Delft University of Technology
Master of Science Thesis in Embedded Systems

Optimizing Connection Establishment and Parameter Adaptation in Bluetooth Low-Energy for Intermittently-powered Devices

Nathan Prins

TU Delft Delft University of Technology

Optimizing Connection Establishment and Parameter Adaptation in Bluetooth Low-Energy for Intermittently-powered Devices

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Nathan Prins

11-10-2022

Author
Nathan Prins (n.prins-2@student.tudelft.nl)
(nathanprins@hotmail.com)
Title
Optimizing Connection Establishment and Parameter Adaptation in Bluetooth Low-Energy for Intermittent
MSc Presentation Date
13-10-2022

Graduation Committee

Dr. Przemysław Pawełczak	Delft University of Technology
Dr. Asterios Katsifodimos	Delft University of Technology
Jasper de Winkel	Delft University of Technology

Abstract

The growing field of research into batteryless or intermittent systems has enabled Internet of Things applications that were previously impossible. For example, the FreeBie system recently introduced Bluetooth Low-Energy (BLE) to intermittent devices, making medium to long range bi-directional communication a reality for the first time. However, this achievement also highlighted that some inefficiencies considered acceptable for conventional systems are unacceptable when working in the intermittent domain. Our key insights are that intermittent peripherals should dictate the connection parameters and not the central, that connection setup overhead should be reduced as much as possible, and that connection parameter updates should be applied faster. To achieve these goals, we 1) introduce a method of sharing connection parameters before a connection is made, 2) introduce methods of caching connection setup packets together with a reconnect procedure called Fast Reconnect that reduces connection setup to a single packet, and 3) apply 1 and 2 in a dynamic algorithm called FRAPPUCcInO that controls connection rate based on energy harvesting capabilities. These three solutions allow intermittent BLE to be used in environments with less ambiently available energy than before by improving efficiency and responsiveness.

"Never gonna give you up" – Richard P. Astley

Preface

This MSc thesis has been carried out within the Embedded and Networked Systems group at the Delft University of Technology under the supervision of Dr. Przemysław Pawełczak and Ph.D. Candidate Jasper de Winkel. The goal from the beginning was always to improve upon the state-of-the-art battery-less system called FreeBie, developed by De Winkel and Dr. Pawełczak. While familiarizing myself with FreeBie and Bluetooth Low-Energy (BLE), it slowly became evident why the current revision of BLE was not perfectly suited for intermittent devices. This gave me motivation and a clear vision for improving BLE for use within the intermittent domain.

In the first year of my MSc, I followed a course called "Wireless IoT and Local Area Networks" by Dr. Pawełczak, which quickly became one of my favorite courses I got to experience at TU Delft. During this course, Dr. Pawełczak presented his research on intermittent devices, which led me to do my thesis with him, with Jasper de Winkel becoming my daily supervisor. Firstly, I would like to thank them both for this opportunity. I would also like to thank them for their continued effort, meticulous eye for detail, and expertise. Finally, and perhaps more importantly, I would like to thank them for what has been a very enjoyable and personal experience during the final year of my MSc journey. To Dr. Asterios Katsifodimos, thank you for accepting the role of committee member and reviewing my work as part of it; I sincerely hope you enjoy it.

Looking back at my time at TU Delft and my BSc before, I am filled with pride and the simultaneous feeling that it has been so long, and it is already coming to an end.

Nathan Prins

Delft, The Netherlands
11th October 2022

Contents

Chapter 1

Introduction

Powered by the decreasing process-node sizes, more efficient battery chemistry, and expanding wireless communication infrastructure, billions of Internet of Things (IoT) devices have entered our lives [1, 10]. However, the promise of "smart dust" seems to elude us still [8]. In this vision, tiny internet-connected devices permeate our clothing, infrastructure, and more, enabling us to monitor everything [9].

One of the key aspects of wireless devices holding us back from this idea is the reliance on rechargeable, chemical batteries like those based on lithium. The lifespan of these batteries is severely limited by the number of charge cycles they can withstand before slowly reducing their total capacity. Moreover, even if they could maintain the same capacity throughout their entire lifespan, charging billions of sensor nodes is inconvenient, if not impossible, especially if they are embedded in concrete structures for maintenance monitoring purposes, for example.

The field of battery-free or batteryless computing aims to solve this by entirely ridding these devices of conventional batteries, opting to use capacitors as energy reserves instead [6, 8, 20]. Capacitors provide numerous advantages compared to pouch or cylinder style lithium batteries, like tiny form factor, no leakage current, solid state architecture, and practically infinite lifetime, at least compared to other components within the device.

Battery-free devices usually get their energy from harvesting. These sources could include solar (through photovoltaic cells) [5, 4], radio-frequency (through induction within the antennae) [7, 18], kinetic (e.g. by pressing buttons that capture kinetic energy) [5] and sometimes a combination of two or more methods. As one might imagine, these sources are sparse and can be unpredictable.

To conserve energy as much as possible, these devices often only power up parts of the system that are strictly necessary. This can take the form of the system only receiving power from its harvesting when being used (e.g., energy-harvesting buttons in smart light switches) or the system scheduling its own power-on events using low-power circuitry. This mode of operation is often called *intermittent computing* and these devices *intermittent* or *intermittently-powered devices*. Figure 1.1 shows how these devices operate versus conventionally-powered devices.

Intermittently-powered operation makes wireless communication difficult. In the state-of-the-art solutions for intermittently-powered devices, Uni-directional

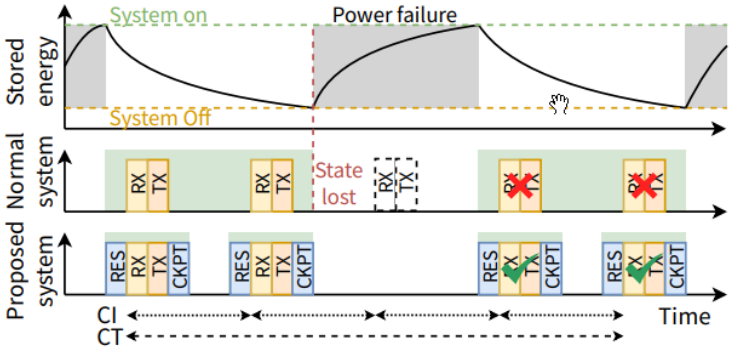

Figure 1.1: **Intermittently-powered device operation versus a conventionally-powered device operation (Figure taken from [6]). When the conventional, non-protected system exhibits a power failure, all network state is lost, and the handshake procedure has to be fully restarted. An intermittently-powered device powers itself down in between network events in order to conserve energy as well as being able to fully recover from a power failure as if it was operating normally.**

communication has been achieved by advertising data when enough power is available. For example, bi-directional communication has been realized at a short range using RFID. With RFID technology, a reader "interrogates" a device that communicates back utilizing the principle of backscatter [18]. Although RFID is bi-directional, it requires a reader to energize the sensor devices at a short distance. As a result, medium-range, high throughput, intermittently-powered communication with active radios has not been possible until very recently. The system proposed by [6], called FreeBie, aims to solve this. It does so by using Bluetooth Low-energy and noting that, between transmission events, the System on Chip (SoC) could be powered down fully, only leaving on an ultra-low-power real-time clock (RTC) tasked with turning the SoC back on just in time for the next transmission. By doing so [6] achieved the first ever battery-free devices capable of maintaining a wireless connection of a widely accepted standard. Even though BLE, as the name would suggest, has been developed for low-energy peripherals, its use within an intermittently-powered device has exposed some inefficiencies previously considered acceptable.

The work within this thesis will be based on the FreeBie system. By modifying the BLE stack source code, we aim to mitigate the inefficiencies of FreeBie. What these inefficiencies are will be discussed in the next section.

Problem Statement

To provide context, we would like to reiterate that intermittent devices receive power by harvesting ambient energy. Although the variability and unreliability of power is the root issue with energy harvesting systems, this can not be solved. However, it does provide context as to why the following issues do not exists with conventionally-powered BLE devices.

The first problem arises when the connection is formed. When the *central*, for example, an Android phone, initiates a connection with a *peripheral*, in this case FreeBie, it gets to decide the initial connection settings. These connection settings define the baseline connection speed and, as a result, the power consumption of the peripheral. After the connection setup, the peripheral can request more suitable connection parameters. However, this means that during the connection setup, we have to contend with the default connection parameters of the central. In the case of a phone running the Android operating system (OS), these settings are high-throughput and high power consumption.

These unfavorable connection parameters forced FreeBie's designers to use capacitor sizes within FreeBie that, after the connection setup is finished, go mostly unused [6]. This means the device as a whole increased in size and cost.

The connection setup itself also has room for improvement. In the naive approach used in common frameworks, like Packetcraft and Zephyr, the connection setup is long and repetitive. For even a simple application, the number of packets required to set up can easily exceed 70 packets, which can be seen in Figure 4.1. At Android's fast settings, this already takes a few seconds. Using slower settings, which are more favorable for these ultra-low power devices, the setup can take up to three minutes. With more complex applications, these numbers increase rapidly.

Considering the variability of the power harvesting methods that these devices employ, it is often advantageous to switch to a faster connection configuration when plenty of energy is available or even crucial to switch to a slower connection configuration to protect against a power failure. Using the procedures available within the BLE specification, it takes a fixed six packets for the central to apply the connection settings after accepting them from the peripheral. This takes anywhere between tens of milliseconds to a minute, depending on the active connection parameters.

Take as an example an intermittent device that takes its power from solar. If someone were to accidentally block the solar panel, then at the lowest connection settings, the device would not be able to throttle the connection fast enough to protect itself against a power failure. From all these remarks, we can gather at least three possible goals to improve BLE for use within intermittently-powered devices:

1. **Goal 1**: Allow the Peripheral to dictate the initial connection parameters.
2. **Goal 2**: Reduce the time required for the connection setup.
3. **Goal 3**: Reduce the time required to apply new connection parameters.

This thesis will address these the above three points with the general goal of improving the viability of BLE for intermittently-powered devices.

Chapter 2 covers the necessary background information. Chapter 3 will explain the architecture of the proposed solutions. Chapter 4 validates and evaluates the efficacy of these solutions. Chapter 5 will discuss future work. Finally, Chapter 6 will draw a conclusion from the evaluation.

Chapter 2

Background

In 1996, Intel, Ericsson, and Nokia came together with the goal of standardizing short-range radio technology for connectivity between different products across many industries. The resulting technology was codenamed Bluetooth, lending its name to King Harald "Bluetooth" Gormsson, who was known for uniting Denmark and Norway in 958 and, more importantly, for his dead blue tooth [23]. From this collaboration, the Bluetooth Special Interest Group (SIG) was formed, which released its first version of the standard in 1999. Today, Bluetooth SIG has over 36 thousand members [22].

Bluetooth 4.0 first introduced Bluetooth Low-Energy (BLE) with a focus on low-power devices like wireless headphones and location beacons. BLE was not intended to replace the classic Bluetooth, now called Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR), but to live alongside it and expand on the feature set available.

Since its inception, Bluetooth has become one of the most ubiquitous wireless communication technologies, reaching nearly 100% market saturation in platform devices (phones, tablets, and PCs) [21]. In 2022, over 5.1 billion devices were shipped with Bluetooth radios, projected to grow to 7 billion by 2026. By that time, Bluetooth SIG expects 95% of all Bluetooth devices to support the LE-variant, highlighting BLE's importance.

In this chapter, the background information will be covered to understand the architecture of the solutions proposed to fix the inefficiencies mentioned in Chapter 1. This information includes a high-level overview of the BLE stack, an explanation of the Operating Systems (OS) used to implement the solutions, and some useful definitions. For a more comprehensive explanation of BLE and how to implement it, we refer to Getting Started with Bluetooth Low Energy [26].

2.1 Wireless Connection

Within the Bluetooth Low-Energy connection, there exists a clear hierarchy, where one device is dominant and is called the *central*. The central is in control of the connection and is usually a phone or a laptop. The other side of the connection (one or more) is called the *peripheral*. Typical peripheral devices include headphones, smart watches, IoT sensors, and push buttons.

For this thesis, we define three phases for a BLE connection. These phases are:

- *Unconnected*: Before any connection is established between the central and the peripheral.
- *Connection Setup*: Starts from the connection request and ends when the last non-application packet is sent.
- *Application*: Starts when connection setup is finished, and only application packets are sent. Application packets are packets that are necessary to fulfill the application.

These terms are not officially defined by the BLE specification but are used throughout this thesis.

2.2 Connection Setup

As previously defined in Section 2.1, the connection setup starts from the Connection Request and ends when the last non-application packet is sent. The packets that occur during Connection Setup can be divided into four groups which correspond to the subjects discussed in the previous sections.

- Link Layer
- Service Discovery
- Configuration
- Application

The order in which they occur is usually Service Discover, Configuration, and then Application, with Link Layer communication starting parallel to the Service Discovery from the start.

2.3 Architecture of the Bluetooth Low-energy Stack

As seen in Figure 2.1, the BLE stack is divided into many layers with increasing levels of abstraction. The bottom-most layers, the Physical Layer (PHY) and Link Layer (LL), form the controller, which is responsible for controlling the connection. The top-most layers, the Generic Access Profile (GAP), Security Manager (SM), Generic Attribute Protocol (GAP), and Attribute Protocol (GATT), provide higher level functionality and APIs which, together with the Logical Link Control and Adaption Protocol (L2CAP) form the host.

The separation of the host and controller is derived from how Bluetooth BR/EDR is implemented, where the host and controller can be implemented separately or even exist on different chips entirely. The host and controller are connected together using a Host-Controller Interface. This separation also allows a single radio to be used simultaneously with a BR/EDR and BLE controller and a generic host to support both versions of Bluetooth.

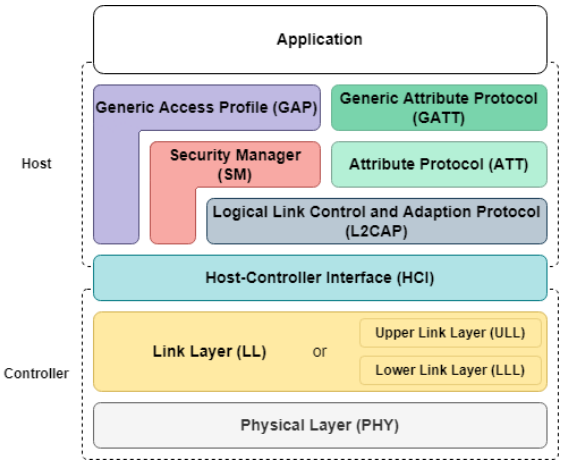

Figure 2.1: **The BLE Network Stack. The controller is comprised of the Physical Layer and the Link Layer. The Generic Access Profile, Security Manager, Generic Attribute Protocol, and Attribute Protocol messages get translated by the Logical Link Control and Adaption Protocol and together form the host. The Host-Controller Interface exists between the host and controller.**

Nordic Semiconductors supplies the SoftDevice for its microcontrollers, which contains the host and controller in a single qualified, pre-compiled binary [14]. The Zephyr framework can interface with Nordic's SoftDevice Controller through HCI or, since version 3.0.0, supports its own custom Zephyr BLE Controller. Packetcraft implements the entire stack from application to PHY.

2.3.1 Physical Layer

The PHY operates at the unlicensed 2.4GHz ISM (industrial, scientific, and medical) band and uses Gaussian frequency-shift keying (GFSK) modulation with adaptive frequency-hopping to reduce collisions. The 2.4GHz band is divided into 40 channels from 2.4000GHz to 2.4835GHz. The radio calculates the next hop using the formula:

$$\text{channel}_{\text{new}} = (\text{channel}_{\text{current}} + \text{hop}) \text{ modulo } 37$$

where the value *hop* is exchanged when the connection is established.

BLE has a theoretical maximum throughput of 1Mbps for version 4.2 and 2Mbps for version 5.0. However, in practice, this lies much lower. In normal operation (uncoded), every bit is represented by one symbol. From version 5.0 onwards, BLE supports *Coded PHY* where a single bit is represented by 2 to 8 symbols for improved range but reduced throughput.

2.3.2 Link Layer

The link layer (LL) directly controls the PHY and manages the link state. The link layer is the only hard real-time layer of the BLE stack and is partly hardware dependent. For this reason, it is usually implemented and kept separate from

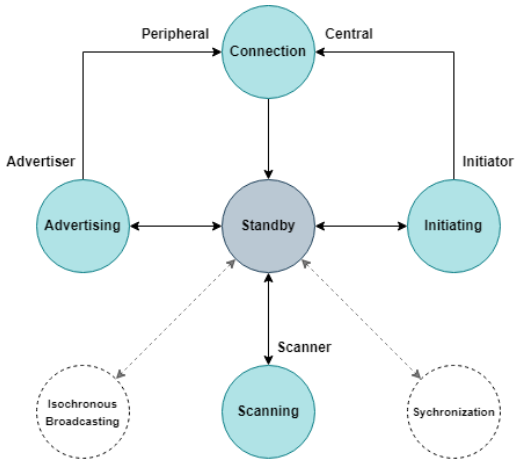

Figure 2.2: **Link Layer state machine diagram. In the standby state, no packets are received or transmitted. Advertisers send advertising packets without a connection, scanners receive advertising packets without the intention of connecting, and initiators initiate a connection with advertisers, which then become the centrals and peripherals, respectively.**

the rest of the stack. However, Zephyr goes even further and splits the link layer into an upper and a lower link layer (ULL and LLL, respectively), where the ULL is hardware agnostic and generic, and the LLL is hardware specific.

States and Roles

The link layer can be described as a system of one or more state machines. The states that each LL state machine can occupy are shown in Figure 2.2. The link layer should have at least one LL state machine that supports *Advertising* or *Scanning*.

The link layer in *standby* does not receive or transmit any packets. When the link layer moves from *standby* to *advertising*, the link layer takes on the role of *advertiser*. As an advertiser, the link layer transmits advertising channel PDUs and possibly responds to requests from other devices for more data. The different types of advertising packets and their content will be discussed later.

A link layer in the *scanning* state takes on the role of a *scanner*. In this role, it can listen for advertising packets and consume their content, but it can not form a connection with the *advertiser*.

If a device intends to form a connection with an advertiser, it should enter the *intitiating* state, taking on the role of *initiator*. When an advertisement packet is received, the *initiator* can respond with a `CONNECT_IND` packet to form a connection. After a connection is formed, the *initiator* becomes the *central* and the *advertiser* becomes the *peripheral*. The *central* defines the timing of transmission events.

The *synchronization* state is used to listen to periodic advertising trains from a specific device within its *Broadcast Isochronous Group* (BIG). The *isochronous broadcasting* state is used to broadcast data to a group of devices in a BIG.

Advertising Type	Connectable	Scannable	Directed
ADV_IND	Yes	Yes	No
ADV_DIRECT_IND	Yes	No	Yes
ADV_NONCONN_IND	No	No	No
ADV_SCAN_IND	No	Yes	No

Table 2.1: **Primary advertising PDU types.**

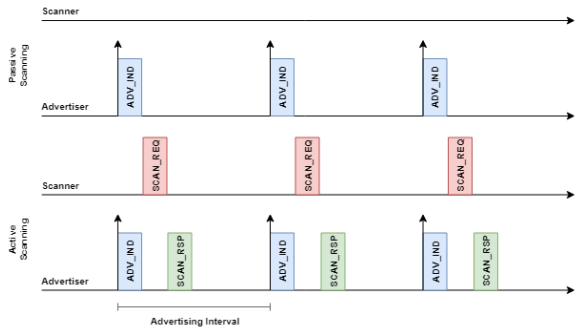

Figure 2.3: **When a scanner receives a scannable advertisement PDU while passively scanning, no further action is performed. However, when actively scanning, the scanner will perform a *Scan Request*, forcing the advertiser to send more data in the form of a *Scan Response*.**

These states are used when a single device continuously streams data to multiple receivers, such as wireless earphones. Figure 2.2 shows these states with dotted borders since they do not apply to this thesis.

Advertising and Scanning

As an advertiser, the device sends out advertising channel packets at a set interval between 20ms and 10.240ms. The advertiser does this for device discovery, to broadcast data or both. There are four primary types of advertisement packet types. These types are shown in Table 2.1.

These advertisement types can have three traits: *connectable*, *scannable* and *directed*. A connectable advertisement type allows initiators to establish a connection using the `CONNECT_IND` PDU. Packet types without the connectable trait are only used to broadcast data.

A scannable advertisement type allows a scanner or an initiator to request more data from the advertiser using a *scan request* or `SCAN_REQ` PDU. This forces the advertiser to respond with a *scan response* or `SCAN_RSP` PDU, which doubles the amount of data that can be broadcast. In this process, the scanner or initiator is said to perform *active scanning*. On the contrary, *passive scanning* is where the scanner or initiator only listens to advertisement packets and never issues a scan request. The difference between active and passive scanning is show in Figure 2.3

A scanning device can listen for these packets and do nothing, which is called *passive scanning*, or can request more data with a *Scan Request*, which is called

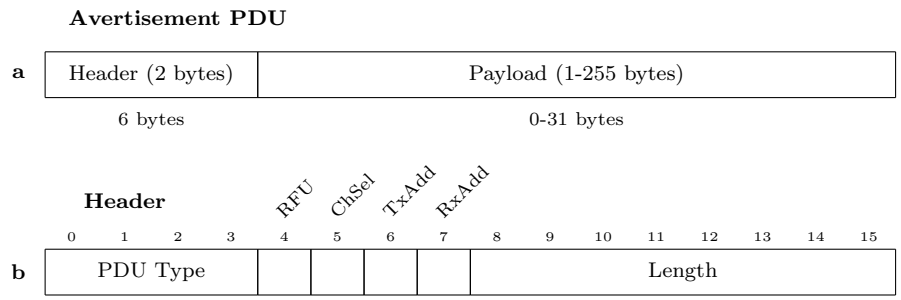

Figure 2.4: **a) Layout of the advertising PDU. b) layout of the advertisement PDU header. PDU Type as defined in Table 2.1. The Length field contains the length of the Payload field in bytes.**

active scanning.

Figure 2.4 shows the content of a generic advertising channel PDU. The first two bytes of the PDU contain the header, followed by 1 to 255 bytes of payload data. The four least significant bits of the header define the type of the advertising channel PDU and refers to the types defined in Table 2.1. The `length` field contains the length of the Payload field in bytes.

The payload for the four advertising channel PDU types defined in Table 2.1 is shown in Figure 2.5a. As seen in Figure 2.5b, *AdvData* can contain any number of *AD Structures*. The AD Structure's `AD Type` field is an 8-bit identifier that refers to one of the many predefined advertisement datatypes. This is followed by the length of the data for this type and then the actual data. The payload of the scan request is shown in Figure 2.6. The payload contains a *ScanA* and *AdvA* field, which contain the address of the scanner and advertiser, respectively. The payload of the *Scan Response* or `SCAN_RSP` PDU is the same as the advertising channel PDUs in Figure 2.5.

Some examples of AD Types include *Local Name*, *TX Power Level*, and *Appearance*. The *Local Name* type allows the device to share its locally assigned device name. For example, one might name its headphones "John's oPods", which scanners can display to the user to identify the device. The *TX Power Level* type can be used to calculate a rough distance estimation between the advertiser and the scanner, and the *appearance* type can tell scanners the general appearance of the advertiser (smartwatch, for example).

Connection Setup and Parameters

The connection setup is initiated by the central by responding to an advertising channel PDU (for example, `ADV_IND`) from an advertiser with a *Connection Request* or `CONNECT_IND` PDU. From this moment, a connection between the devices is made, and the setup can begin. If the central receives no empty PDU back as an acknowledgment of the connection request, then the central will retry for a specific (user-defined) number of attempts.

Figure 2.7a shows the payload of the `CONNECT_IND`. The payload consists of an InitA, AdvA, and LLData field, which contain the address of the initiator and advertiser, and the link layer configuration, respectively. As can be seen

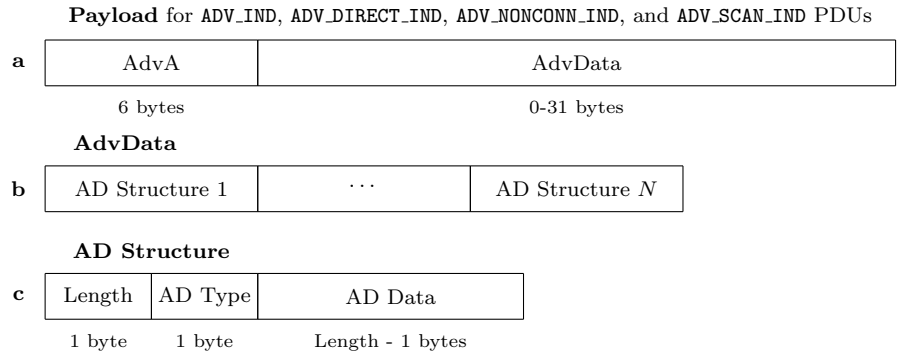

Figure 2.5: **a) Advertising channel PDU payload for advertising types of Table 2.1. b) AdvData can contain as many AD Structures (contained pieces of advertisement data) as fits in 31 bytes. c) The AD Structure starts with the length of the AD Type (type of advertisement data as defined as defined in the Core Specification Supplement [3]) and AD Data fields combined, followed by the AD Type, and finally, the AD Data as defined in the specification for that AD Type.**

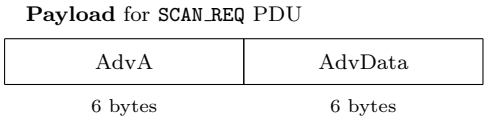

Figure 2.6: **The Payload field of a *Scan Request* or SCAN_REQ PDU consists of ScanA and AdvA fields. The ScanA field contains the address of the scanner, and the AdvA field contains the address of the scanner.**

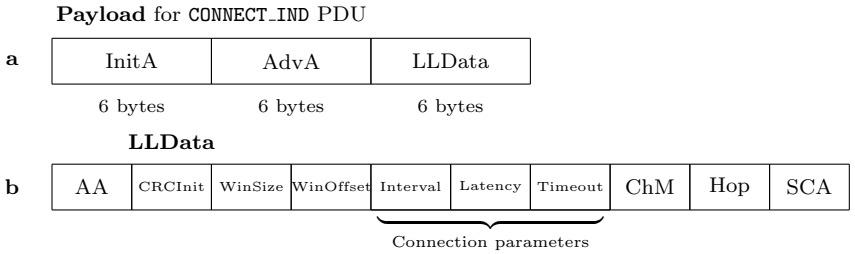

Figure 2.7: **a) The payload consists of an InitA, AdvA, and LLData field, which contain the address of the initiator and advertiser, and the link layer configuration, respectively. b) The content of the LLData field in a, containing link layer configuration parameters (like the *connection parameters*, for example).**

in Figure 2.7b, the LLData field can be further split up into the following ten fields:

- **AA**: Contains the Access Address, which is the identifying address of the link used by both devices.
- **CRCInit**: Contains the initialization value for the *cyclic redundancy check* (CRC), which is an algorithm that can be used to check if a piece of data has been corrupted.
- **WinSize**: Contains the transmission window size in units of 1.25ms. Used to allow the central to schedule transmission events for multiple peripherals efficiently.
- **WinOffset**: Contains the transmission window offset in units of 1.25ms. Used to allow the central to schedule transmission events for multiple peripherals efficiently.
- **Interval**: Contains the *connection interval* in units of 1.25ms.
- **Latency**: Contains the *peripheral latency*, which is unitless.
- **Timeout**: Contains the *supervision timeout* in units of 10ms.
- **ChMap**: Contains a bitmask where each bit represents a *used* (1) and *unused* (0) channels.
- **Hop**: Contains the *hop* value used in the formula examplained in Section 2.3.1.
- **SCA**: Contains the worst-case *sleep clock accuracy* of the central. The SCA determines how much slack should be accounted for when waking up for the next transmission event.

The most important parameters for the connection timing are *connection interval* (CI), *peripheral latency* (PL), and *Supervision Timeout* (ST). The connection interval defines the time between periodic *connection events* (CE). A connection event starts with a packet transmission from the central. Every packet from the central has to be followed by a packet from the peripheral. Multiple transmissions can be chained together in a single connection event. The peripheral latency defines how many connection events can be skipped by the peripheral to conserve energy. If a valid packet has not been received for more time than the supervision timeout defines, then a connection is considered broken. For example, take a CI of 1 second and a PL of 0. In that case, the central transmits a packet every second, and the peripheral responds. Now take a CI of 1 and a PL of 1. In this case, the central transmits every second, but the peripheral is allowed to sleep every other packet, effectively making the time between transmissions two seconds. The second example is displayed in Figure 2.8. The units and ranges for the three connection parameters are defined in Table 2.2.

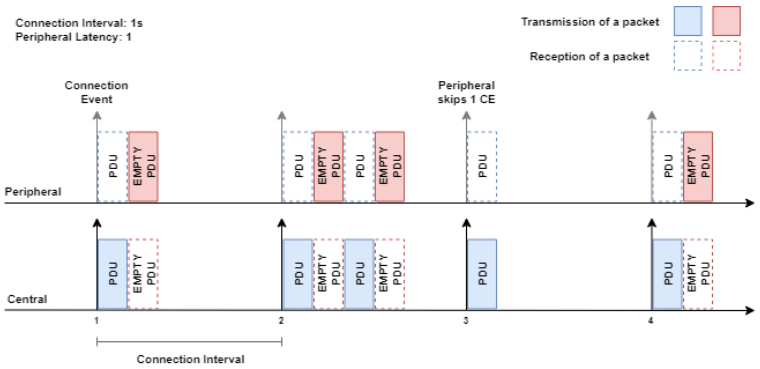

Figure 2.8: **Example of a timeline between a central and peripheral with CI=1s and PL=1. The CE at 1 second shows a normal connection event with one packet from the central and one response from the peripheral. The CE at 2 seconds shows how multiple transmissions can be changed during a single event. Finally, the CE at 3 seconds shows the peripheral using its PL of 1 to sleep during an event.**

	Range	**Unit**	**Real Range**
CI	0x0006 to 0x0C80 (hexadecimal) 6 to 3200 (integer)	1.25ms	7.5ms to 4000ms
PL	0x0000 to 0x01F3 (hexadecimal) 0 to 499 (integer)	N/A	N/A
ST	0x000A to 0x0C80 (hexadecimal) 10 to 3200 (integer)	10.00ms	100ms to 32000ms

Table 2.2: **The three connection parameters which define the timing of connection events [2]. CI is the time in between connection events. PL defines how many connection events can be skipped by the peripheral. ST is the maximum time allowed since the last packet after which a connection is considered lost**

Feature Expansion and Compliance

Newer revisions of BLE can add new features to the link layer, adding new functionality or improving throughput. A vendor can also choose not to implement functionality not required for their application to reduce complexity. The supported features are stored in a 64-bit bitmask. The bitmask needs to be exchanged at the beginning of the connection to make sure a controller on one side does not use a procedure that the other controller does not support. Some examples of these features include *LE Encryption* (bit 0), *LE Data Packet Length Extension* (bit 5), *LE 2MB PHY* (bit 11), and *LE Coded PHY* (bit 11), which might sound familiar[2, p. 2827].

A feature that has been mentioned in the list above but not yet addressed is *LE Data Packet Length Extension*. By default, the link layer packet data unit (PDU) allows for 27 bytes of data to be transmitted. The LE Data Packet Length Extension feature allows this to be increased up to 251 bytes for vastly improved throughput. However, higher-level protocols like ATT are encapsulated within the LL PDU, so the effective application data per packet will be lower than 251 bytes.

2.3.3 Host Controller Interface

The host controller interface provides a standardized way of communicating between the hardware-specific, real-time part of the stack (controller) and the rest, which is more hardware agnostic (host). The HCI can be implemented as a software API if the host and controller exist within the same silicon or using a hardware interface (UART, SPI, I^2C, USB) when the controller is a separate chip. It is common that the host and controller are implemented on the same chip since that reduces power consumption, which is often a large consideration for BLE devices.

The BLE specification defines a set of standardized HCI commands and events that the host and controller exchange with each other, together with a data packet format and control flow rules[26]. Some vendors choose to extend the functionality of HCI by adding custom commands [24], which can allow the developer to control low-level settings of the radio.

2.3.4 Logical Link Control and Adaption Protocol

The task of the Logical Link Control and Adaption Protocol (L2CAP) layer is twofold. First, it encapsulates upper-level protocol packets into the standard LL PDUs, functioning as a multiplexer. The converse of this process is called decapsulation. The second function of L2CAP is fragmentation, which happens when an upper-level protocol packet exceeds the supported LL PDU size and has to be split into multiple L2CAP fragments. The L2CAP layer of the receiving side then has to defragment the packets to create the original packet. L2CAP adds a four byte header to each packet it encapsulates, thus reducing the effective packet data size further from 27 to 23 bytes.

2.3.5 Security Manager

The Security Manager (SM) layer contains both a protocol and a group of security-related algorithms and procedures. The algorithms are used to generate

security keys that can be distributed using the defined procedures. Afterward, the Security Manager Protocol (SMP) can be used by other layers to safely connect and exchange data.

The following procedures are supported by the Security Manager:

- *Pairing* is when a set of temporary security keys are generated to encrypt the link.
- *Bonding* can be done after *pairing* by exchanging permanent security keys, which are stored in non-volatile memory.
- *Encryption Re-establishment* is done to re-use previously stored permanent keys for a new encrypted link without having to *pair* and *bond*.

The pairing process requires the user to confirm the connection by entering a pin code from the peripheral on the central.

2.3.6 Generic Access Profile

The Generic Access Profile (GAP) layer defines the generic procedures related to the discovery of BLE devices and link management aspects of connecting to BLE devices. In addition, it also defines procedures required to attain certain security levels, as well as standard format requirements for parameters that are available at the user level. Most importantly, GAP defines certain *roles* which govern the hierarchy of a BLE network, resulting in the asymmetric power requirements that allow the peripheral devices to operate efficiently.

Roles

GAP defines two pairs of roles for a total of four GAP roles. These pairs are the *broadcaster* and *observer* pair, and the *peripheral* and *central* pair.

A broadcaster is a device with a link layer set to advertising that only transmits data through advertisements of the *non-connectable*. A device in the *broadcasting role* should have a transmitter, but a receiver is optional. Devices that are suitable for broadcasting include temperature sensors and locating beacons.

An observer is a device that has its link layer state set to scanning and is only interested in reading advertisement packets without pursuing a connection. An observer should have a transmitter, but a receiver is optional. Phones are often observers when listening for BLE locating beacons to improve localization accuracy.

A peripheral (GAP role) is any device that accepts connection requests from other devices. This means any device which is an advertiser sending out advertisements of the connectable type. When a peripheral (GAP role) goes to the connected link layer state, it changes from an advertiser (LL role) to a peripheral (LL role). A device in the *peripheral role* should have a transmitter and a receiver.

A central (GAP role) is any device that initiates a connection with a peripheral. When central (GAP role) goes to the connected link layer state, it changes from an initiator (LL role) to a central (LL role). A device om the *central role* should have a transmitter and a receiver.

Depending on the application requirements, a device may support multiple roles simultaneously. For example, a phone can be connected with wireless

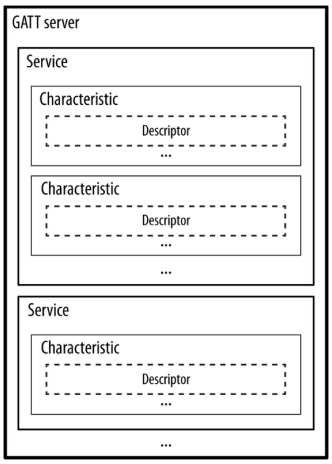

Figure 2.9: **The architecture of the GATT Server. Functionality is grouped as services. Services contain data as characteristics that can be read from or written to. Descriptors are used as metadata to describe characteristics and for configuring the server to notify the client of updates from the characteristic they are grouped under (Figure taken from [26]).**

headphones, making it a central, and listening for locating beacons, making it an observer.

2.3.7 Attribute Protocol and Generic Attribute Profile

The Attribute Protocol (ATT) layer allows information to be discovered and transferred between BLE devices in a structured manner. ATT uses a client/server model, where the data is stored in the ATT server as attribute, and the client requests to read or manipulate the attribute data. Each attribute contains a known UUID used to identify the type of data contained in the attribute, and a 16-bit handle to identify the attribute itself.

The *Generic Attribute Profile* extends the functionality of ATT by creating a hierarchy and data abstraction model on top of it. The GATT server groups attributes into *Services*. Services contain data points called *Characteristics*. These characteristics can be interpreted and configured using fields called *Descriptors*. See Figure 2.9 for a schematic layout of the GATT Server.

For a real-world example, see the Heart Rate Service (HRS) in Figure 2.10. The HRS allows a Client to read the heart rate sensor of a device. The HRS contains a characteristic called Heartrate. One of the descriptors tells us that the unit is defined as **beats per minute**. We could read the characteristic value manually, but if we would like to be notified when a new measurement is done, then we can set the Notify bit of the *Client Characteristic Configuration* or CCC descriptor.

To be able to read or write to a characteristic, we need its handle. A handle is a number that is unique for a characteristic within a GATT Server. To find this handle, we can perform a *Find By Type Request* using its 16-bit Universally Unique Identifier (UUID). Bluetooth SIG has predefined UUIDs for many

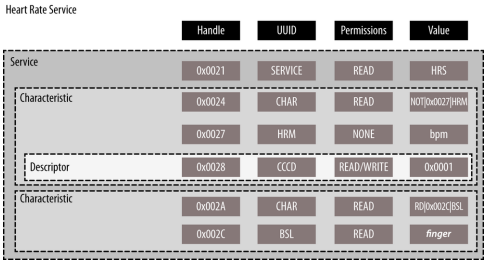

Figure 2.10: **An example of a service. Each element is searchable through its UUID and then uniquely identifiable using its consecutively numbered handle [26].**

predefined Services. Each predefined service has a specification that defines the shape of the service. This includes all the characteristics, descriptors, and their respective UUIDs.

The process of finding all these handles is what is called *Service Discovery*. The result of this process is a list of handles that can be used to read and write to the server. After Service Discovery is done, the GATT Servers need to be configured. This usually means writing to the CCC descriptor to configure notifications for the desired characteristics. This process will, from here on, be referred to as Configuration.

Both the Central and the Peripheral can be Servers and Clients at the same time. For example, a phone can provide the central time for a smartwatch to display on the watch face, while the smartwatch measures the heart rate for the phone to present within a health application. This means that a Service Discovery and Configuration need to be performed by both the Central and the Peripheral.

2.4 Operating Systems Used to Implement the System Proposed

The FreeBie project was originally developed using the open-source BLE stack Packetcraft [16]. Packetcraft is a *real-time operating system* (RTOS), which means it allows for real-time scheduling of tasks. Real-time scheduling is essential to run the link layer controller. Due to its architecture, it was especially suitable to be modified in the way required for intermittent operation. This is because all OS tasks are managed using a single scheduler which uses a generic sleep method. This sleep method was altered to configure the real-time clock (RTC), save the memory to FRAM, and fully power down the System-on-Chip (SoC).

However, since FreeBie was originally developed, Packetcraft has gone into closed-source development. For this reason, a newer RTOS was chosen for the central, called Zephyr. Zephyr OS is a modern RTOS developed by the Linux Foundation and is now the officially supported stack for Nordic Semiconductors [25]. A side-effect of implementing the proposed solutions on two platforms is proving that the approach used to solve the issues outlined in Section 1 is platform-agnostic.

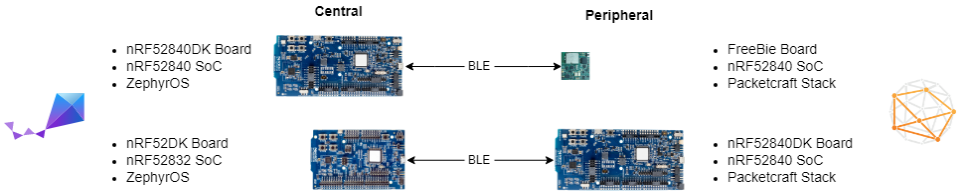

Figure 2.11: **The combination of central and peripheral on top is used for testing, while the combination on the bottom is used for development.**

2.5 Demo System

The modifications we aim to make to the system will attempt to remove certain procedures without losing their functionality. We have developed a demo system to verify that these functions still work. The demo system consists of a single central and peripheral, shown in Figure 2.11. The configuration on top consisted of an nRF52840DK as a central and the FreeBie as a peripheral and was mostly used when verifying new code, when intermittency was required, or during the final testing phase and evaluation. The configuration on the bottom was primarily used during development since the FreeBie codebase runs on any nRF52840DK when intermittent operation is disabled. This configuration allows us to access the serial debug trace of Packetcraft.

We developed a Packetcraft application for FreeBie to act like a smartwatch peripheral and a Zephyr application for the central to work like a stand-in for a phone. Appendix A contains a list of all available terminal commands, a list of all available services and characteristics, as well as a description of how to use the demo system and how it was developed.

Chapter 3

Architecture

This chapter will cover the architecture of the improvements proposed to fix the inefficiencies mentioned in Chapter 1. The improvements have been divided into four stages, where every successive stage contains the new improvement as well as the improvements of the previous stage. This is because some improvements build on top of changes from the previous stage and cannot exist independently. Consequently, the order of these stages also corresponds to the order in which the improvements have been implemented. For reference, these stages are:

	Stage			
Improvement	**1**	**2**	**3**	**4**
Connection Parameter Sharing	**Yes**	**Yes**	**Yes**	**Yes**
Service Discovery Cache	No	**Yes**	**Yes**	**Yes**
Fast Reconnect	No	No	**Yes**	**Yes**
Skip Reconfiguration	No	No	**Yes**	**Yes**
Feature Exchange Cache	No	No	No	**Yes**
Data Length Extension Cache	No	No	No	**Yes**

Table 3.1: **Four stages of optimization.**

3.1 Peripheral Dictating Connection Parameters

In the following section we present the solution to address **Goal 1** mentioned in Section 1.

3.1.1 Stage 1: Connection Parameter Sharing

There is no standard procedure for providing control to the peripheral when deciding the initial connection parameters. The GAP service does provide a characteristic definition called the *peripheral preferred connection parameters* (PPCP) [2]. The fields contained in the PPCP characteristic are shown in

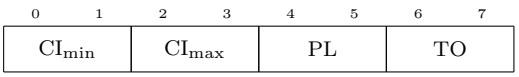

Figure 3.1: **Format of the *peripheral preferred connection parameters* characteristic value of the GAP service as defined by the Bluetooth Core specification [2].**

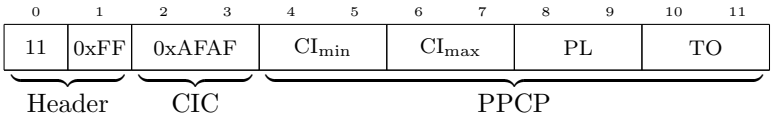

Figure 3.2: **AD Stucture. The meaning of each field is as follows. Byte 0 contains the number of remaining bytes in the AD Structure, and byte 10 contains the AD Type value for *Manufacturer-Specific Data* in hexadecimal, which together form the header. Byte 2-3 contains the custom Company-Identifier Code we defined. Finally, byte 4 to 11 contain the PPCP as defined in Figure 3.1.**

Table 3.1. However, this can only be read once the connection has already been established [2, p. 1361].

The way we share the PPCP before a connection has been established is by using the `AdvData` field within the advertisement packets. The specification defines an AD Type called *Service Data*, which allows data from a service to be shared as advertisement data. See Figure 2.5 for an explanation of how advertisement channel payloads are formatted. The format for sharing data provided by a service has to be defined in specification for that service. However, the GAP service does not define this for the PPCP characteristic.

In order to be BLE compliant, the AD Type *Manufacturer-Specific Data* [3], with hexadecimal value OxFF, is used. This AD Type contains a Company Identifier Code (CIC) in the first two bytes, followed by data part of *Manufacturer-Specific Data*. The CIC is a 16-bit identifier which allows us to discern between our custom data versus that of another company. For the sake of this thesis we used a CIC of 0xAFAF. The complete AD structure is shown in Figure 3.2.

Now that the data is present in the advertisement packets, we need a way to parse it on the side of the Central. Algorithm 1 shows how this works. Essentially, the first procedure, `ON_DEVICE_FOUND`, is called every time the central receives a new advertisement packet. For each new MAC address that is detected, a new device entry is added to the device list. After that, any advertised data is parsed by the central using the `PARSE_ADV_DATA` procedure and added to the device entry within the device list. If the peripheral corresponding to a device entry has been flagged as the target device on the central, the central retrieves the appropriate connection parameters and responds to the advertisement packet with a connection attempt.

To parse the data field of the advertising packet, two things happen. First, the 16-bit CIC is compared with the one we have defined ourselves. This ensures that data from another vendor is ignored. If that is correct, then the remaining

8 bytes, which contain the PPCP, are copied by the central into the device list entry structure, containing a 16-bit integer for each field of the PPCP.

Algorithm 1 Parsing and usage of PPCP by Central

```
1: procedure ON_DEVICE_FOUND(addr, data)
2:     if addr not in record then
3:         create entry in record for addr
4:     end if
5:     dev ← find entry for addr
6:     PARSE_ADV_DATA(dev, data)
7:     if dev = target_dev then
8:         if dev contains ppcp then
9:             params ← pcpp(dev)
10:        else
11:            params ← default parameters
12:        end if
13:        CONNECT(dev, params)
14:    end if
15: end procedure
16: procedure PARSE_ADV_DATA(dev, data)
17:    for ad_structure in data do
18:        if type(ad_structure) = 0xFF then
19:            if cic(ad_structure) = 0xAFAF then
20:                ppcp(dev) ← store ppcp(ad_structure) in entry
21:            end if
22:        end if
23:    end for
24: end procedure
```

3.2 Reducing Connection Establishment Time

The solutions within the following three sections target **Goal 2** of Section 1. In general, these solutions try to decrease the connection setup time by decreasing the number of packets being sent. This can be done since, in most cases, from one connection to another, the same actions need to be taken to set up the connection. By caching the results of these actions on both sides of the connection and defining when this cache can be reused, the connection setup can theoretically be brought back to a single packet.

3.2.1 Stage 2: Caching Service Discovery

For even the most trivial applications, the process of service discovery contributes by far the most packets to the connection setup time. As Figure 4.1 shows, even in our relatively simple demo system, nearly 70% of the packets are for service discovery. The BLE specification provides some methods of detecting chances within the GATT server.

A simplified version of the process of service discovery is captured in Algorithm 2. First, a *Read By Type Request* is sent using the UUID of the service

that needs to be discovered. We can define a search range for the GATT server to look between. Since we have not discovered anything prior, the range is set to the maximum possible values that a handle can have, which are 0 and 65536. The result of this request will be the first handle of the service (which points to the service itself) and the last handle of the service. See Figure 2.10 for reference. The start and end handle provide the new bounds for the *Read By Type Request*, which is sent for each characteristic and descriptor within the service.

Algorithm 2 Service Discovery within a GATT server

```
1: procedure DISCOVER_SERVICES
2:     for service in services do
3:         send read_by_type_request(uuid(service), 0, 65536)
4:         receive start, end ← read_by_type_response()
5:         for characteristic in characteristics(service) do
6:             send read_by_type_request(uuid(characteristic), start, end)
7:             receive handle ← read_by_type_response()
8:         end for
9:     end for
10: end procedure
```

At the end, when this process has finished, the result is a list of 16-bit integer values which correspond to all the discovered characteristics. This list is the information that needs to be cached.

The GATT service provides a characteristic called the Database Hash to detect changes within the GATT server. This is a 128-bit value containing a **AES-CMAC** hash calculated from the database structure. If any single handle is changed or the order of the services is different, then the hash will be different, and the client must rediscover the database. If this hash is the same, then it is safe to assume that all handles are the same as before.

Packetcraft

Packetcraft provides a framework for service discovery. This framework contains hooks for the developer to add platform and application-specific code to implement generally useful functionality, like discovery, caching, and configuration. This is done by registering a callback that contains code to handle state changes, which is explained in Section A.2.1 of Appendix A.

Within the stage change callback, there are two states that need to be modified to enable caching. During the initialization state (*APP_DISC_INIT*) state two pointers need to be registered that point to the handle list and the database hash. Afterward, in the completion state (*APP_DISC_CMPL*), when the database hash has been read, and the new database is discovered, these can be stored.

For most systems, this means reading from and writing to a form of non-volatile storage. However, since the Peripheral (FreeBie) dumps its memory to FRAM every time it goes to sleep, all its memory is functionally non-volatile. So to support caching of database handles in Packetcraft, we need to create two arrays in global memory and provide these during the initialization state within the state change handler and copy memory back to those pointers when discovery has been completed.

Zephyr

For Zephyr, this is a bit more involved and requires some custom logic to reach the same functionality as is provided in Packetcraft. The logic is captured in Algorithm 3. Since the original code is made up of multiple different functions chained together as asynchronous callbacks, these callbacks have been placed inline in Algorithm 3 to provide a linear, more easily understood version.

Algorithm 3 Linear Version of the Asynchronous Database Cache Algorithm of the Central Device

```
1:  procedure ON_CONNECTED
2:      send READ_DB_HASH_REQUEST
3:      receive hash_remote ← READ_DB_HASH_RESPONSE
4:      if has_cache then
5:          if hash_local = hash_remote then
6:              handles ← READ_CACHED_HANDLES
7:              CONFIGURE_SERVICES(handles)
8:              return
9:          end if
10:     end if
11:     handles ← DISCOVER_SERVICES
12:     CONFIGURE_SERVICES(handles)
13:     WRITE_CACHE(hash_remote, handles)
14: end procedure
```

Reading and writing to the cache are done using Zephyr's NVS library. This library allows the developer to write to non-volatile flash memory using a simple interface. Data is stored as $(id, data)$ pairs. The id is a 16-bit integer and the $data$ can be an arbitrary size blob of 8-bit integers. After writing, the data can be read again using the unique id.

3.2.2 Stage 3: Fast Reconnect

After the database discovery has been completed, it is common for both devices to perform some form of configuration. This means reading the initial values of certain characteristics and writing CCC values such that the GATT server sends notifications when a characteristics value has changed.

In our demo system, configuration accounts for about eight packets. However, the amount of packets during configuration grows linearly with every characteristic that is necessary for the application. Aside from this, due to the Database Hash requests that are sent as a result of the solution in the previous section, four more packets are reintroduced into the connection setup.

Ideally, we would like to get rid of these packets altogether. However, this is not possible to do all the time. Creating a Hash of the CCCs would allow us to detect changes, but it would reintroduce another four packets to transfer the hashes. To remove these packets entirely, it is necessary to define a scenario where these checks and procedures can safely be skipped.

It is only safe to assume that all discovery and configuration has successfully occurred when currently in a connected state. This is where **Fast Reconnect**

comes in. When a connection is terminated from either side using the **Fast Reconnect** termination reason, with hexadecimal value 0x46, both the Peripheral and Central set the fast reconnect flag. If this flag is set when a new connection is created, then the configuration and Database Hash requests are skipped.

For both platforms, this means the following modifications need to be made:

1. Add a new valid disconnect reason called Fast Reconnect.
2. When a Fast Reconnect happens, make the CCCs persist across connections.
3. When a Fast Reconnect happens, skip the Database Hash request.

How these modifications are implemented for each platform is explained in the following sections.

Packetcraft

The first modification is done by adding a macro definition to the source file to define `HCI_ERR_FAST_RECONNECT` (this naming scheme follows the source's conventions) as **0x46**. There are no checks elsewhere in the source code which check if this is a valid disconnect reason.

To persist the CCCs, it is possible to abuse functionality that already exists as part of the specification. The BLE specification allows for persistent configurations when using a bonded connection. However, connection bonding is not possible using FreeBie. It is possible to disable the bond check entirely, thus enabling persistent CCCs for all connections.

Although the configuration is now persistent, the device will still redo the configuration when reconnecting. To fix this, the discovery framework is modified. The framework stores its state in a control block. This control block is created with the connection and subsequently destroyed when a connection is dropped. If the disconnect reason is set to Fast Reconnect when disconnecting, then the control block is kept in memory. When a new connection is opened, the same control block is reused, and the discovery framework is left in the same state as just before the devices disconnected. Since the discovery framework is also responsible for requesting the Database Hash, this solves both modifications 2 and 3.

Zephyr

The first modification is the same as with Packetcraft, but in the case of Zephyr, the macro has to be added to a list of valid disconnect reasons. As with Packetcraft, to enable persistent CCCs, the check for a bonded connection is also disabled. Algorithm 3 needs to be slightly altered for the remaining modifications. Algorithm 4 shows this updated variant.

3.2.3 Stage 4: Caching Link Layer Exchanges

After the optimizations done in the previous sections, only nine packets remain. These packets are the `CONNECT_IND`, which initiates the connection (cannot get rid of this), and eight packets that belong to link layer procedures. These procedures exchange link layer parameters such that each controller knows what the other controller supports. The two procedures are:

Algorithm 4 Altered Version of Algorithm 3

```
1: procedure ON_CONNECTED
2:     if has_cache is set and fast_reconnect is set then
3:         unset fast_reconnect
4:         return
5:     end if
6:     send READ_DB_HASH_REQUEST
7:     receive hash_remote ← READ_DB_HASH_RESPONSE
8:     if has_cache then
9:         if hash_local = hash_remote then
10:            handles ← READ_CACHED_HANDLES
11:            CONFIGURE_SERVICES(handles)
12:            return
13:        end if
14:    end if
15:    handles ← DISCOVER_SERVICES
16:    CONFIGURE_SERVICES(handles)
17:    WRITE_CACHE(hash_remote, handles)
18: end procedure
```

Fields	Content	Example
Max RX octets	Number of bytes	27
Max RX time	Microseconds	328
Max TX octets	Number of bytes	27
Max TX time	Microseconds	328

Table 3.2: **Content of LL_LENGTH_REQ and LL_LENGTH_RES.**

- Data Length Update Procedure
- Feature Exchange Procedure

The *Data Length Exchange Procedure* is required to increase the size of the Data Channel Packet Data Unit (PDU). This is the size of the packet as transferred over the air. See Table 3.2 for the contents of the request and response packets. When the application requests a read from a characteristic with a size larger than 27 bytes, then it first has to perform a Maximum Transferrable Unit (MTU) Exchange to increase the size of the ATT payload to a maximum of 251 bytes. However, if the Data Length Exchange has not yet increased the Data Channel PDU size, then the controller will divide the ATT payload in 27 byte L2CAP fragments. This is because the ATT payload is encapsulated within the Data Channel PDU.

The *Feature Exchange Procedure* might be initiated to exchange the Link Layer parameter for the currently supported feature set. The feature set of a controller is represented as a bitfield where each bit corresponds to a feature that is supported (1) or not supported (0). The feature exchange exchanges these bitfields, and the used feature set is the logical AND of these fields. Lets say the Central has $FeatureSet_A$ and the Peripheral has $FeatureSet_B$ then after the exchange $FeatureSet_{USED} = FeatureSet_A \wedge FeatureSet_B$.

Both procedures can be initiated by either the Central or the Peripheral. Although both procedures only need to be initiated once, they often occures twice. This happens because the Link Layer is programmed as an asynchronous state machine and these requests are all pushed into the transmission queue at the start of a connection before anything has been exchanged.

Although the BLE specification mentions that these procedures can be cached, neither Packetcraft nor Zephyr have implemented this functionality in their controller.

Packetcraft and Zephyr

Both Packetcraft and Zephyr's essentially work the same. Their link controllers keep a connection context for each connection. All Link Layer parameters are stored in this context. When a connection is created, this context is allocated, and when a connection is dropped, the context is freed. Like with the previous solutions, this is where the information will be cached.

As mentioned previously, the Link Layer is programmed as a state machine. During initialization, the state machine schedules all the procedures in a queue. To keep this from happening when the Link Layer parameters have been restored, a restore flag is added. Only when the restore flag is *unset* will the procedures be scheduled.

3.3 Faster Connection Parameter Adaption

When using **Fast Reconnect** with Stage 4 optimizations, the connection can be dropped and restarted significantly faster than a connection update request is applied. Aside from the obvious general advantages of quickly being able to set up a connection, this can also be used to change the connection interval (and thus the connection speed) much faster than the regular `Connection Update Request`.

The specification defines that, when `Connection Update Request` is accepted, the new parameters will be applied *at least* six packets after the current packet. In practice, however, this means that from the point of the `Connection Update Request` this usually takes about 10-11 packets. Even at a fast Connection Interval (for intermittent devices) of 1 second, this already takes more time than a **Fast Reconnect** cycle.

For a regular, battery-powered device, it is not a significant issue if updating the connection parameters takes several minutes since the battery reserve is large and it is only charged occasionally. However, for an intermittent device, which has a small energy reserve and harvests energy, this poses a bigger issue. You are left either going **1)** the conservative route and leaving a lot of potential performance (throughput) on the table but always having enough charge or going **2)** the greedy route and having the maximum performance but risking not being able to recoup the spent energy because of changing energy availability.

When FreeBie goes to sleep in between transmission events, it calls the `PalSysSleep` is called. This is a modified version of Packetcraft's sleep method, which handles checkpointing memory regions and configuring the RTC to wake up the microcontroller in time for the next scheduled system event. When the

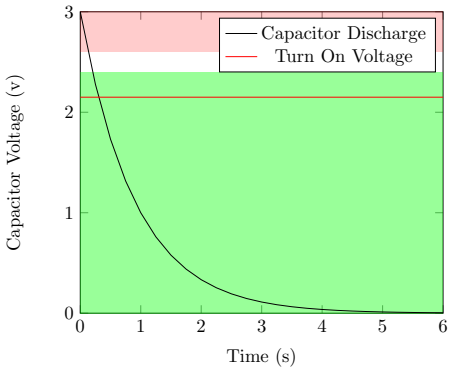

Figure 3.3: **The three different target charge rates. Red region: negative charge rate. White region: voltage is maintained. Green region: positive charge rate.**

system is woken up again, the checkpoint is restored, and the system resumes from within the sleep function.

When the system resumes, the voltage level of the capacitor can be measured using the onboard Analog to Digital Converter (ADC), and the connection rate can be adjusted appropriately.

3.3.1 FRAPPUCInO

Algorithm 5 shows the pseudocode of the algorithm. As the name would suggest, `AdaptConnParamsToVdd` contains the logic for calculating the new connection parameters based on the capacitor voltage.

The parameter that is controlled by the system is $preset_{new}$ and is an integer that represents the power level. This integer has a value between 0 and N and for each value maps to a *preset* within the *preset list*. Each *preset* contains the PPCP that corresponds to a certain connection speed and, as a result, power draw. All `presets` within the *preset list* are ordered on expected power draw from lowest power draw (0^{th} item) to highest power draw (N^{th} item).

The algorithm starts by calculating the slope of the voltage level over time. This is done by taking the difference in voltage between now and the last time the function was called (vdd_{diff}) and multiplying by the number of seconds that have elapsed since then ($rate_{actual}$). The slope of the voltage over time is essentially the charge/discharge rate.

A target rate $rate_{target}$ is selected based on the voltage level of the capacitor. If the voltage is low, then a positive charge rate is chosen. If it is high, then a negative charge rate is chosen. For anything in between, the charge rate is set to zero to try and maintain the voltage. Figure 3.3 visualizes these different charge zones.

The difference between $rate_{target}$ and $rate_{actual}$ is calculated and multiplied by a factor **P** which is then added to the current preset level. The method `LoadPreset` then selects a preset from a list of presets which are ordered in decreasing connection interval and thus in increasing power consumption.

`LoadPreset` copies the new preset into the advertisement data. If the preset is different than the one that was already loaded, a **true** is returned, otherwise, a **false** is returned. Afterward, the network can be dropped using **Fast Reconnect**, the device starts advertising again with the newly loaded connection parameters, and the host will reconnect using these new parameters.

Algorithm 5 Algorithm for Calculating New Connection Parameters

procedure PalSysSleep
    some code before...
    if not *conn_param_change_pending* **then**
        AdaptConnParamsToVdd
    else
        if not *net_restored* **then**
            $net\_restore\_pending \leftarrow$ **true**
        else
            Disconnect(FAST_RECONNECT)
        end if
    end if
    ...some code after
end procedure
procedure AdaptConnParamsToVdd
    $vdd_{current} \leftarrow$ MeasureCapacitorVoltage
    $vdd_{diff} \leftarrow vdd_{current} - vdd_{last}$
    if *conn_is_active* **and** *conn_setup_done* **then**
        if $skips = 0$ **then**
            $rate_{actual} = \frac{vdd_{diff} * ticks_{per\_second}}{ticks_{current} - ticks_{last}}$
            $rate_{target} \leftarrow$ SelectChargeRate($vdd_{current}$)
            $rate_{error} \leftarrow rate_{target} - rate_{actual}$
            $adjustment \leftarrow P * rate_{error}$
            $preset_{new} \leftarrow preset_{current} + adjustment$
            if *adjustment* **and** LoadPreset($preset_{new}$) **then**
                $conn\_param\_update\_pending \leftarrow$ **true**
                $skips \leftarrow 1$
            end if
        else
            $skips \leftarrow skips - 1$
        end if
    end if
end procedure

Adjusting the preset based on the charge rate error essentially creates a P-controller (P_{rate}) for the charge rate. Moreover, the `SelectChargeRate` function which selects a charge rate based on the capacitor voltage creates a P-controller (albeit stepped due to the zones) on the capacitor voltage ($P_{voltage}$). Together these control systems resemble cascaded P-controller, where the outer controller $P_{voltage}$ creates the setpoint for the inner controller P_{rate}.

If only a single P-controller was used for the voltage, it could happen that due to a slightly too large P value the controller would not step the connection rate down enough to keep the capacitor from discharging in every situation.

Due to the second P-controller for the charge rate, the system will always keep adjusting the connection rate in order to target a positive charge rate for the capacitor. In theory, this should provide superior robustness and response to non-linear gains.

Chapter 4

Evaluation

The evaluation of our system is split up in a *static* and a *dynamic* part. The static evaluation quantifies the connection setup performance for each optimization stage. The static behavior will be benchmarked on *number of packets during connection setup*, *connection setup time*, *connection parameter update time*, *time until first useful packet*, and *power consumption*. The dynamic evaluation is about quantifying the performance of FRAPPUCcInO, which will be based on *average throughput*, and *responsiveness*.

This separation is made because the static performance improvement will be applicable to both intermittently-powered and conventionally-powered devices. Reducing connection setup time and power consumption on their own are attractive propositions since it improves responsiveness and battery life. The FRAPPUCcInO algorithm is not the only possible application of *Fast Reconnect*. For example, Fast Reconnect could also be leveraged to quickly build a connection when ambient energy is insufficient for even the maximum connection interval, allowing a pseudo-connected state.

The demo system described in Section 2.5 was also used in all experiments. The central is an nRF52840DK development board [12], and the peripheral is the modified FreeBie platform [6]. The nRF52840-Dongle was used as a sniffer for Wireshark [13]. During the static testing, the Nordic Power Profiler Kit II (PPK-II, [11]) was used to power FreeBie and measure its power consumption. During the dynamic testing, the PPK-II was replaced with a 46.5μW solar cell from Panasonic [17], and a Saleae Logic Pro 8 was used to capture digital and analog traces [19]. Finally, an Ikea TRÅDFRI smart light was calibrated and controlled through `python` to simulate changing energy harvesting conditions.

The `CMakeLists` file of both applications have been retrofitted to enable easy switching between optimization stages during testing. To enable a specific stage, the `CMake` variables must be set according to Table 4.1.

4.1 Results

4.1.1 Static Evaluation

Between the number of packets sent between the `CONNECT_IND` packet and the first packet that does not belong to the connection setup, shown in Figure 4.1, a significant reduction can be seen. From Stage 0 (no modifications) to Stage 4 (all

Variable	Stage 1	2	3	4
CACHE_SERVICE_DISCOVERY		1	1	1
SKIP_RECONF			1	1
CACHE_LL_FEAT_EXCH				1
CACHE_LL_DL_EXCH				1

Table 4.1: `CMake` **variables to set for a given optimization stage. Set the variable to 1 if the cell contains a 1, otherwise do not set the variable.**

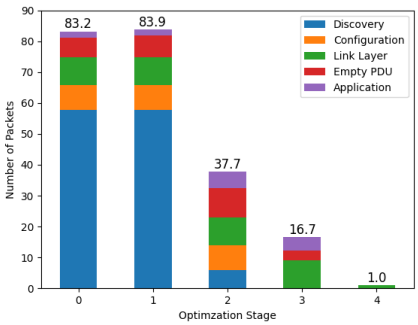

Figure 4.1: **Number of packets from the `CONNECT_IND` up to the last connection setup packet.**

modifications) this is a reduction of **98.80%**. When only sharing the preferred (slower) connection parameters (Stage 1), the average number of packets goes up by 0.7 packets. This increase in packets is a result of slightly more Empty PDUs as a result of different timing. Stage 2 still counts six packets for *discovery*, even though service discovery caching is being used because reading the Database Hash requires three packets for both the central and peripheral.

Figure 4.2 shows the time required to set up a connection, measured from `CONNECT_IND` up to the last packet that does not belong to the connection. At first, Stage 1 through 3 seems significantly worse than stage 0. However, it is important to remember that in Stage 0, the peripheral is forced to use the default connection parameters of the central. As a result, our system needs to be designed with a much larger capacitor than required after the connection is established. Although slower, Stage 1 allows us to reduce the size of the capacitors used in our system, which allows the system to recover much faster after a full power failure, as well as reducing the overall size and cost of the system. Stage 4 reduces the connection setup to a single packet, resulting in a reduction of **99.93%** compared to Stage 0.

To measure the fastest possible time until a useful packet is received by the central, we perform a *Read Request* from the central on the *battery level* characteristic. A *useful packet* is defined as anything that directly contributes to the function of an application. For example, a read or write request on a characteristic, or a write request on a CCC to enable Notifications.

From Figure 4.3 one can see that Stage 4 is about 4.46 times slower than

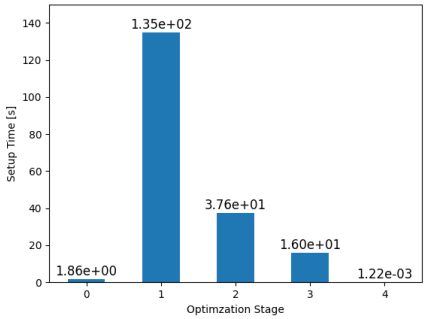

Figure 4.2: **Connection setup time, measured from `CONNECT_IND` up to the last packet that does not belong to the connection setup.**

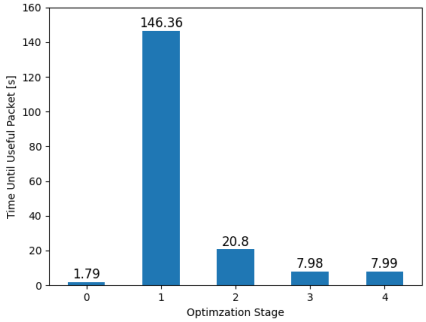

Figure 4.3: **Time until the first useful packet. Measured from `CONNECT_IND` up to the first useful application packet.**

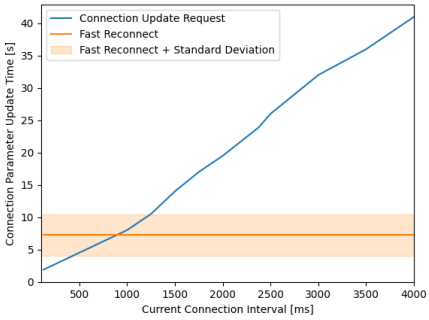

Figure 4.4: **Time until new connection parameters are applied versus currently used connection parameters.**

Stage 0. However, the default connection interval used by Zephyr in Stage 0 is 50 milliseconds, which is 80 times slower than the connection interval used by Stage 4. The improvements in Stages 2, 3, and 4 are able to recoup a significant amount of the time gained by using the slowest connection parameters, which is visible from Stage 1 in Figure 4.3. It is also notable that eight seconds is the lower limit that is possible at a connection interval of four seconds since a complete read operation from a GATT server requires two packets. The fact that there is no decrease from Stage 3 to Stage 4 is supported by the idea that the link layer procedures

Figure 4.4 shows the time it takes for new connection parameters to be applied when using the regular *connection update request* versus *Fast Reconnect*. Since the time required to reconnect using fast reconnect is independent of the connection interval, the time to update connection parameters does not increase as the connection interval increases. However, fast reconnect is more variable since it requires the central to receive a new advertisement from the peripheral. One standard deviation is shown in Figure 4.4 with light orange. An optimal approach would use the conventional connection update request when CI is below 1250, and fast reconnect for CI above 1250.

As one can see in Figure 4.5a, when conventionally powered and using slow connection parameters, the total energy consumed only drops below Stage 0 when using all optimizations. This is because the microcontroller is still using power while idling, so extending the setup time by increasing the connection interval will cause higher energy usage. However, when using all optimizations, the energy usage is **reduced by 99.46%**. When using these optimizations in devices that do not employ energy harvesting, it is advised to reduce the connection parameters after the setup process is done. This is shown in Figure 4.5b.

Figure 4.6 shows the energy used during the connection setup while intermittently-powered. Stage 3 and 4 show a significant reduction in energy consumed with **44.96%** and **93.97%**, respectively. However, Stage 1 and 2 might still be an improvement when intermittently-powered since the average current draw during connection setup is 68.96μA, and 71.13μA, compared to 1.21mA of Stage 0. This reduced continuous current draw might allow an intermittently-powered device to remain operational using Stage 1, while Stage 0 would result in the

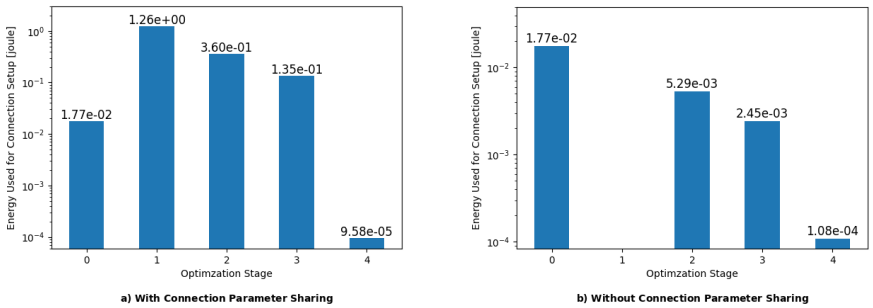

Figure 4.5: **Energy consumed (in Joule) during connection setup versus optimization stage when conventionally-powered. a) Uses Connection Parameter Sharing (Stage 1). b) Does not use Connection Parameter Sharing.**

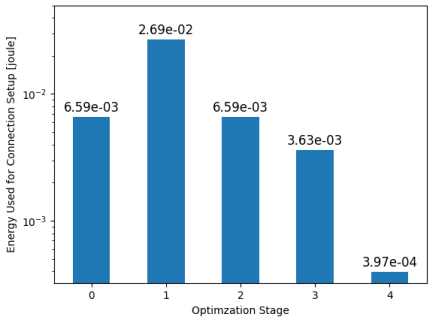

Figure 4.6: **Energy consumed (in Joule) during connection setup versus optimization stage when intermittently-powered.**

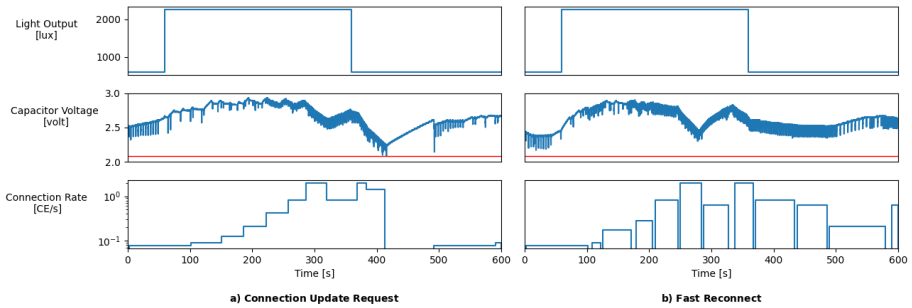

Figure 4.7: **Connection Rate versus Light Output. The average throughput when using Fast Reconnect (b) is 24.29% higher and the peak throughput comes 15.86% earlier when compared to using the regular Connection Update Request (a).**

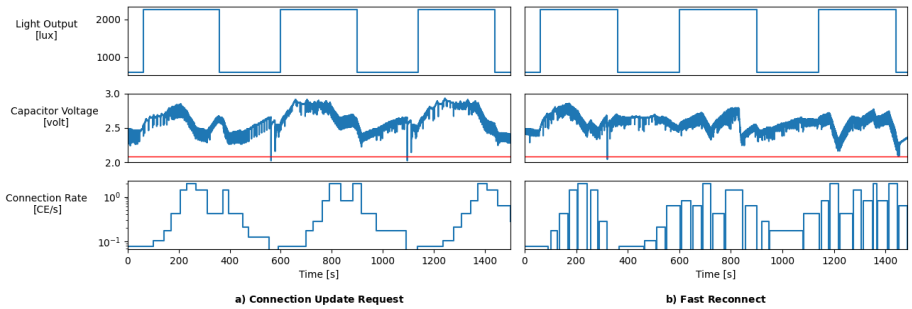

Figure 4.8: **Connection Rate versus Light Output. left) Connection Update Request. right) Fast Reconnect.**

capacitor voltage becoming critically low.

4.1.2 Dynamic Evaluation

Figure 4.8 shows an optimal situation for using FRAPPUCcIno, which is a sharp increase followed by a sharp decrease in available power after using slow connection parameters. In this scenario, using Fast Reconnect allows an earlier rise, as well as allowing the system to respond in time to prevent a power failure. As a result, FRAPPUCcInO using Fast Reconnect is able to maintain an average throughput of **24.29%** higher when compared to FRAPPUCcInO using the regular Connection Update Request. We define the responsiveness as the peak-to-peak time between the *light output* and the *connection rate*. Using this definition, FRAPPUCcInO with Fast reconnect reaches its peak 191 seconds after the light output rises, and FRAPPUCcInO using Connection Update Request reaches its peak after 227 seconds. The responsiveness is therefore improved by **15.86%**.

When performing a long run, the advantage in throughput is less drastic but still significant. Figure 4.8 shows a 25 minute run with multiple peaks in light output. In this scenario, the average throughput is **10.87%** higher for

FRAPPUCcInO using Fast Reconnect. However, responsiveness has improved by **107%**. This improvement can be attributed to the fact that FRAPPUCcInO with Fast Reconnect is able to decrease the throughput earlier to start charging the capacitor up. As a result, FRAPPUCcInO reaches the voltage where it can increase the throughput again much earlier. We should note that during runs longer than 15 minutes, both systems were very like to encounter hard faults. These instability issues should be fixed before we are able to fully assess the performance of FRAPPUCcInO over longer periods.

Chapter 5

Future Work

While the method of sharing connection parameters within the advertisement data works, it requires compliance from the central to work. Further research could focus on standardizing the procedures around sharing the connection parameters so that other platforms can support this feature and Bluetooth Low-Energy becomes more hospitable to batteryless devices.

Although Fast Reconnect covers all link-layer communication used in our demo system, further testing must be done to assess if our method can be applied to all link-layer exchanges. A generic approach built into the link layer controller would be ideal and would guarantee support for future revisions of BLE.

Finally, while FRAPPUCcInO provides some promising initial results, it is clear that further work is required to improve stability and optimize the algorithm further. A combination of using *Fast Reconnect* and the regular *connection parameter update request* as the methods of applying parameters could prove even more effective.

Chapter 6

Conclusions

We have presented a new method of reducing connection setup overhead and energy usage in Bluetooth Low-Energy (BLE) for battery-less devices, as well as conventional battery-powered devices. This method is called Fast Reconnect and is platform agnostic. Fast Reconnect applies caching in multiple levels of the BLE stack, from the GATT layer all the way down to the link layer, to reduce the connection setup process to a single packet. To improve connection parameter adaption time within FreeBie, we also presented FRAPPUCcInO, which combines Fast Reconnect with a control system to improve throughput and responsiveness under highly variable energy harvesting conditions. Using Fast Reconnect, together with FRAPPUCcInO, intermittent devices can become smaller, perform with even less ambiently available energy, and even enjoy a more responsive user experience.

Bibliography

[1] Arif, Rauf. With an economic potential of $11 trillion, internet of things is here to revolutionize global economy. https://www.forbes.com/sites/raufarif/2021/06/05/with-an-economic-potential-of-11-trillion-internet-of-things-is-here-to-revolutionize-global-economy/?sh=1f82f8015f29, 2021. Last accessed: August 31st, 2022.

[2] Bluetooth SIG. Core specification 5.3. https://www.bluetooth.com/specifications/specs/core-specification-5-3/, 2021. Last accessed: September 1st, 2022.

[3] Bluetooth SIG. Core specification supplement 5.3. https://www.bluetooth.com/specifications/specs/core-specification-supplement-10/, 2021. Last accessed: October 5th, 2022.

[4] Alexei Colin, Emily Ruppel, and Brandon Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. *SIGPLAN Not.*, 53(2):767–781, mar 2018.

[5] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(3), sep 2020.

[6] Jasper de Winkel, Haozhe Tang, and Przemysław Pawełczak. Intermittently-powered bluetooth that works. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 287–301, New York, NY, USA, 2022. Association for Computing Machinery.

[7] Shyamnath Gollakota, Matthew S. Reynolds, Joshua R. Smith, and David J. Wetherall. The emergence of rf-powered computing. *Computer*, 47(1):32–39, 2014.

[8] Josiah Hester and Jacob Sorber. The future of sensing is batteryless, intermittent, and awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Marr, Bernard. Smart dust is coming. are you ready? https://www.forbes.com/sites/bernardmarr/2018/09/16/smart-dust-is-coming-are-you-ready/?sh=39bf23d05e41, 2018. Last accessed: August 31st, 2022.

[10] Microsoft. 2019 manufacturing trends report. https://info.microsoft.com/rs/157-GQE-382/images/EN-US-CNTNT-Report-2019-Manufacturing-Trends.pdf, 2019. Last accessed: August 31st, 2022.

[11] Nordic Semiconductor. Nordic power profiler kit ii.

[12] Nordic Semiconductor. nrf52840 dk product page.

[13] Nordic Semiconductor. Nrf52840 dongle.

[14] Nordic Semiconductor. Softdevices.

[15] Nordic Semiconductor. Building a bluetooth application on nrf connect sdk - part 3 optimizing the connection. https://devzone.nordicsemi.com/guides/nrf-connect-sdk-guides/b/software/posts/building-a-bluetooth-application-on-nrf-connect-sdk-part-3-optimizing-the-connection, 2022. Last accessed: September 1st, 2022.

[16] Packetcraft, Inc. Packetcraft website.

[17] Panasonic. Panasonic am-1454ca datasheet.

[18] M. Philipose, J.R. Smith, B. Jiang, A. Mamishev, Sumit Roy, and K. Sundara-Rajan. Battery-free wireless identification and sensing. *IEEE Pervasive Computing*, 4(1):37–45, 2005.

[19] Saleae. Saleae logic pro 8 usb logic analyzer.

[20] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile '19, page 45–50, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Bluetooth SIG. 2022 market update. https://www.bluetooth.com/2022-market-update/.

[22] Bluetooth SIG. About us. https://www.bluetooth.com/about-us/.

[23] Bluetooth SIG. Origin of the name bluetooth. https://www.bluetooth.com/about-us/bluetooth-origin/.

[24] Texas Instruments. Ti documentation for the ble protocol stack.

[25] The Linux Foundation Projects. Zephyr project, Sep 2022.

[26] Kevin Townsend, Cufi Carles, Akiba, and Robert Davidson. *Getting started with Bluetooth Low Energy*. SPD, 2015.

Appendix A

Demo System

This appendix includes supplementary information on the features of the demo system, as well as a description of how it was implemented and developed.

A.1 Using the System

A.1.1 Connecting and Powering the System

Connecting and powering up the "development" configuration shown in Figure 2.11 is as simple as connecting both development boards to a computer using two micro USB to USB A cables. Be sure to check if all power switches are set to the ON position.

Connecting and powering the "testing" configuration shown in Figure 2.11 is a bit more involved. The central is connected to the computer using a micro USB-to-USB A cable like in the development setup. Connect FreeBie to a Segger JLink debug probe using a 5-pin debug cable. Attach the debug probe to the computer using a USB B-to-USB A. Power the FreeBie using the Nordic Power Profiler Kit II (PPK2), by connecting the the *Vout* and *Gnd* on the PPK2 to the *Vbat* and *Gnd* on FreeBie. Finally, download and open the Power Profiler application and change the settings to *Volt Meter* and *2660mV*.

A.1.2 Using the Terminal

Command	Help
`ble list show`	Show all discovered devices.
`ble list find <local name>`	Find device by name.
`ble connect <index>`	Connect to a device using its index in the device list.
`ble disconnect`	Terminate the current connection using the *Remote User Terminated Connection* reason.
`ble conn reconnect`	Terminate the current connection using the *Fast Reconnect* reason.
`ble cache delete <name>`	Delete the cached data and hash of an entry by name (e.g. "handles").
`ble cache scramble <name>`	Scramble the cached hash of an entry by name (e.g. "handles").
`ble cts notify`	Trigger a notification for the Current Time characteristic of the Central Time Service.
`ble test read`	Perform a large read request ($\geq$ 27 bytes) on the Test characteristic of the Test service. Used to validate *LE Data Length Extension* caching.
`ble test write`	Perform a large write request ($\geq$ 27 bytes) on the Test characteristic of the Test service. Used to validate *LE Data Length Extension* caching.

Table A.1: **Terminal commands available on the central.**

Command	Help
`disconnect <id> [--fast]`	Terminate a connection by id (usually 1) using the *Remote User Terminated Connection* reason (or *Fast Reconnect* when the "–fast" flag is added).
`param_update <preset index>`	Perform a parameter update to a predefined preset using *Fast Reconnect*.

Table A.2: **Terminal commands available on the peripheral**

When the development boards are connected to the laptop, virtual COM ports are created on `/dev/ttyACM0`, `/dev/ttyACM1`, etc. A serial terminal application, such as *PuTTY*, can be used to connect to the central and peripheral with a baud rate of 115200 bits per second. When connected, a `help` command can be entered to display the available commands.

Table A.1 and Table A.2 show commands that can be invoked on the central and peripheral, respectively. These commands are used to perform basic functions required for testing new code and specific procedures to validate caching

methods.

A.1.3 Making a Connection

To perform a connection setup, use the following steps:

1. Connect and power on the central and peripheral as described in Section A.1.1.

2. Open a terminal session to the central using the method described in Section A.1.2.

3. Enter `ble list show` and look for the local name "Watch" or enter `ble list find Watch` if the list exceeds the terminal size.

4. Enter `ble connect <index>`, where `index` is the index of the device entry found in the previous step.

A.1.4 GATT Services and Characteristics

Service	**Characteristic**	**UUID**	**Perm.**
GAP	Device Name	0x2A00	R
	Appearance	0x2A01	R
	Central Address Resolution	0x2AA6	R
	PPCP	0x2AC9	RW
GATT	Service Changed	0x2A05	R
	Service Changed CCC		RW
	Client Supported Features	0x2B29	RW
	Database Hash	0x2B2A	R
CTS	Current Time	0x2A2B	RWN
	Current Time CCC		RW
Test	Supported New Alert Category	0x2A47	R
	New Alert	0x2A46	N
	Supported Unread Alert Category	0x2A48	R
	Unread Alert Status	0x2A45	N
	Alert Notification Control Point	0x2A44	W

Table A.3: **Services that are available in the central application. Permissions have been abbreviated in the permissions (perm.) column to R for Read, W for Write, and N for Notification**

Service	Characteristic	UUID	Perm.
GAP	Device Name	0x2A00	R
	Appearance	0x2A01	R
	Central Address Resolution	0x2AA6	R
	Resolvable Private Address Only	0x2AC9	R
GATT	Service Changed	0x2A05	R
	Service Changed CCC		RW
	Client Supported Features	0x2B29	RW
	Database Hash	0x2B2A	R
	Server Supported Features	0x2B3A	R
Battery	Battery Level	0x2A19	R
	Battery Level CCC		RW
Test	Text		RW
	Text CCC		RW

Table A.4: **Services that are available in the peripheral application. Permissions have been abbreviated in the permissions (perm.) column to R for Read, W for Write, and N for Notificatio**

Table A.3 and Table A.4 show the services and characteristics that were added to the central and peripheral applications, respectively. If a specific use is mentioned in the table, then it was explicitly added to validate our modifications. The remaining services and characteristics provide a more realistic workload when testing service discovery.

A.2 System Implementation

A.2.1 Packetcraft

Registering GATT Services with the ATT server

In this example, it is shown how to add a simple GATT service with a single characteristic that contains the string "Lorem ipsum". First, define three variables for the service declaration, characteristic declaration, and the characteristic value.

```
1  /* Test service declaration */
2  static const uint8_t testValSvc[] = {TEST_UUID_SERVICE};
3  static const uint16_t testLenSvc = sizeof(testValSvc);
4
5  /* Test text characteristic */
6  static const uint8_t testValTxtCh[] = {ATT_PROP_READ |
       ATT_PROP_WRITE, UINT16_TO_BYTES(TEST_TXT_HDL), TEST_UUID_TEXT};
7  static const uint16_t testLenTxtCh = sizeof(testValTxtCh);
8
9  /* Test text value */
10 static uint8_t testValTxt[TEST_TXT_LEN] = { 'L','o','r','e','m',' '
       ,'i','p','s','u','m', 0 };
11 static const uint16_t testLenTxt = sizeof(testValTxt);
```

These variables can be used to describe the ATT attributes that make up the GATT service.

```
1  static const attsAttr_t testList[] =
```

```
{
    /* Service declaration */
    {
        attPrimSvcUuid,
        (uint8_t *) testValSvc,
        (uint16_t *) &testLenSvc,
        sizeof(testValSvc),
        0,
        ATTS_PERMIT_READ
    },
    /* Characteristic declaration */
    {
        attChUuid,
        (uint8_t *) testValTxtCh,
        (uint16_t *) &testLenTxtCh,
        sizeof(testValTxtCh),
        0,
        ATTS_PERMIT_READ
    },
    /* Characteristic value */
    {
        testTxtUuid,
        testValTxt,
        (uint16_t *) &testLenTxt,
        sizeof(testValTxt),
        (ATTS_SET_UUID_128
        | ATTS_SET_READ_CBACK
        | ATTS_SET_WRITE_CBACK),
        (ATTS_PERMIT_READ | ATTS_PERMIT_WRITE)
    },
};
```

Finally, the GATT service can be registered with the ATT server.

```
/* Test group structure */
static attsGroup_t svcTestGroup =
    {
        NULL,
        (attsAttr_t *) testList,
        NULL,
        NULL,
        TEST_START_HDL,
        TEST_END_HDL
    };

AttsAddGroup(&svcTestGroup);
```

Supporting Notifications for a Client

To allow a GATT client to enable Notifications on a characteristic using its CCC descriptor, we need to register the CCCs with the ATT server. To do this we first define the CCC set.

```
/*! enumeration of client characteristic configuration descriptors
    */
enum
{
    APP_BATT_LVL_CCC_IDX,   /*! Battery Level */
    APP_CCC_COUNT
};

```

```
/*! client characteristic configuration descriptors settings,
    indexed by above enumeration */
static const attsCccSet_t appCccSet[APP_NUM_CCC_IDX] =
{
    /* APP_BATT_LVL_CCC_IDX */
    {
        BATT_LVL_CH_CCC_HDL,    // ccc handle
        ATT_CLIENT_CFG_NOTIFY, // value range
        DM_SEC_LEVEL_NONE       // required security level
    }
};
```

The CCC set can then be registered with the ATT server.

```
static void appCccCback(attsCccEvt_t *pEvt)
{
    /* Perform any necessary action to enable or disable
    notifications, like starting periodic timers. */
}

AttsCccRegister(APP_CCC_COUNT, (attsCccSet_t *) appCccSet,
    appCccCback);
```

Service Discovery

To discover a single service, define a list of characteristics to discover. Afterward, perform a service discovery using the characteristic list and pass the function a pointer to the global handle list.

```
uint16_t globalHandleList[2] = {};

/* Characteristic 1 */
static const attcDiscChar_t ch1 =
{
    Characteristic1Uuid, /* uuid */
    ATTC_SET_UUID_128 /* optional characteristic settings */
};

/* Characteristic 1 */
static const attcDiscChar_t ch2 =
{
    Characteristic2Uuid,
    0
};

/*! List of characteristics to be discovered; order matches handle
    index enumeration  */
static const attcDiscChar_t *charList[] =
{
    &ch1,
    &ch2
};

AppDiscFindService(connId,
                   ATT_16_UUID_LEN,
                   (uint8_t *) ServiceUuid,
                   GATT_HDL_LIST_LEN,
                   (attcDiscChar_t **) charList,
                   globalHandleList);
```

Configuration of a Remote GATT Server

Configuration consists of initial reads from and writes to characteristics and enabling notifications for characteristics. However, since notifications are enabled by writing to the CCC descriptor of a characteristic, configuration is essentially only initial reads from and writes to characteristics. Packetcraft automates configuration by allowing us to define a list of characteristics to read from or write to automatically. For example, first define a list containing an initial read on the Central Time characteristic, as well as writing to the Central Time CCC descriptor to get updates of the Central Time in the future.

```
/* Default value for CCC notifications */
static const uint8_t watchCccNtfVal[] = {UINT16_TO_BYTES(
    ATT_CLIENT_CFG_NOTIFY)};

/* List of characteristics to configure after service discovery */
static const attcDiscCfg_t discoveryConfig[] =
{
  /* Read:  CTS Current time */
  {NULL, 0, (TIPC_CTS_CT_HDL_IDX + WATCH_DISC_CTS_START)},

  /* Write:  CTS Current time ccc descriptor */
  {watchCccNtfVal, sizeof(watchCccNtfVal), (TIPC_CTS_CT_CCC_HDL_IDX
     + WATCH_DISC_CTS_START)},
};

AppDiscConfigure(connId,
                 APP_DISC_CFG_START,
                 WATCH_DISC_SLAVE_CFG_LIST_LEN,
                 (attcDiscCfg_t *) discoveryConfig,
                 WATCH_DISC_SLAVE_HDL_LIST_LEN,
                 globalHandleList);
```

Custom Discovery and Configuration Procedure

This example shows how to register a callback to customize Packetcrafts default discovery and configuration behavior.

```
static void watchDiscCback(dmConnId_t connId, uint8_t status)
{
    switch(status)
    {
    case APP_DISC_INIT:
        /* perform custom initialization */
        /* For example: read the cache   */
        break;
    case APP_DISC_READ_DATABASE_HASH:
        /* Read peer's database hash */
        AppDiscReadDatabaseHash(connId);
        break;
    case APP_DISC_SEC_REQUIRED:
        /* request security */
        AppSlaveSecurityReq(connId);
        break;
    case APP_DISC_START:
        /* discovery first service */
        break;
    case APP_DISC_FAILED:
    case APP_DISC_CMPL:
        /* perform custom completion */
```

```
23          /* For example: write the cache    */
24          /* start configuration */
25          AppDiscConfigure(...);
26          break;
27      case APP_DISC_CFG_START:
28          /* start configuration in case */
29          /* discovery is skipped */
30          AppDiscConfigure(...);
31          break;
32      case APP_DISC_CFG_CONN_START:
33          /* Configuration for connection setup started */
34          break;
35      case APP_DISC_CFG_CMPL:
36          AppDiscComplete(connId, status);
37          break;
38      default: break;
39      }
40 }
41
42 AppDiscRegister(watchDiscCback);
```

A.2.2 Zephyr

Registering GATT Services with the ATT server

This example shows how to add a simple GATT service with a single characteristic containing the string "Lorem Ipsum".

```
1  static uint8_t text_value[] = { 'L','o','r','e','m',' ','i','p','s'
       ,'u','m', 0 };
2
3  /* Service Declaration */
4  BT_GATT_SERVICE_DEFINE(test_service,
5      BT_GATT_PRIMARY_SERVICE(TEST_SERVICE_UUID),
6      BT_GATT_CHARACTERISTIC(TEST_SERVICE_TEXT_UUID,
7                             BT_GATT_CHRC_READ | BT_GATT_CHRC_WRITE,
8                             BT_GATT_PERM_READ | BT_GATT_PERM_WRITE,
9                             NULL, NULL,
10                            text_value),
11 );
```

Supporting Notifications for Clients

The example in the previous section can easily be extended to support notifications.

```
1  static void ccc_cfg_changed(const struct bt_gatt_attr *attr,
       uint16_t value)
2  {
3      /* Perform any necessary action to enable or disable
       notifications, like starting periodic timers. */
4  }
5
6  /* Service Declaration */
7  BT_GATT_SERVICE_DEFINE(test_service,
8      BT_GATT_PRIMARY_SERVICE(TEST_SERVICE_UUID),
9      BT_GATT_CHARACTERISTIC(TEST_SERVICE_TEXT_UUID,
10                            BT_GATT_CHRC_READ | BT_GATT_CHRC_WRITE,
11                            BT_GATT_PERM_READ | BT_GATT_PERM_WRITE,
12                            NULL, NULL,
```

```
                                text_value),
    BT_GATT_CCC(ccc_cfg_changed, BT_GATT_PERM_READ |
    BT_GATT_PERM_WRITE),
);
```

Service Discovery

The discovery of GATT service in Zephyr is a very manual, cumbersome process. To make this easier we developed a library called *gatt_disc*. This library allows us to easily automate the discovery of any service. As an example, we automate the discovery of the GAP service.

```
#include "gatt_disc.h"

/* automatic index enumeration */
enum {
  GAP_DEVICE_NAME_IDX,
  GAP_APPEARANCE_IDX,
  GAP_CAR_IDX,
  GAP_LEN
};

gatt_disc_service_t gap_service = {
    .name = "GAP",
    .attrs_len = GAP_LEN,
    .attrs = {
        {
            .name = "Device Name",
            .type = BT_GATT_DISCOVER_CHARACTERISTIC,
            .idx = GAP_DEVICE_NAME_IDX,
        },
        {
            .name = "Appearance",
            .type = BT_GATT_DISCOVER_CHARACTERISTIC,
            .idx = GAP_APPEARANCE_IDX,
        },
        {
            .name = "Central Address Resolution",
            .type = BT_GATT_DISCOVER_CHARACTERISTIC,
            .idx = GAP_CAR_IDX,
        },
    }
};

void gap_discover_service(struct bt_conn *conn, uint16_t *handles,
    gatt_disc_done_t done_func) {
  gatt_disc_service_init(&gap_service, BT_UUID_TYPE_16, BT_UUID_GAP
    , (struct bt_uuid *[]){
      BT_UUID_GAP_DEVICE_NAME,
      BT_UUID_GAP_APPEARANCE,
      BT_UUID_CENTRAL_ADDR_RES
  }, GAP_LEN, handles);

  gatt_disc_service(conn, &gap_service, done_func);
}
```

We can then discover the service at any time using `gap_discover_service`.

```
uint16_t handles[GAP_LEN] = {};

void disc_done(struct bt_conn *conn, gatt_disc_service_t *service)
```

```
4 {
5     /* discovery is done */
6 }
7
8 gap_discover_service(conn, handles, disc_done);
```

Configuration of a Remote GATT Server

This example shows how we can enable notifications for a single characteristic.

```
1  struct bt_gatt_subscribe_params subscribe_params;
2
3  uint8_t notify_func(struct bt_conn *conn,
4                      struct bt_gatt_subscribe_params *params,
5                      const void *data, uint16_t length)
6  {
7      /* notification received */
8  }
9
10 subscribe_params.value_handle = CHARACTERISTIC_HANDLE;
11 subscribe_params.notify = notify_func;
12 subscribe_params.value = BT_GATT_CCC_NOTIFY;
13 subscribe_params.ccc_handle = CHARACTERISTIC_CCC_HANDLE;
14
15 int err = bt_gatt_subscribe(conn, &subscribe_params);
16 if (err && err != -EALREADY) {
17     /* subscription failed */
18 } else {
19     /* subscribed */
20 }
```