00:00:0f:0f:f0	LE LL	39	0		12	SCAN_REQ
1415	0.000479	149	f0:00:00:0f:0f:f0	Broadcast	LE LL	39	0	TockOS	33	SCAN_RSP
1416	0.312057	284188	f0:00:00:0f:0f:f0	Broadcast	LE LL	37	0	TockOS	33	ADV_IND
1417	0.000783	429	f0:00:00:0f:0f:f0	Broadcast	LE LL	38	0	TockOS	33	ADV_IND
1418	0.000715	428	f0:00:00:0f:0f:f0	Broadcast	LE LL	39	0	TockOS	33	ADV_IND
1419	0.307657	289433	f0:00:00:0f:0f:f0	Broadcast	LE LL	37	0	TockOS	33	ADV_IND
1420	0.000694	428	f0:00:00:0f:0f:f0	Broadcast	LE LL	38	0	TockOS	33	ADV_IND

Figure 6.1: *Screenshot from Wireshark showing when the device acts as an advertiser. The device, called "TockOS", is sending packets on all the primary advertising channels in the right order, as the green area shows. It also successfully replies to a scan request from another device, within roughly* 150 µs*, as the red area shows.*

Advertising may not be a new functionality but is still needed to create a connection. As we edit large chunks of the code, it is essential to make sure that we maintain the existing functionality.

When our device advertises, its behaviour meets our definition of advertising correctly. In Figure 6.1 we can see that the device indeed advertises on channel 37, 38, 39, and in that order.

The device replies to scan requests after 150μs

The second requirement includes that when an advertiser receives a scan request from any other device, it shall respond with a scan response within 150 µs.

The scan response is allowed to contain the same data as the advertising packet, with an exception for the part that indicates what type of packet it is. This allows us to create a simple test. We make the scan response contain the device name, and remove it from the standard advertising data. The reason is that if the advertising packet and the scan response look the same, there is no way of telling whether a scanning device has received the scan response or is just showing that data from the advertisement. A scanning device that is interested in learning the name of our advertising device thus has to send a scan request and obtain the name from the scan response.

We then let our device advertise at the same time as a computer runs LightBlue. When the device name shows up in LightBlue, we conclude that the scan response works according to the specification.

Figure 6.1 shows that the second requirement is met. Here the advertising device receives a scan request and replies to it with a scan response. The time between the request and the response is within reasonable limits around 150 µs.

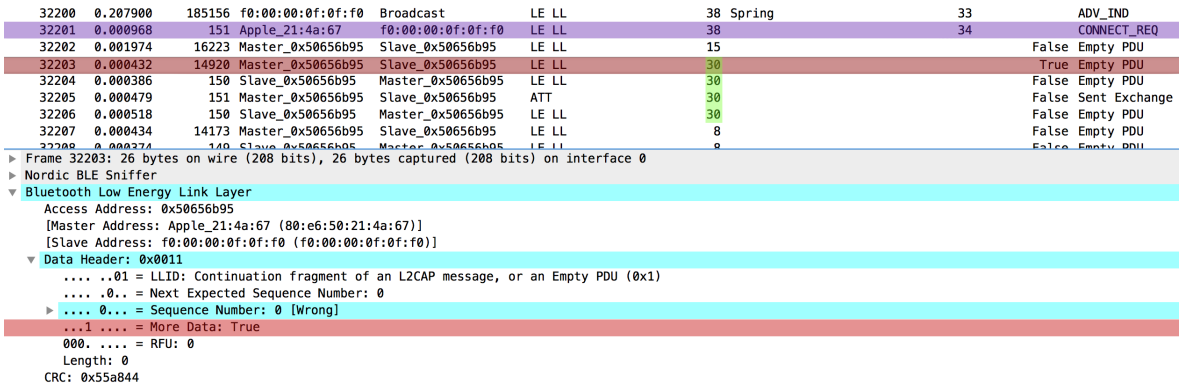

Figure 6.2: *Screenshot from Wireshark showing the start of a connection. The upper red line highlights a packet in which the more data (MD)-bit is set, which is shown as* `True` *in the second to last column. As the vertical green line shows, this packet is received by the slave on channel 30. The slave does as expected and stays on the channel until it has received another packet from the master.*

The device establishes a connection when a connection request is received

As Section 2.4.2 explains, a connection is not established until a device has received a data packet from its peer. Therefore, the test for this requirement includes both that our device receives a connection request, receives a data packet from the master and then replies to the latter.

To verify that our device can establish a connection in the slave role, we let it start as an advertiser and send a connection request to it using LightBlue. We monitor in Wireshark if the two devices start to send packets to each other.

Figure 6.2 shows the start of a connection. "Apple_21:4a:67" is a computer running LightBlue and "f0:00:00:0f:0f:f0" is our nRF52 board. As the violet line highlights, the computer sends a connection request to our device, which means that the computer becomes master and our device becomes a slave. After they both have entered the connection, the master is the first one to send a packet, to which the slave replies. Therefore, we conclude that the third requirement is met.

The device replies to the master on the correct channel

This requirement includes the slave to be on the same channel as the master, as well as replying to the packets. As Section 2.4 describes, a device can continue to send on the same channel if it sets the MD-bit in the packet header. Therefore, the slave must make sure not to switch its channel if this bit is set. The expected behaviour of the slave is thus to stay on the same channel until the MD-bit is not set.

We use Wireshark to monitor if the slave replies to each of the master's packets. Afterwards, we print the channel we just transmitted on during a non-time critical period. This allows us to verify that the transmission was sent on the expected channel. We can also quickly find out if a failure of replying to the master is caused by our device sending on the wrong channel or if it does not receive any packet from

the master.

Using Wireshark, we can see that when the slave receives a packet where the More Data (MD)-bit is set, it stays on the channel and waits for another packet from the master. When this second packet is received, the slave also replies to that packet (see Figure 6.2). This shows that the MD-bit is respected and the connection works as expected.

There are occasions when the slave does not reply to the master at all. Our printing suggests that this is a result of that the slave has a timeout, i.e. has waited without receiving and therefore moves to the next channel. The timeouts could be either due to a packet loss or mistakes in our code. As the devices are relatively close, the latter seems more likely.

Even though what is described above, the slave recovers from the missing packets and continues to reply. Therefore, we still argue that the slave's behaviour meets this requirement.

The device should change channel map if the master updates it

As the slave and the master rely on a combination of a channel selection algorithm and a channel map to switch channel in the same way, both devices, of course, have to use the same algorithm and channel map. As the slave includes in its advertising packets what algorithm it wants, we only have to make sure that the two devices are using the same channel map during the connection.

The channel map is, as Section 2.4.2 describes, included in the connection request sent by the master, but it can also be changed during the connection. To do this, the master sends a new channel map to the slave, as Figure 6.3 shows. The master specifies in the same packet when the new channel map becomes valid.

We test that the above-mentioned channel map is noticed and respected by the slave. This is easily done as we know from reading the packet both when the slave is supposed to start using the new channel map and which channels the master specifies as valid.

Using Wireshark, we can read the packet and use the same algorithm as the slave just before and after the change of channel map to make sure that the transition indeed takes place.

Another way to check would be to simply run the connection and assume that is works if it does not terminate soon after a new channel map starts being used. We find this way of testing to be less reliable as two channel maps can overlap. This means that even after the slave starts using the new channel map, it might continue to jump between channels that are valid according to the old channel map for a while. Therefore the interesting part to test is the part where the old and the new channel map does not overlap.

Our slave device successfully changes channel map as often as the master requires, and therefore we conclude that this requirement is fulfilled.

11957	0.005972	149	Slave_0x50656ad1	Master_0x50656ad1	LE LL	31	5363	False	Empty PDU
11958	0.001833	14690	Master_0x50656ad1	Slave_0x50656ad1	LE LL	0	5364	False	Empty PDU
11959	0.000445	150	Slave_0x50656ad1	Master_0x50656ad1	LE LL	0	5364	False	Empty PDU
11960	0.102424	14690	Master_0x50656ad1	Slave_0x50656ad1	LE LL	6	5365	False	Empty PDU
11961	0.005101	150	Slave_0x50656ad1	Master_0x50656ad1	LE LL	6	5365	False	Empty PDU
11962	0.001888	14690	Master_0x50656ad1	Slave_0x50656ad1	LE LL	12	5366	False	Control Opcode: LL_CHANNEL_MAP_REQ
11963	0.001182	150	Slave_0x50656ad1	Master_0x50656ad1	LE LL	12	5366	False	Empty PDU
11964	0.000640	14626	Master_0x50656ad1	Slave_0x50656ad1	LE LL	18	5367	False	Empty PDU
11965	0.001759	150	Slave_0x50656ad1	Master_0x50656ad1	LE LL	18	5367	False	Empty PDU
11966	0.001924	14690	Master_0x50656ad1	Slave_0x50656ad1	LE LL	24	5368	False	Empty PDU
11967	0.001044	150	Slave_0x50656ad1	Master_0x50656ad1	LE LL	24	5368	False	Empty PDU
11968	0.000480	14690	Master_0x50656ad1	Slave_0x50656ad1	LE LL	30	5369	False	Empty PDU
11969	0.000331	150	Slave_0x50656ad1	Master_0x50656ad1	LE LL	30	5369	False	Empty PDU
11970	0.000302	14689	Master_0x50656ad1	Slave_0x50656ad1	LE LL	36	5370	False	Empty PDU
11971	0.000427	151	Slave_0x50656ad1	Master_0x50656ad1	LE LL	36	5370	False	Empty PDU
11972	0.000428	14689	Master_0x50656ad1	Slave_0x50656ad1	LE LL	5	5371	False	Empty PDU
11973	0.000462	149	Slave_0x50656ad1	Master_0x50656ad1	LE LL	5	5371	False	Empty PDU

Figure 6.3: *This figure shows how the master informs the slave that they are about to change channel map. The red line shows the packet that contains the new channel map*

The device listens for packets on a channel for no longer than the timeout

A timeout means that our device has waited for a packet long enough and now moves to the next channel. The reason for this behaviour is that if this is not done, a device might miss the timing constraints specified in the Bluetooth specification. This requirement has to be tested for the device both when it advertises and when it is in a connection.

When the device is advertising, it should spend a maximum of 10 ms on each channel (see Section 2.4). For a connection, this timing varies depending on parameters set by the master.

To test this, we run our device as an advertiser without starting receive mode, while still starting the timeout. If we do not start receive mode, the radio will timeout every time since no packets can be received. The idea is to check that the radio still changes its channel while being unable to receive packets. Using Wireshark, we can verify this behaviour.

After this test, we conclude that the timeout function indeed works for a device in the advertising state.

Discussion

From our tests, it seems like the six requirements stated at the start of this subsection are indeed met. We see that an advertiser can both transmit packets in the expected way and that it responds to scan requests. These two behaviours are fundamental for a scannable advertiser. Also, even if it was not explicitly stated as a requirement, the device continues to advertise after replying to a scan request. This behaviour is vital for the functionality of the advertiser, as it would stop functioning otherwise. Moreover, the ability for the device to advertise its existence is essential for other devices to send connection requests to it, as a device is not allowed to send a connection request to another device from which it has not received an advertisement.

During our test runs the advertiser has been stable, meaning we have not seen any unexpected behaviour. We can, though, in some executions see that other devices send scan requests to our advertising device during several advertising intervals

before they get any response. This is most likely due to the amount of traffic on the 2.4 GHz radio band. A greater amount of traffic means that the advertiser receives more packets that it has no interest in, which could, for example, be advertising packets from other advertisers. During the time our advertiser handles one such packet, it will not receive any other packets and thus miss any scan request. Also, if the received packet is not of interest, the device will move on to the next advertising channel, meaning that any other device which wishes to send a scan request to our device also has to change to the next channel, and wait for our device to send the next advertisement.

As the only way a connection can be created is for the advertiser to receive a connection request, the third requirement has to be fulfilled. Similar to the case of scan requests, the slave tend to miss a couple of connection requests before it finally receives one. As also this has to do with the reception mechanism, we assume that the reason is the same as for scan requests.

Both the fourth and the fifth requirements state the needed behaviour for keeping a connection alive, and as we have not spotted any cases in which the slave shows a strange behaviour, we assume these to work without any difficulties.

The sixth and last requirement is critical for a connection to be possible. As long as the master sends a data packet on each channel, and the slave receives this packet, the timeout is not necessary. In that case, it is only used for lowering the power consumption by allowing the slave to get idle if no more packets are to be sent during the current connection event. On the other hand, if the slave fails to receive the packet from the master, or if the master fails to send it, the timeout is a crucial tool to force the slave to the next channel. This way, the slave can recover from packet loss.

Even if our BLE implementation passes all the requirements we set up to validate it, the slave device behaves oddly during the connection. At some point, it loses the master device. Continuously, it tries to receive a packet from the master, but every time it gets a timeout. We are not sure what is the underlying cause to this bug, but the result seems to be that the slave has an offset to the master, which causes it to start listening either too early or too late. This might suggest a bug in the timeout mechanism, but we have not been able to prove this, nor have we found any valid reason to discard the idea. We have not been able to find a pattern on when the slave gives up on the master; often it happens after a couple of thousand connection events, but sometimes the connection lives long enough for the event counter to flip over and start counting again from zero, i.e., the connection lives for more than 65 536 connection events.

Even in the light of the issue mentioned above, we claim that our design of the minimal stack is reasonable. The design and implementation are related, but they are different parts of the creation of software, and different implementations can be created from the same design description. Therefore we argue that the fault lies in our implementation rather than in the design itself. As the design is a construction of our definition of a minimal stack, we argue that as the design fulfils our goals, so does also the definition.

```
1     adv_params.itvl_max = 480;
2     adv_params.itvl_min = 300;
```

Listing 6: *The parameters in Apache Mynewt changed by us to better match the configuration used by Tock.*

6.3 Performance Testing

In this section, we describe the part of our evaluation that includes measurable values. The tests are designed to check the performance of our implementation. This is of interest even if the goal of this master's thesis does not state anything regarding the effectiveness of our implementation, as embedded devices have to be both reliable and concerned of how they use their limited energy.

We have chosen to analyse three attributes:

1. Reliability - how large percentage of the packets sent by the master is received
2. Power consumption - the energy consumed by the nRF52 board
3. Timing - closely related to the reliability of the system. The timing is measured as how soon after reception of a packet from a master the slave replies.

Measurements of one implementation have to be compared with others, as they are not very meaningful on their own. Therefore, we use the same tests to evaluate the performance of Apache Mynewt BLE implementation as we do for our implementation. This allows us to compare the two, and to see how well our implementation stands against that of another open source operating systems.

We are not discussing how differences in the results might be related to the programming language used by the two operating systems. Even though this is an interesting aspect to investigate, the BLE implementations are quite different and are also running on different operating systems, making a direct comparison possibly unfair and difficult. The results are presented and discussed in Section 6.3.5.

6.3.1 Setup

For Apache Mynewt's NimBLE, we use one of their provided examples, namely 'bleprph'. By observing the nRF52 board's behaviour when it runs this code, we can see that the example's advertising intervals are much shorter than those of our implementation. As this can affect the overall power consumption, giving an unfair comparison, we change these parameters to match those used by us, as is seen in Listing 6. After this preparation, we perform the measurements of performance, power consumption, and timing.

6.3.2 Reliability

We measure the reliability of our implementation as the percentage of the packets sent that a slave device receives. This measurement is only conducted after the

device has created a connection, i.e. from the first data channel packet sent by the master device.

In a connection, both the master and the slave counts how many connection events have passed. As the master should send at least one packet in each connection event, we know that the number of packets sent will not be less than the number of connection events. More packets might have been sent, as several packets can be sent within the same connection event. These extra packets are included in the calculation by counting the number of times two packets have the same event counter, as this means that the packets have been sent during the same connection interval. From this, we get the following expression:

$$\frac{P_{rec}}{E_{count} + P_{doublet}}$$

, where P_{rec} is the number of packets received by the slave, E_{count} the connection event counter, and $P_{doublet}$ the number of packets that shares the same event counter.

We run the connection for an arbitrary number of connection events, as the percentage is not directly related to the number of counted packets and that we assume that the failure of receiving packets are evenly spread out over the connection event.

If one had any reason to believe that packet loss was, for example, more likely to happen later during a connection, it would have been better to only count the number of packets during a predefined number of connection events. As we have not noticed any trend in packet loss, we do not have such a reason.

Results The results of our reliability are measured using Wireshark. The reason for this is to measure both the master and the slave from an equal point of view and to minimise the adverse effect measurements on the board could have.

The results are presented as the percentage of packets missed by the slave, as we assume that the master succeeds in its task to send a packet in each connection event, and also due to it is the slave that runs our code.

We have to account for that we cannot know why a packet is not showing up in Wireshark. Either the sniffer could have just missed it, or the master never sent it. We counter this by counting the number of packets that sniffer claims not to have received from the master, and subtract this number from the number of packets the sniffer has not received from the slave. By doing this, we only include the number of packets to which the slave actually could reply.

Our results for packet loss for Tock: 1.56 %, and Mynewt: 1.37 %.

6.3.3 Power Consumption

The power consumption is not directly measured, rather we measure the current. This should not be done when that nRF52 board is powered using USB, and therefore we connect the board to an external power source instead. This allows for a voltage level between 1.7 V and 3.6 V, compared to 3.3 V when USB is used. We choose a voltage of 3.0 V.

Before the current can be measured, the board has to go through some preparation steps. The nRF52 board has a connector with two pins that are used for current measurements. A solder bridge has to be cut to enable these pins, as the current normally flows over this bridge. After this is done, another solder bridge has to be shorted to allow it to be connected to an external power supply.

As we are using an oscilloscope, we also have to add a 10 Ω resistor between the two pins of the connector mentioned above, as this enables to measure the voltage drop over the resistor.

After the setup, we calculate the current as

$$I = \frac{U_{drop}}{R}$$

where U_{drop} is the voltage drop over a resistor with resistance R.

From this, we can derive the power usages in watt as

$$P = I \cdot U_{supplied}$$

where $U_{supplied}$ is the voltage level from the external power supply.

The measurements are done separately for advertising and connection. Using the oscilloscope, we collect data points representing the voltage drop during a specific period. For advertising, we choose to measure over a whole advertising event, during the part of the interval when the radio is active (TX/RX) and during a period when the radio is idle.

As a connection event can vary between runs, we choose to collect data over a predefined number of connection events. We then calculate the average power consumption based on this data.

Another factor that could affect the power consumption is the packet length. A larger packet will contain more data, causing the radio to consume more power while transmitting this packet. Due to this, we check that the packet length is the same for both Tock and Mynewt.

Results The result from the measurements sometimes gave us negative values. As it does not make sense to include negative values when calculating the average and also to get results that are comparable with those Nilsson and Adolfsson got last year, we choose to do the same as they did and set all negative values to zero.

As can be seen in Figure 6.4, TX is more power consuming than RX. The measurements also reveal that when the radio switches from TX to RX mode, the power consumption drops to around 10 mW, while going back from RX to RX only yields a drop to roughly 30 mW. This is due to our design, in RX an `END` event marks the end of the reception, while in TX the module waits until a `DISABLE` event occurs. Since a `DISABLE` event always comes later than an `END` event (see Figure 5.1 this could account for some of the timing difference. Also, due to hardware reasons, it takes longer for the hardware radio to transition from TX to `DISABLE` than from RX to `DISABLE` [2]. These two reasons together could provide an explanation why the hardware module is ready to handle the event faster after an RX than after

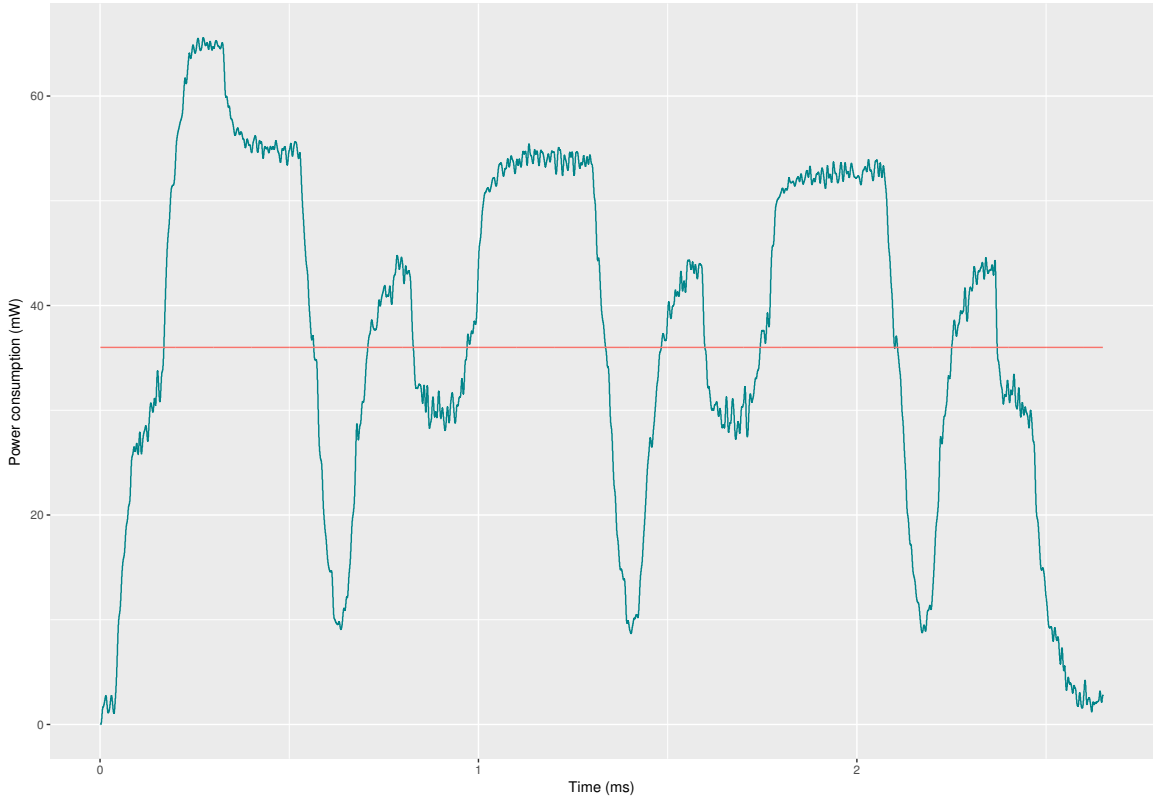

Figure 6.4: *Graph showing Tock's power consumption during its active time period of an advertising event. The higher peaks (the first, third and fifth) represents when the radio is in TX mode, while the lower peaks (the second, fourth and sixth) is times when the radio is in RX mode. The average value is highlighted in orange.*

TX and might explain why the power consumption does not reach the bottom in between these events.

We can also see that compared to the BLE implementation from last year, our implementation has increased the power consumption during advertising in Tock. Nilsson and Adolfsson present that their implementation measure 1.39 mW during idle time, 27.67 mW for the transmission and 1.72 mW in total. This is compared to our 1.73 mW, 36.35 mW and 1.83 mW. One probable reason is that as we have added listening to the advertising event the time the radio is doing useful work has been extended. Also, Nilsson and Adolfsson turned off the radio when it was not used, which further lowers the power consumption.

Our measurements of the advertiser show that the BLE implementation in Tock beats that of Mynewt both in the test for idle power consumption and power consumption during an entire advertising interval, as shown in Figure 6.5. On the other hand, Mynewt's power consumption during a period of transceiving (TX and RX) is about 36% of that of Tock's.

As an advertising interval is a combination of the transceiving period and a period when the radio is idle, the power consumption during these two periods directly affects the power consumption during the advertising interval. The idle time is much larger than the period of transceiving, and therefore it is not surprising to

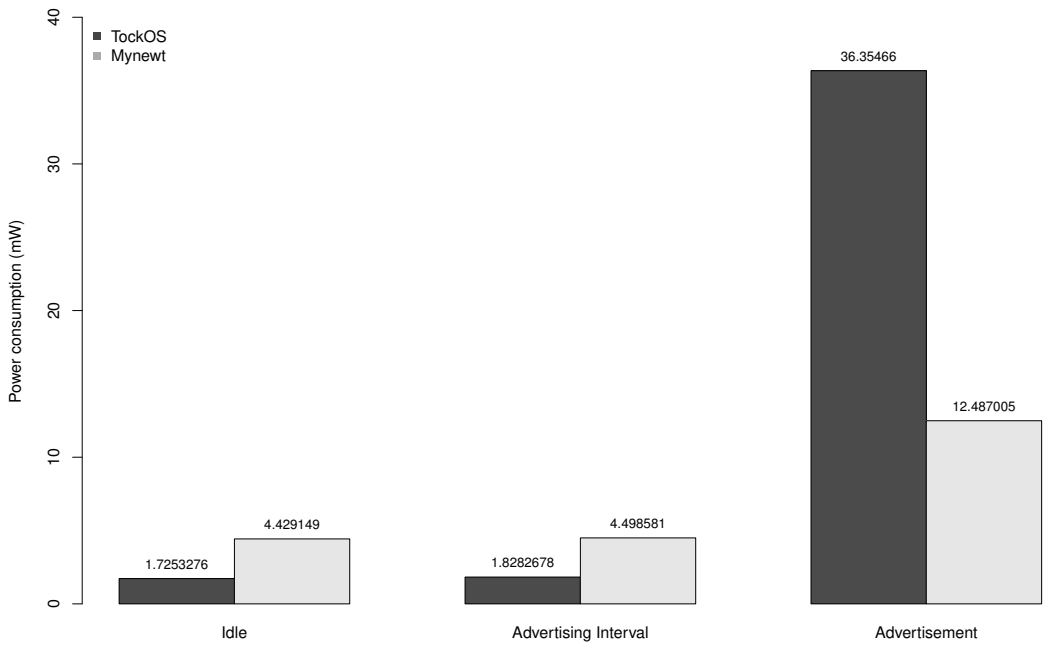

Figure 6.5: *Bar chart comparing power consumption of Apache Mynewt and TockOS during different events in advertising.*

see that Tock has a lower power consumption during an advertising interval.

If the idle time was to be shortened, the transceiving period has a more significant impact on the result, which in our case would increase Tock's average power consumption during an advertising interval.

Comparing the power consumption during idle time and a transceiving period indicates the importance of the idle time periods. As embedded devices want to minimise their power consumption, it is of utmost importance for them to strive for maximising their idle time. Of course, if a device has a valid reason to communicate more often, it also has to keep the advertising intervals shorter, resulting in less idle time and higher power consumption.

Except for the periods spent in TX, RX and idle, other things can affect the power consumption. One such thing is the number of scan requests the device receives, as these require a scan response to be sent. Due to this, in our measurements, we choose only to include advertising intervals in which the advertiser received no scan requests.

Also, the test of a connection shows that our implementation has higher power consumption during the period of transceiving compared to Mynewt, while measuring during a whole connection interval shows a similar power consumption between the two operating systems.

The results show that our implementation in Tock has a slightly lower power consumption during the transceiving period in a connection than when it advertises. Most likely due to during an advertising interval the device transmits three times

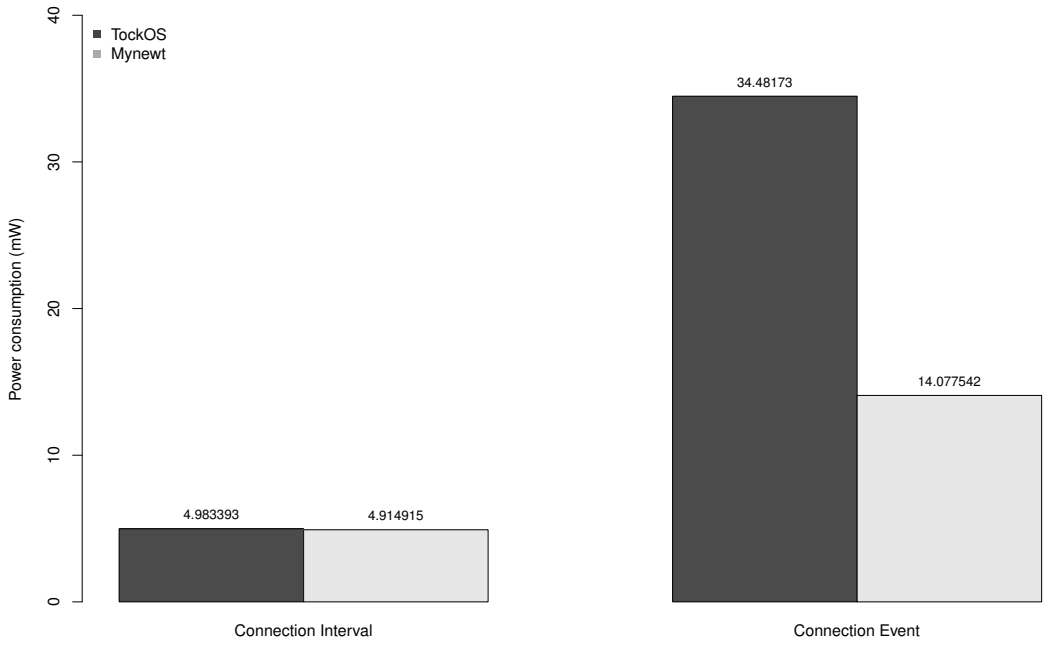

Figure 6.6: *Bar chart comparing power consumption of Apache Mynewt and TockOS during different events in connection.*

more packets than during a connection interval, assuming that the MD-bit is never set.

Comparing the results over an advertising interval and a connection interval, we can see that the power consumption increases for both the operating systems when the device enters a connection. For Tock, this increase is quite drastic, as it is more than doubled (roughly 273% of that during advertising). Even if an advertising interval differs significantly from a connection interval, comparing the measurements from the two could provide a better understanding of the importance of idle periods. This is clear from the results for Tock: during the advertising, the idle time is around 280 ms, while for the connection it is around 15 ms. This means that even if the power consumption during transceiving in a connection event is lower than that for the advertising, a connection has higher overall power consumption.

One needs to keep in mind that both the connection interval and thereby the number of sent packets might vary between runs. Therefore, we only include connection intervals where precisely one packet was sent from each peer. Like for the idle time during advertising, a longer connection interval will mean a long idle time, which is beneficial for the device's power consumption.

6.3.4 Timing

If a slave is too quick or too slow at responding to a packet from the master device, the communication might suffer. If the packet is sent too soon, the master device

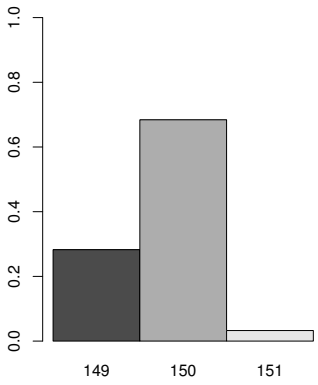

Figure 6.7: *Bar chart comparing the ratio of packets received across the response time for the packet. Extreme values are not included in the figure.*

might not have prepared the radio for receiving, and thus the packet might be missed. On the other hand, if the slave is too slow at responding, the master might have given up on listening and moved on to the next channel.

From this, it is clear that the timing is of importance, and is a parameter that we should evaluate. We choose to measure during connections only, since the way the scheduling of response packets are done similarly in advertising and connection. Also, if an advertiser on a few occasions fails to respond to a scan request at the right time the result is not catastrophic. A connection can, on the other hand, be terminated as a consequence of too many missed timings.

The timing is measured using Wireshark and is the time between the end of a packet sent by the master and the start of the response from the slave.

Results Our results for timing measurements are presented in Figure 6.7. The majority of the packets arrive at 150 µs, ± 1 µs. It is only a matter of ± 1 µs, but as several parts of BLE are sensitive to small changes in timing, this could have a large impact on the functionality of the implementation. One might be tricked to believe that replying faster than the 150 µs would be a positive thing, but as explained in Section 6.3.4, this is not the case.

There are also a couple of values in most of the runs that are much higher or lower, for example at 4 294 966 717 µs (or roughly 4295 s). We are not sure why these deviate so much from the expected value of 150 µs. One suggestion is that this might be a misinterpretation by Wireshark or the sniffer. For the packets we have investigated, the higher values have their CRC marked as invalid by Wireshark. Also, the slave continues the communication with master after this packet, indicating that the slave cannot have transmitted the packet thousands of seconds later than it was supposed to transmit. For the lower values, we can see a similar trend, with the difference that it varies if it is the slave or the master that sends a packet with invalid CRC. Therefore, we do not fully trust the results that show a value that deviates too much from the expected 150 µs and disregard them as errors in measurement.

6.3.5 Discussion

During the performance testing, we have observed both strengths and weaknesses in our implementation.

The measuring of packet loss shows that Tock has a slightly higher packet loss than Mynewt, but still, it is not unreasonably high. It is difficult to know the reason behind this, but it might be due to timing issues or problem due to interference. During the test we assume that the traffic on the 2.4 GHz band does not vary so much that it will give an unfair testing environment, i.e., we assume that both the operating systems suffered equally much from other traffic. Even so, a question arises regarding if the test actually measures the reliability of the BLE implementation or is a measure of how crowded the radio band is. Performing the measurements in a calmer environment might have yielded a different result. Even so, the results still show how well a device running the implementation can handle its current environment, and therefore we still find the test to be reasonable. In either case, a device which manages to communicate using wireless protocols even in the presence of disturbance is highly desirable. This is both due to the cost of resending as well as the impracticality with a too large performance drop as a result of interference on the communication medium.

From our results of the power consumption, it is clear that our BLE implementation in Tock if far from optimal. If the power consumption is decreased during periods of transceiving, the implementation will stand better against that of other operating systems.

One possible explanation for Mynewt's lower power consumption is that they are using dynamic transmission power. We have not considered power consumption during the implementation in Tock, and therefore the radio uses a default transmission power of 0 dBm. Adapting this value depending on signal strength, as done in Mynewt, would result in improved energy efficiency.

In Figure 6.7 we see that almost 70% of the response packets manage to be sent after exactly 150 µs, even if it the majority of the packets does not vary more than ± 1 from that value. Our method of measuring cannot be guaranteed to provide an exact value, neither does it give us an understanding of whether the value shown in Wireshark is accurate or if it sometimes might be a bit off. Either way, one would like to improve the percentage of times that the timing value is exactly 150 µs.

This result is a bit surprising. As we schedule a response with the same delay each time, the difference shown in Wireshark could be due to a non-deterministic behaviour in the program itself. During a connection, the slave should respond in the same way each time, performing the same operations. Therefore, we found it likely that the ± 1 could be caused by rounding errors in Wireshark, even though we cannot discard the fact that the values actually could be correct.

7
Conclusion

7.1 Conclusion

The Bluetooth specification describes a whole variety of functionality, some mandatory and other voluntary. This master's thesis presents what we argue is the simplest connection that can be created and maintained between two IoT devices. From this, we derive a definition of the minimal Bluetooth Low Energy stack needed to enable this simple connection.

To try out this definition, we realise it by implementing it in an operating system for embedded devices. Out of several such operating systems we choose to use Tock as our platform. This is a relatively new operating system, which is implemented in the programming language Rust. Rust promises near C performance, and it is therefore fascinating to see how well-suited it is for implementing a BLE stack.

Our design of the BLE stack is formed from both what is required to enable the communication with another device, as well as what is suitable for Tock. Also, as a small BLE stack, with support for only advertising, once existence or for finding other devices, already existed, we aimed at keeping the changes to is at a minimum.

Our implementation introduces new functionality, but also requires some restructuring and optimisations of the existing code. The optimisations focus mostly on how we can utilise the hardware to improve timings. We enable this by allowing the control flow to be driven by interrupts from the hardware.

The first part of the evaluation tests that our implementation meets six different requirements. Some of the requirements test that the device knows how to advertise, as this is fundamental for a connection to be created. Other tests of the requirements check so that the connection between our device and a master device works as defined in the Bluetooth specification.

Even though the performance of the implementation is not directly related to the goal of this master's thesis the second part of the evaluation tests how well our device performs compared to Mynewt, an operating system with a full BLE stack implementation. The tests highlight that there is room for optimisation of the power consumption.

As of today, our implementation of BLE has way too high power consumption to run for extended periods of time with a battery as its only power source. The transceiving period has to be improved upon for both during advertising and during a connection for the implementation to be useful. Another way of improving the overall power consumption is to turn off the radio for the device if it is not used during long periods.

Even if there are things to improve upon, the implementation is shown to be relatively reliable and hold the timings within the BLE specification, which indicates that it has its advantages and might make a solid foundation from which to continue the implementation.

7.2 Future Work

If one aims at building a full BLE stack for Tock, there is quite some work to be done. Many features can be implemented without changing the design that Chapter 4 presents, while others require additional layers to be added. A natural first step would be to improve upon the existing code for the advertising state and the slave, and later include support for a scanner, an initiator and master of a connection.

Improvements would include both increasing support for handling a connection after it has been established, as well as optimising the code to lower the power consumption. For example, in the current implementation, the radio is always on. As it is one of the most power consuming peripherals, devices that have much idle time would benefit from turning it off when possible.

Adding support for different types of advertisements could also be a part of future work, as it would allow for different types of advertisers, not only the scannable and connectable as we have implemented.

Further, a logical next step would be to have the slave returning to advertising after a connection is terminated. The termination could be triggered either voluntary by any of the peers by sending a disconnect command, or after a device has waited for an extended period without receiving any packets. It would be preferable if the slave could handle both of these cases. As it is today, the slave device continuously tries to receive packets from the master, even after the master has ended the connection.

Bibliography

[1] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kb computer safely and efficiently," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (Shanghai, China), pp. 234–251, ACM, 2017. [Online]. Available: `https://dl.acm.org/citation.cfm?doid=3132747.3132786`, Accessed: 2018-04-05.

[2] Nordic Semiconductor, *nRF52832 Objective Product Specification*, November v0.6.3, Nordic Semiconductor, Nov. 2015. [Online]. Available: `http://infocenter.nordicsemi.com/pdf/nRF52832_OPS_v0.6.3.pdf`, Accessed: 2018-03-02.

[3] M. Honkanen, A. Lappetelainen, and K. Kivekas, "Low End Extension for Bluetooth," in *Proceedings. 2004 IEEE Radio and Wireless Conference (IEEE Cat. No.04TH8746)*, (Atlanta, GA, USA), pp. 199–202, IEEE, Sept 2004. [Online]. Available: `http://ieeexplore.ieee.org/document/1389107/?arnumber=1389107`, Accessed: 2018-02-20.

[4] L. R. Wilhelmsson, M. M. Lopez, and D. Sundman, "NB-WiFi: IEEE 802.11 and Bluetooth Low Energy Combined for Efficient Support of IoT," in *2017 IEEE Wireless Communications and Networking Conference (WCNC)*, (San Francisco, CA, USA), pp. 1–6, IEEE, March 2017. [Online]. Available: `http://ieeexplore.ieee.org/document/7925808/`, Accessed: 2018-04-09.

[5] Bluetooth Special Interest Group, "SIG introduces Bluetooth Low Energy wireless technology, the next generation of Bluetooth wireless technology," 2009. [Online]. Available: `https://www.bluetooth.com/news/pressreleases/2009/12/17/sig-introduces-bluetooth-low-energy-wireless-technologythe-next-generation-of-bluetooth-wireless-technology`, Accessed: 2018-02-20.

[6] N. Diakopoulos and S. Cass, "The Top Programming Languages 2017 - IEEE Spectrum," 2017. [Online]. Available: `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017`, Accessed: 2018-02-21.

[7] F. Nilsson and N. Adolfsson, *A Rust-based Runtime for the Internet of Things*. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden, 2017. [Online]. Avail-

able: http://studentarbeten.chalmers.se/publication/250074-a-rust-based-runtime-for-the-internet-of-things, Accessed: 2018-03-28.

[8] Nordic Semiconductor, "nRF52832." [Online]. Available: https://www.nordicsemi.com/eng/Products/Bluetooth-low-energy/nRF52832, Accessed: 2018-05-01.

[9] Nordic Semiconductor, "nRF52 Series SoC." [Online]. Available: https://www.nordicsemi.com/Products/nRF52-Series-SoC, Accessed: 2018-05-01.

[10] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis, "Integrating concurrency control and energy management in device drivers," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (Stevenson, Washington, USA), pp. 251–264, ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294286, Accessed: 2018-03-23.

[11] K. Townsend, C. Cufí, Akiba, and R. Davidson, *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking*. Sebastopol, CA, USA: O'Reilly Media, 2014. [Online]. Available: https://www.safaribooksonline.com/library/view/getting-started-with/9781491900550/, Accessed: 2018-02-28.

[12] Bluetooth Special Interest Group, *Bluetooth Core Specification*, vol. 6. 5 ed., Dec 2016. [Online]. Available: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043&_ga=2.173440995.1790814702.1512399515-1514272347.1510473596,https://www.bluetooth.com/specifications/bluetooth-core-specification, Accessed: 2017-12-04.

[13] Contiki, "Contiki: The Open Source OS for the Internet of Things," 2012. [Online]. Available: http://contiki-os.org, Accessed: 2018-04-05.

[14] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "Tinyos: An operating system for sensor networks," in *Ambient Intelligence* (W. Weber, J. M. Rabaey, and E. Aarts, eds.), pp. 115–148, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-27139-2_7, Accessed: 2018-04-05.

[15] Apach Mynewt, "Apache Mynewt," 2017. [Online]. Available: https://mynewt.apache.org, Accessed: 2018-04-05.

[16] Gavin Jefferies, Justin Mclean, David G. Simmons et al., "apache/mynewt-core: An OS to build, deploy and securely manage billions of devices." [Online]. Available: https://github.com/apache/mynewt-core, Accessed: 2018-05-23.

[17] Apache Mynewt, "BLE Introduction." [Online]. Available: https://mynewt.apache.org/network/ble/ble_intro/, Accessed: 2018-05-23.

[18] Apache Mynewt, "Bluetooth Low Energy 4.2 - Apache Mynewt." [Online]. Available: https://mynewt.apache.org/pages/ble/, Accessed: 2018-05-23.

[19] Apache Mynewt, "Apache Mynewt." [Online]. Available: https://mynewt.apache.org, Accessed: 2018-05-23.

[20] Apache Mynewt, "apache/mynewt-nimble." [Online]. Available: https://github.com/apache/mynewt-nimble, Accessed: 2018-05-23.

[21] Apache Mynewt, "apache/mynewt-nimble." [Online]. Available: https://github.com/apache/mynewt-nimble/tree/f6bd8dc1f496557523697825f806d22bf73acb21/nimble/controller/src, Accessed: 2018-05-23.

[22] Apache Mynewt, "apache/mynewt-nimble." [Online]. Available: https://github.com/apache/mynewt-nimble/blob/f6bd8dc1f496557523697825f806d22bf73acb21/nimble/controller/src/ble_ll.c, Accessed: 2018-05-23.

[23] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*, (Tampa, FL, USA, USA), pp. 455–462, IEEE, Nov 2004. [Online]. Available: http://ieeexplore.ieee.org/document/1367266/, Accessed: 2018-04-03.

[24] Contiki, "Contiki Hardware." [Online]. Available: http://contiki-os.org/hardware.html, Accessed: 2018-04-03.

[25] M. Spörk, *IPv6 over Bluetooth Low Energy using Contiki*. Master's thesis, Institute for Technical Informatics, Graz University of Technology, Graz, Austria, 2016. [Online]. Available: https://michaelspoerk.com/wp-content/uploads/2017/11/IPv6-over-Bluetooth-Low-Energy-using-Contiki.pdf, Accessed: 2018-06-03.

[26] S. Akhshabi and C. Dovrolis, "The evolution of layered protocol stacks leads to an hourglass-shaped architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. 206–217, Aug. 2011.

[27] Nieminen et al., "IPv6 over BLUETOOTH(R) Low Energy," RFC 7668, IETF, July 1995.

[28] Bluetooth Special Interest Group, *Bluetooth Core Specification*, vol. 1. 5 ed., Dec 2016. [Online]. Available: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043&_ga=2.173440995.1790814702.1512399515-1514272347.1510473596,https://www.bluetooth.com/specifications/bluetooth-core-specification, Accessed: 2017-12-04.

[29] Bluetooth Special Interest Group, *Bluetooth Core Specification*, vol. 3. 5 ed., Dec 2016. [Online]. Available: https://www.bluetooth.

org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043&_ga=2.173440995.1790814702.1512399515-1514272347.1510473596,https://www.bluetooth.com/specifications/bluetooth-core-specification, Accessed: 2017-12-04.

[30] Nordic Semiconductor, "nRF Sniffer/Bluetooth Low Energy/Products/Home - Ultra Low Power Wireless Solutions from NORDIC SEMICONDUCTOR." [Online]. Available: https://www.nordicsemi.com/eng/Products/Bluetooth-low-energy/nRF-Sniffer, Accessed: 2018-05-01.

[31] Wireshark, "Wireshark." [Online]. Available: https://www.wireshark.org, Accessed: 2018-05-01.

[32] Apple Inc., "LightBlue on the Mac App Store." [Online]. Available: https://itunes.apple.com/se/app/lightblue/id639944780?mt=12, Accessed: 2018-05-01.