Bluetooth Mesh Interoperability Analysis

Fredrik Danebjer
dat12fda@student.lu.se
Casper Schreiter
ala10csc@student.lu.se

Department of Electrical and Information Technology
Lund University

Advisor: Jens Andersson, EIT

October 16, 2017

Printed in Sweden
E-huset, Lund, 2017

Popular Science Summary

Casper Schreiter & Fredrik Danebjer

The "Internet of Things" (IoT) is a continually growing area in the world of IT. As it expands, more and more previously non-computing devices are provided with a processor that makes it possible for them to implement various messaging protocols. These devices are often far more restricted in their computing capability in comparison with desktop computers or even smart phones. The share amount of new such devices, as well as the fact that they are less capable, puts more pressure on the underlying wireless communication technologies. One of these wireless technologies that is widely in use, in e.g. smart phones, stereos, thermostats, etc., is Bluetooth. Bluetooth creates small local personal area networks (PAN) with up to eight devices in a peer-to-peer manner. In this network one device is the master of the network and all communication is reliant on this device. Having all communication going through one device is not very suitable for networks that are rapidly growing in scope. Bluetooth Special Interest Group (Bluetooth SIG), the organization that maintains and develops the Bluetooth technology, realized this and developed a new way of utilizing Bluetooth, the Bluetooth Mesh protocol.

Bluetooth Mesh utilizes a mesh topology, instead of the former peer-to-peer communication. In a mesh network there are nodes capable of rebroadcasting messages not intended for themselves, this means that the range of the network can greatly be increased. The mesh topology also makes it possible for devices to communicate directly with each other without going through a master device first. Bluetooth Mesh is supposed to be able to be used by devices that implements Bluetooth Low Energy (BLE), a newer version of the classic Bluetooth technology that allows devices to minimize power usage. This new Bluetooth Mesh protocol is designed to work as a software update, meaning that no new additional hardware has to be put in place. With billions of different BLE devices already on the market, available from numerous amount of different manufacturers, Bluetooth Mesh could be a highly desirable networking technology.

For the same reason that Bluetooth Mesh is desired due to hardware availability, it would in the same manner be fruitful if a single software design could be applied across different platforms. Not only because less work would be required to apply this software update on all desired platforms, but also since developers does not need to gain expertise in a number of different SDKs.

The authors investigated this possibility of a single software design across three different so called System-on-a-Chip (SoC), a constrained hardware platform, with the purpose of minimizing power usage. These platforms all provided their own BLE protocol stack implementation, as well as their own software architectures to run on the processors. The authors argues that adopting a component based design

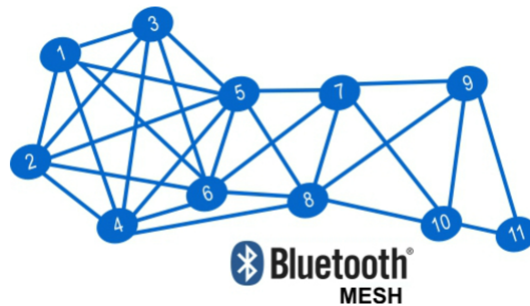

Fig. 1. Bluetooth Mesh

yields the greatest potential to achieve this interoperable solution. However, different SoCs indeed uses very different architectures, and the resources required to achieve the goal are not always in place. These resources are things such as real-time timers, and different solutions that allows computing to take place concurrently at the same time. With the differences that was found across the surveyed SoCs, it was shown that hoping to achieve complete software interoperability in the current state is not feasible. Instead, as a benefit for all embedded system software, time should be put into developing these required resources to enable future interoperable solutions.

As it was shown that it was possible to put this software update on top of all the different surveyed BLE implementations, the conclusion is that Bluetooth Mesh indeed seems to be able to be applied as a software update, as stated by Bluetooth SIG. The fact that so many devices already has BLE capacity, both in the form of hardware and software, makes Bluetooth Mesh a serious contender as the upcoming leading wireless IoT networking technology. Although wireless mesh technologies is no new phenomenon, technologies such as ZigBee has been on the market for quite some time, the potential reach of Bluetooth Mesh outshines all of its competitors as new expensive hardware does not need to be developed.

Abstract

The expansion of the Internet of Things into our day to day life brings with it an advancement in wireless technology in order to facilitate the growing market. Bluetooth Mesh is an attempt to bring Bluetooth units with Bluetooth Low Energy capabilities into this ever expanding network of connected devices, doing so with only a software update. With the enormous quantity of Bluetooth devices from a massive amount of different manufacturers the question arises whether or not it is possible to create a general solution which can be used on a number of different devices. This thesis explores this possibility and attempts to bring to light difficulties when designing software with the aim of interoperability, and more specifically what pitfalls Bluetooth Mesh comes with.

Three different System on a Chip from three different manufacturers where used for the implementation: nRF52832, BGM121, and CC2650 from Nordic Semiconductors, Silicon Labs, and Texas Instruments respectively. The result yielded that in order to truly create an interoperable solution certain core features, such as same Real-Time Operating System as well as plenty of memory capabilities are needed. When more complexity is put on the implementation it also brings more and more work in order to maintain the interoperable solution. The main conclusion is thus that it is not feasible to aim for interoperability when trying to bring Bluetooth Mesh to already functional Bluetooth devices.

Acknowledgements

We would like to thank Jens Andersson for continuously giving us valuable input along the thesis lifetime and help us tie it all together in order for us to successfully get our Masters degree. The thesis was carried out at Sigma Connectivity, where we not only got to experiment with the different hardwares but also got a lot of very valuable and crucial guidance from many of the employees. Among these we would like to extend a special gratitude to our supervisor Xu Xiaodong for his invaluable support and input. Lastly, we would also like to thank Maria Kihl as the examiner of this thesis.

Casper Schreiter & Fredrik Danebjer

Table of Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Project Scope . . . 2
1.3 Methodology . . . 2
1.4 Related Work . . . 4
1.5 Disposition . . . 5

2 Theory **7**
2.1 Bluetooth . . . 7
2.2 Bluetooth Low Energy . . . 8
2.3 Bluetooth Mesh . . . 12

3 Implementation Considerations & Requirements **23**
3.1 Mesh Nodes . . . 23
3.2 Compilers . . . 25
3.3 Wrapping and General Hardware Limitations . . . 26

4 Hardware **27**
4.1 Nordic Semiconductor nRF52832 . . . 27
4.2 Silicon Labs BGM121 . . . 28
4.3 Texas Instruments CC2650 . . . 29
4.4 Role Creation . . . 31

5 Modeling **35**
5.1 Design . . . 35
5.2 Proofs of Concepts . . . 43

6 Results & Evaluation **47**
6.1 Use-cases . . . 47
6.2 Component Implementation . . . 48
6.3 Abstraction Layer . . . 51
6.4 Memory Analysis . . . 53
6.5 RTOS . . . 55
6.6 Provisioning . . . 56

6.7 Scalability 56

7 Conclusions **57**
7.1 Discussion 57
7.2 Future Work 58

References **61**

A SDK Signatures **63**

List of Figures

2.1 BLE Protocol Stack . . . 9
2.2 Advertisment Interval . . . 11
2.3 Bluetooth Mesh Protocol Stack . . . 12
2.4 Friendship Procedure . . . 16
2.5 Example of a Bluetooth Mesh Network Topology . . . 17
2.6 Provisioning Protocol Stack . . . 20

3.1 Demo Network . . . 23

4.1 nRF52832 SoC . . . 27
4.2 BGM121 SoC . . . 28
4.3 CC2650 SoC . . . 29

5.1 Design Suggestion . . . 36
5.2 Example of a Bluetooth Mesh Network Topology . . . 40
5.3 LE Node state machine . . . 41
5.4 Friend Node state machine . . . 42
5.5 Implementation Proof of Concept Network . . . 44
5.6 Interoperable Proof of Concept Network . . . 44

6.1 Memory Usage in nRF52832 SoC . . . 54
6.2 Memory Usage in BGM121 SoC . . . 54
6.3 Memory Usage in CC2650 SoC . . . 54

List of Tables

3.1 Mesh Feature Requirements 24

4.1 SDK Features 30
4.2 SDK Mesh Feature Functionality 31
4.3 Platform Designs 32

6.1 Scanning and Advertising functionality across the SDKs 50
6.2 Mesh Implementation Flash Memory Usage 53
6.3 Mesh Implementation SRAM Usage 55

List of Acronyms & Abbreviations

ATT - Attribute Protocol
Bluetooth SIG - Bluetooth Special Interest Group
BLE - Bluetooth Low Energy
BR - Basic Rate
GAP - Generic Access Profile
GATT - Generic Attribute Profile
HFP - Hands-Free Profile
iCall - Indirect Call
IoT - Internet of Things
IVI - Initialization Vector Index
LE - Low Energy
LPN - Low Power Node
OS - Operating System
PAN - Personal Area Network
PDU - Protocol Data Unit
RSSI - Received Signal Strength Indication
RTOS - Real-Time Operating System
SDK - Software Development Kit
SDP - Service Discovery Protocol
SoC - System on a Chip
TI - Texas Instruments
TTL - Time To Live

Chapter 1
Introduction

This thesis explores the new Bluetooth Mesh protocol that was released during the process of this thesis. Bluetooth Mesh works as a software update to earlier Bluetooth Specifications and is supposed to be compatible with all implementations of Bluetooth v.4.0 and later. Focus will be put on implementing the protocol as a software update, and in more detail what issues may arise when applying this protocol to several different System-on-a-Chip (SoC) and Software Development Kits (SDK). These interoperability issues is explored in the scope of this thesis.

1.1 Background

The Internet of Things (IoT) aims to bring us into a more connected world, where we can control lights with our cellphone or let a stethoscope call to inform the doctor if we are sick. The "Things" in the "Internet of Things" are referring to traditionally non-computing devices that nowadays almost always has some sort of processor built into them. As time goes by, more and more products gain processing capability and thus also the ability to communicate with each other. It is projected that by 2020 almost 26 billion of these "Things" will exist on the market [1]. With this projection it is clear that a huge market exist for how to bring these devices together in a connected world and different technologies compete to offer the best solution for this revolution in connectivity.

Bluetooth is a technology that emerged as the use of cellphones started to increase. It was created at Ericsson in 1994 with the main focus to enable quick and easy data transfer between devices. It operates in a so called Personal Area Network (PAN) which consist of at least two devices in a master/slave relation. The communication is peer-to-peer with a master device that can handle up to seven slave devices. Bluetooth is one of the most popular choices to transmit data between devices, in 2014 90% of all cellphones had Bluetooth capabilities [2].

Since Bluetooth focuses on peer-to-peer communication it suffers by being quite short-ranged. A mesh network overcomes this obstacle by creating a rebroadcasting network that floods all messages until the recipient has been reached. This is a very popular technology used by e.g. ZigBee with great success. One disadvantage of ZigBee is that if one would like to build such a mesh network, every device needs

to be extended with a ZigBee chip, which could become both tedious and costly. Since Bluetooth already is in place in most cellphones, and other devices such as speakers, headphones, senors, etc., utilizing this technology for mesh might offer a speedy and reliant solution to the expansion of the IoT.

In the summer of 2017 Bluetooth Mesh was released, and with a single software update most modern Bluetooth devices can be upgraded to participate in a Bluetooth Mesh network in various ways. This thesis intends to identify what issues might arise from this software update, especially so when trying to apply it to several different SoCs.

1.2 Project Scope

Bluetooth and Bluetooth Low Energy (BLE) is already in use as a means of data transfer for the IoT. Using its mesh protocol could be considered cheap since the hardware technology already is in place in a lot of devices. Its specification states that a software update should be enough to enable a Bluetooth v.4.0 device into a Bluetooth Mesh device. Since the amount of Bluetooth devices is as great as it is a question whether it would be possible to create a somewhat generic solution arises.

This thesis was performed at such a time when Bluetooth Mesh v.1.0 was not yet released, meaning that other implementations of the protocol was hard to come by. The scope of this thesis does not include analysis on how well the protocol performs or how interoperable it is in regard to other implementations. Instead focus was put into how interoperable one solution is between a number of different SoCs and SDKs and thus see to what extent a total reimplementation could be avoided.

It should also be mentioned that we by no means claim to implement a fully functioning Bluetooth Mesh network. A complete network relies on numerous devices and roles, and to implement it all would be far too time consuming. Focus was put on finding possible pitfalls and discuss how to deal with these which, hopefully, shines a light on the core problems in regards to interoperability of the protocol.

1.3 Methodology

An extensive literature study was made in order to get initial knowledge about IoT, Mesh-topology solutions, and design of interoperable software in embedded systems. This included both academic work, technical specifications, and other sources related to the project scope area. This was complemented with an in-depth study of the Bluetooth Mesh v.0.9 specification.

Different SoCs was surveyed with regard to SDK and hardware specification, where SDK architecture and design was analyzed to form a common design structure.

A first implementation was made on one of the chips, using its SDK as little as possible. The design was made so that SDK dependencies would be minimized and isolated for effortless porting. This solution was then ported to other SoCs, all with a different SDK. The porting was analyzed in terms of interoperability, portability, and ease of implementation.

Finally, as a proof of concept of the protocols interoperability, a demo was created using all SoCs with a ported solution.

The evaluation will be made based on different criteria, namely interoperability, portability, and scalability, they are described in sections 1.3.1, 1.3.2, and 1.3.3.

1.3.1 Interoperability

Interoperability is a very ambiguous term in computer science, one definition could refer to two or more technologies working together on a single SoC. An example of this would be ZigBee and BLE, both of which operates on the same bandwidth and thus have the capability to distort each others signals. If they seamlessly are able to overcome this issue they are said to be interoperable. At the same time interoperability could refer to two devices using e.g. the TCP protocol and wishing to establish a direct link, thus requiring them to have to have physical layers that can understand each other. If they do not they will not be interoperable. A third case could be the capability of software written in different programming languages to interact with each other.

In this thesis interoperability is regarded as the property of an implemented solution to work seamlessly on different platforms. This property is closely related to, but not the same as, portability (see section 1.3.2). The question that are of interest to answer when measuring this interoperability is whether only porting is required for an implemented solution to work on a new SoC/SDK pair. If the answer is no and there are issues in the design of the software, then to what extent redesign is needed could indicate how non-interoperable the solution is.

1.3.2 Portability

If some software developed and executed on hardware A with some degree of work can be transferred and executed on hardware B, then it is said to be portable. The amount of work required to complete this process is a measurement of how portable the software is. If some software is interoperable between different hardware then by extension it also must be portable. The reverse is not necessarily true.

If some software is not portable between different hardware platforms, then they cannot be interoperable.

Since we focus on software and design interoperability, porting can be seen as a sub-issue that that guarantees compiled software behavior to be the same after the porting process as before. The amount of work required to guarantee this

behavior is a measurement that together with interoperability measurements can be used to make an assessment of feasibility in a generic solution.

1.3.3 Scalability

Scalability is the attribute of growing with ease. When an implementation/software update takes into account more and more platforms will it be feasible to continue on that update or is it more sensible to split it due to too large platform discrepancies? If it can be foreseen that such a split is the only reasonable course of action within a few software updates, then the value of a solution must be weighed and analyzed together with the amount of work required to port and to achieve interoperability.

1.4 Related Work

While there have not been much work done in regard to the interoperability of the Bluetooth Mesh protocol across different SoCs and SDKs, since the protocol as of writing has just been released, there have been work done in overlapping areas. The issues with interoperable code across different SoCs is well known, both in regards to hardware access and SDK differences.

The importance of portable software was pointed out by Sedov et al.[3], and rest to great extent on the fact that algorithms usually only needs to be implemented and tested once, while hardware usually gets improved and needs to be upgraded every few years. To be able to easily port algorithms and systems is thus highly profitable as time to re-implement already working code can be greatly reduced. They suggested a developing and maintenance process to achieve this goal, the focus lies however on a single company to do this for its own environment.

Medvidovic et al.[4] had another solution, they proposed that instead of enforcing an interoperable standard that all developers should adopt and/or implement, focus should instead lie on developing new software and SDKs to be as modular as possible, by adopting a component-programing style. This should aim to bridge the interoperability impossibilities. They also underlined how important it is that students in an early stage learn different design architectures so to be able to implement this style and understand the different systems the components need to be able to be attached to.

With the aim of interoperability and easy porting across different platforms Obiltschnig[5] described the different tools that are essential when developing software for embedded systems. He mentioned several of them as crucial for porting systems, and pointed out differences and similarities that must be considered for a modular system.

Finally Shodhganga[6] pointed out the importance of code size, as it now consumes the absolute largest physical area of modern Embedded Systems. His re-

sults showed that this needs to be kept in mind while developing, as a modular system easily can grow out of control when trying to appease a large amount of different architectures.

1.5 Disposition

The rest of this thesis is organized as follows:

- **Chapter 2** presents the theory of the different Bluetooth protocols required for a Bluetooth Mesh implementation.
- **Chapter 3** points out the different requirements needed in SDKs to implement the various parts of a Bluetooth Mesh network. It also points out other considerations.
- **Chapter 4** introduces the different SoCs with their corresponding SDK. It also discusses differences between them in regard to the Bluetooth Mesh protocol.
- **Chapter 5** is dedicated to designing an as interoperable solution as possible, this with regard to limitations and architectures discussed in earlier chapters.
- **Chapter 6** presents the results of the implementation and makes an analysis of these in regard to interoperability, portability and scalability.
- **Chapter 7** holds the conclusion, as well as a discussion about future work.

Chapter 2

Theory

In order to understand the need for and implementation of Bluetooth Mesh a closer look needs to be taken on the underlaying technologies. This chapter will contain a brief description of classic Bluetooth protocol, its more recent extension Bluetooth Low Energy, as well as an introduction to the Bluetooth Mesh technology itself.

2.1 Bluetooth

Bluetooth is a short-range wireless technology that aims at bringing reliable communication with a low cost and low power consumption. There are two forms of Bluetooth wireless technology systems; Basic Rate (BR), which the classic versions are based upon and Low Energy (LE), which was introduced in the Bluetooth Core Specification 4.0. The LE version will be discussed further in section 2.2. Devices can implement one or both of these subsystems, but devices that only implement one of them can not communicate with devices that only implement the other.[7]

Bluetooth has been around for some time now. As stated in the introduction it was developed in the mid to late 1990's at Ericsson. It is maintained and updated by the Bluetooth Special Interest Group (Bluetooth SIG), which is an organization made up by over 30,000 member companies [2]. Not all member companies have any input in how the development of Bluetooth is going to progress though, some just have the privilege to get the specifications of upcoming improvements/changes before they are formally finalized. Some of the companies that leads the Bluetooth SIG is Ericsson, Apple, and Microsoft. Almost all of the information relayed in this section is coming from the Bluetooth Core Specification 4.2 [8].

2.1.1 Piconet

Devices that connect with one another via Bluetooth form a network topology that is called piconet. It is a type of ad hoc network, which in itself is what a network is called if there are no pre-existing communication path. In other words it is a network that is created on the fly, as the term "ad hoc" (for this) implies, and does not have routers or network cords for connection. The piconet operates in a master/slave relation where it is the master that initiates the connection and

controls the timing of the network. The simplest piconet is a peer-to-peer connection between two devices, for instance between a smartphone, master, and a headset, slave. A master device can have up to seven slave devices connected at the same time in a peer-to-multipoint piconet. In addition to the seven active slaves a master can have 255 more parked slave devices. The master can then decide when to activate a parked slave device and thus also change a state of an active device to parked, assuming that seven slave devices already is in the active connected state.

Piconets can be combined with other piconets to form a scatternet. However, since the timing of each piconet is determined by the master's clock it is impossible for a device to be the master of two separate piconets simultaneously. Individual piconets of a scatternet is therefore connected through having slaves in common or by having a device acting as a master of one piconet simultaneously as being a slave in another.

2.1.2 Profiles

Bluetooth uses a concept of profiles in order to generalize behavior. Profiles defines what kind of communication that is supported; if two devices are going to be able to communicate with each other they have to support the same profiles. Each profile implements a customized protocol stack that is specified for its particular use-case. For instance the Hands-Free Profile (HFP) is customized in a way that allows audio to be transported to earpieces in order to allow car drivers to talk on the phone without holding the phone. This profile would be totally useless for a thermometer which only purpose is to send temperature data, hence a Bluetooth thermometer would not support the HFP.

There is one fundamental profile that all Bluetooth devices must support and that is the Generic Access Profile (GAP). GAP defines all the basic requirements for a device. It also describes for example the device discovery, connection establishment, and security of a device.

2.2 Bluetooth Low Energy

BLE builds on the classic Bluetooth technology, but is designed to minimize energy usage and subsequently have a lower data-rate transfer. It emerged in 2010 in the Bluetooth Core Specification Version 4.0. BLE operates in a different way than previous iterations of Bluetooth and is not compatible with the older versions.

2.2.1 BLE Protocol Stack

The BLE Protocol Stack (see Figure 2.1) specifies both connectible and broadcasting communication for devices that either are a low energy device or wish to communicate with one. As was mentioned in section 2.1.2 GAP is still an obligatory profile. BLE devices also has to implement the mandatory Generic Attribute

Profile (GATT) as well as the Attribute Protocol (ATT), which GATT is built on top of.

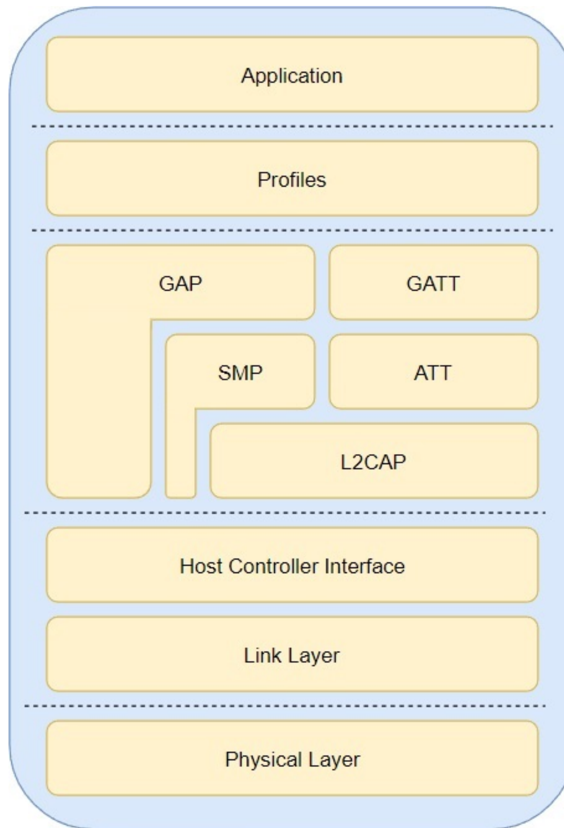

Figure 2.1: BLE Protocol Stack

At the bottom of the stack lies the Physical Radio, which operates on 2.4 GHz.

Generic Access Profile

As in the older versions of Bluetooth the connections and other basic requirements of a BLE device is handled by GAP. How GAP works is quite different from previous iterations of Bluetooth though, as should come as no surprise since optimization of the communication is of course heavily connected to the Bluetooth device energy consumption. The broadcasting, i.e. connectionless communication is entirely handled by GAP.

GAP in BLE devices defines four different profile-roles that operates over a LE physical transport: central, peripheral, broadcaster, and observer. Two of these roles relies on establishing a connection in order to transfer data:

- **Central** is a device that initiates a connection. It must have a transmitter and a receiver.
- **Peripheral** is a device that accepts incoming connections. It must have a transmitter and a receiver.

The other two roles relies on connectionless communication that uses advertisements (see the Advertisement subsection in section 2.2.1) for communication:

- **Broadcaster** is a device that sends advertisements. It must have a transmitter and may have a receiver.
- **Observer** is a device that receives advertisements. It must have a receiver and may have a transmitter.

Several of these roles could be implemented in one device at the same time.

For connectible devices, typically but not necessarily, a peripheral would be a low power device and the central would be a high energy device. This could e.g. be a light bulb controller as a peripheral and a smart phone as a central. The phone would then request a connection with the peripheral which in turn accepts or rejects this connection. Once the connection is established the phone would be able to toggle the light on and off.

For devices with connectionless communication both of the roles could commonly be implemented in a Low Power Node (LPN). A low energy device could e.g. serve as a thermometer in the form of a broadcaster, which simply advertises its current temperature. An observer could then quickly read the current temperature simply by listening for the advertisement.

An observer could be implemented to record all visible devices in its proximity during a long period of time, along with their advertised data. If the device also would have the peripheral role implemented it could then be connected to a central device which in turn could extract the gathered data.

Service Discovery Protocol

The Service Discovery Protocol (SDP) defines how to identify devices and services in the BLE protocol, and what services are supported by each other. In case of the light bulb and the smart phone this protocol's definitions are what allows the devices to communicate compatibility between the applications.

Advertisement

Advertisement is an important part of BLE. It is a method of broadcasting out small amounts of data, sent by the link layer. Usually it consists of identifiers defined by the SDP so that e.g. a central device can identify peripherals in its proximity, it can however also consist of purely user defined data.

Advertisement data are sent over GAP and is only 31 bytes, it can be configured to be sent with various intervals and timeouts. 31 bytes can be enough if the peripheral device, for example, is a thermostat that has very little data to share. If the advertiser have both a transmitter and a receiver it may extend the advertisement with a Scan Response, which is a secondary advertisement sent only when requested, see Figure 2.2.

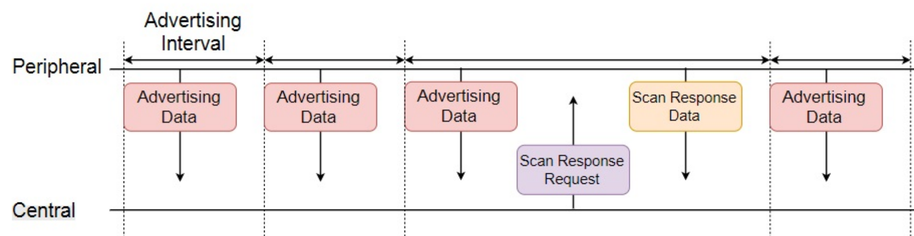

Figure 2.2: Advertisment Interval

Communication with such a limited amount of bytes per message can however be quite problematic if the devices are more complex. In order to handle this situation GAP can be used in order to establish a connection via GATT instead.

Generic Attribute Profile

GATT contributes a way for BLE devices to communicate directly with one another, it is the part of the stack that is used to establish a connection between two devices. Once this connection is established a continuous stream of data can be transfered between the two devices, as opposed to the advertisements which is much more limited in size.

A peripheral device can only have one GATT connection to a central device.

2.2.2 Applications

BLE can be used for many of the applications suited for classic Bluetooth. However, because of the lower data-rate transfers in BLE it is not suited for things such as voice/audio transfers in e.g. headsets, which is common for classic Bluetooth. In addition it is developed to allow new things for it to be featured in, such as near-proximity monitors, wearables, and other things to connect to the IoT.

A BLE node is a device that slumbers most of the time and thus uses less energy. It connects to a master, or gateway, which in turn potentially could connect to the "outside world".

2.3 Bluetooth Mesh

Bluetooth Mesh is a new Bluetooth technology that will work on all Bluetooth devices which implements the 4.0 core specification or higher, and can thus be applied in the form of a software update. It provides mesh-topology, allowing the Bluetooth units to bypass the piconet and scatternet topologies.

In the state as of writing, that is as of v.0.9 [9], Bluetooth Mesh will be released as a flooding based mesh topology. Plans already exist for a routing based topology of the protocol which should be included in an upcoming version of the specification.

2.3.1 Protocol stack

The Bluetooth Mesh Protocol Stack consists of eight layers in total, as can be seen in Figure 2.3. Out of these the upper six are completely new functionality that forms the Bluetooth Mesh core. The Bearer layer ties the mesh functionality together with the Bluetooth Low Energy Core Specification.

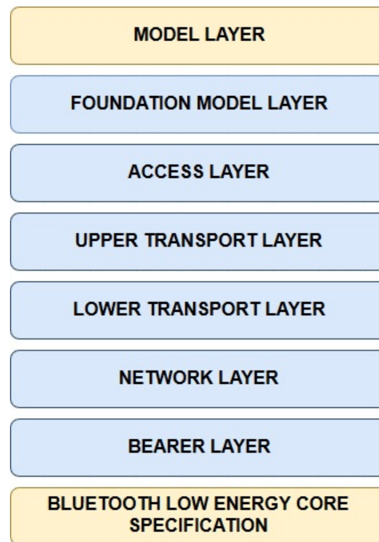

Figure 2.3: Bluetooth Mesh Protocol Stack

Bluetooth Low Energy Core Specification

This layer holds the BLE Core specification, i.e. how BLE is specified to communicate at the lowest levels. Bluetooth Mesh is primarily supposed to run on top of the BLE Core Specification with connectionless communication in the form of a GAP multi-role of observer-broadcaster. It can also be run as GATT profiles, but then utilizes this in a connectionless manner.

Bearer Layer

Bearers defines how network messages are transported between nodes. At the time of writing there are two bearers defined, the advertising bearer and the GATT bearer, which are explained in 2.2.1. The latter is used for backwards compatibility and legacy, this since in the BLE protocol there is no need for pure central device to implement an advertising bearer as it does not need to advertise its existence, only to discover others.

The advertising bearer is supposed to be the main bearer, and it is implemented by simultaneously implementing the observer and broadcaster roles, only sending and receiving user defined data.

Connections are established through a handshake with messages defined in the Network Layer. Once a connection has been established a stream of data can be sent over the bearers. This allows streams of data to be sent over advertising communication, which was not possible in the BLE protocol.

Network Layer

The Network Layer defines how messages are addressed and identified between nodes in the network. It is at this layer that it is decided if messages are to be rejected, relayed, or accepted for processing.

Lower Transport Layer

The Lower Transport Layer defines how messages from the Upper Transport Layer are segmented into Lower Transport Protocol Data Unit (PDU) and reassembled. This enables messages that are longer than 31 bytes to be sent over the Advertising Bearer in several smaller packages and later reassembled.

Upper Transport Layer

The Upper Transport Layer encrypts, decrypts, and authenticates the application data. This layer also defines the Transport Control Messages, which are used to manage the Upper Transport Layer between nodes. This includes functionality such as forming and clearing friendships, setting up subscription lists, and much more.

Access Layer

This layer defines how higher layer application accesses the Upper Transport Layer.

Foundation Model Layer

The Foundation Model Layer defines states, messages and models that are required to configure and manage the mesh network.

Model Layer

The Model Layer defines models (section 2.3.3) which are used to standardize operations for users. Bluetooth SIG defines some standard models in Bluetooth Mesh Model Specification [10], for example the `GenericOnOffClient` Model that can send `OnOff` messages, and `GenericOnOffServer` Model which can receive messages from the client model.

2.3.2 Mesh/Overview

As previously mentioned the mesh topology uses a technique called flooding. Flooding means that when a node receives data which itself is not the addressee of it will relay this data to all of its adjacent nodes. This processes is repeated for all nodes that receives the message (thus flooding the network) except when it reaches the addressee.

In order to not permanently flood the network, the network PDU has a Time To Live (TTL) flag which specifies how many relay-jumps the message at most may be relayed before being discarded. However, the TTL flag does not prevent messages to bounce back and forth between two network nodes. To avoid this the nodes implement a network cache where it can see if a newly received message already has been processed or not. If the message already has been processed it is simply discarded.

Bluetooth Mesh Subnets

A Bluetooth Mesh network is defined as nodes that shares four common resources:

- Network addresses (to identify source and destination of messages)
- Network keys (for securing and authenticating messages at network layer)
- Application keys (for securing and authenticating messages at upper transport layer)
- Initialization Vector Index (IVI)

Within a Network there may exist one or more subnets; such a subnet is a group of nodes that all share the same network key.

The IVI is a tag that says for how long the network should be kept alive.

These resources are allocated by a node called the provisioner. It is the provisioner that generates and distributes the keys, as well as manages the allocation of addresses such that they are kept unicast.

When a Bluetooth mesh enabled node wishes to join a network it advertises its existence to the provisioner, which in turn invites the node to the network if it can be authenticated. First after joining the network it can send and receive messages. A device might be part of more than one network.

Bluetooth Mesh Nodes

All nodes in a mesh network must have the ability to both transmit and receive messages, meaning they must all implement both the Broadcasting and Observer roles defined in 2.2.1. Aside from this ability mesh nodes can have specialized functionalities that are called features. In specification version analyzed in this thesis there are three different features:

- Relay
- Low Power
- Friend

A node might support one or more of these features, and with the exception of the Low Power feature these features may be enabled or disabled. Should a feature be enabled it might be in use or not.

The relay feature means that the node has the ability to upon receiving a message re-transmit it to its neighbors.

The Low Power feature is a feature belonging to a node that tries to minimize its power consumption. It achieves this by going to sleep and then only wakes up in cycles to receive messages. In order to achieve this it must first form a friendship with a neighboring node which has the Friend feature, and has it enabled.

The Friend feature is what allows the LPNs to participate in the mesh network. It allows a node to become a Friend with another node. When a LPN befriends another node it becomes dependent upon it. The befriended node then works as a cache for the LPN and stores messages for it. These messages are then sent to the LPN when it wakes up during its so called receive cycle. At that time the LPN sends a poll message to its Friend, which in turn sends the messages to the LPN.

Friendship Procedure

The Friendship Procedure can be explained by Figure 2.4. It consists of first forming a friendship with a node with an enabled Friend feature, through a handshake of a few steps. When friendship is established the procedure continues with a poll-send phase to deliver messages to the LPN.

When the LPN notices that it has no Friend it starts to broadcast a Friend Request message and then immediately enters sleep mode. After a little while it wakes up again and starts to scan for responses. Should it not receive any response in this scanning interval it simply tries again and keeps doing this until a response have been found. If it receives one or several Friend Offer messages it deems sufficiently good it then choses one of them to respond to. The criteria for choosing the best friend is developer specific, since different LPNs might have different needs. Criterions to be used could include proximity and/or available cache size in the Friend.

If we again refer Figure 2.4 we can see that a Friend node have sent a Friend Offer message as a response to the LPNs Friend Request message. The Friend starts a timer of one second at the same time it sends this offer, and if it does not receive a Friend Poll message from the corresponding LPN before this timer timeouts it should consider the friendship establishment failed. However, if it does receive the Friend Poll message the Friend instead sends a Friend Update message and considers the friendship established.

After the LPN has sent the Friend Poll message it waits for the Friend Update message, if no such message is received within its next scanning cycle it tries again, and keeps doing this up to five times. If no Friend Update message has been received by then it considers the friendship establishment failed.

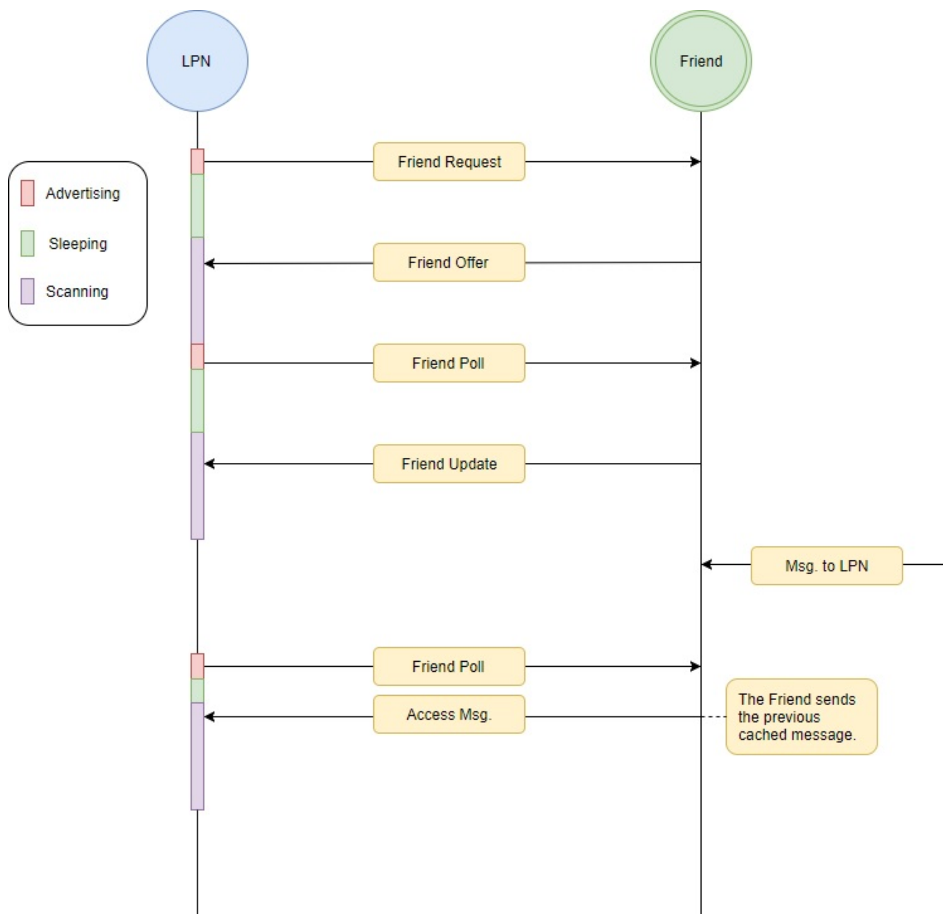

Figure 2.4: Friendship Procedure

When the friendship is established the procedure continues with the LPN waking up during its now low rate duty cycles and proceeds with sending a Friend

Poll message to the Friend. When the Friend receives the Friend Poll message it responds by sending messages from its cache linked to that LPN. If the cache is empty a new Friend Update message is generated, put into the cache and sent.

Should the LPN not receive a response in this Poll-Answer interval it tries again, and keeps doing this for up to five times. If no answer have been received by then the LPN considers the friendship terminated and starts looking for a new Friend.

Topology

An example of a Bluetooth Mesh Network can be seen in Figure 2.5. In this example we can imagine that node H wish to send a message to node G, it would proceed in the following manner:

- H sends his message with destination G to all his neighbors, i.e. to node E.
- E is not G, but since E have the relay feature it relays the message to all its neighbors, i.e. to B, C and F.
  - B is not G, and does not have G as a Friend, nor does it have the relay feature, so it discards the message.
  - C is not G, and does not have the Friend feature or the relay feature, so it discards the message.
  - F is not G, but it does have G as a Friend, so it caches the message for G.
- Some time later G wakes up and requests updates from F by sending a poll message.
- F receives the poll message from G and in response sends the cached message from H to G.

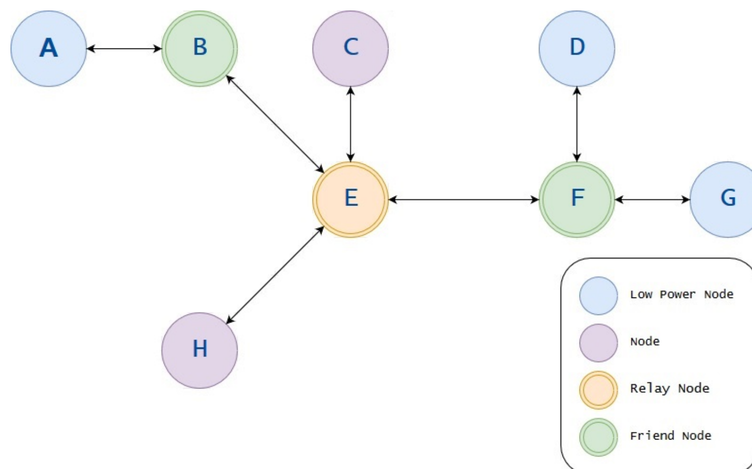

Figure 2.5: Example of a Bluetooth Mesh Network Topology

2.3.3 Architecture

The architecture of the Bluetooth Mesh protocol is centered around elements. Elements are the addressable units within a node, in other words it is the thing that the communication is directed at. Each mesh node needs to have at least one element, the primary element, but may have more. The number of elements and how they are composed in a node is static during a nodes lifetime. If a node needs to be updated in some way then it needs to be re-provisioned to be able to be part of the network once more. The mesh network operates in a client/server paradigm where the elements are in a particular condition, called state. Elements that exposes their state(s) are called servers and elements that accesses states are called clients. When a message is received by an element it sends it to one of its models; which model it sends the message to is dependent on the message.

Models

Nodes can have several different functionalities and each of these functionalities are defined in what is called models. Models are attached to the elements and each element has one or more models. The models define the states that the elements are in and how messages changes the states in order to achieve the wanted functionality. An element may not have several overlapping models; two models are overlapping if a message can act upon both of the models. If two of the same models need to be present in the same node then the node will need to have two elements.

An application in a Bluetooth Mesh network needs to define a Server Model, Client Model, and a Controller Model. A Server Model defines the messages it can receive and transmit. It also defines how states, and therefore the element, will act upon reception/transmission of messages. A Client Model does not contain states but rather only defines the messages that can be sent to a Server Model and act upon the value of the state that has been requested of the Server Model. Controller Models contain functionality to make Client Models and Server Models communicate with each other. All these Models can be included in a single device.

The Server Model, Client Model, and Controller Model are by no means the only Models that can be created in a Bluetooth Mesh network, they are merely the required ones in order to create an application due to the intrinsic Client-Server paradigm of the network. Models for key management, relaying of messages, power control, and lightning control are some other examples of Models that may be required in an extensive Smart Home Bluetooth Mesh network.

Messages

The communication between the models is based around the publish-subscribe pattern. When using this pattern no message that a sender, i.e. publisher, sends is directed at any specific node. Instead the publisher sends out the message to an address, which can be a unicast, group, or virtual address. Elements that would want to receive the message then subscribe to these addresses. For example, a

node that contains a light model is positioned in the kitchen and thus subscribe to the kitchen group address, when an "on" message is sent to the kitchen group all of the light models that subscribe to this group are turned on.

The messages that are sent to the elements operates according to the state of the element. Servers define what messages can be used at which state, and thereby clients can ask the server for which state it is in and which messages it can send in order to change the state. Some messages will trigger a response messages, these are called reliable messages, and if the messages do not trigger a response they are called unreliable messages. In the case of reliable messages, the response message is directed directly towards the source address of the incoming message and not to the models publish address. Hence the messages that are processed by an element are either directed directly towards the element, to an address that the element subscribe to, or toward the group in which the element is a part of.

Messages have an opcode, associated parameters, as well as a behavior. It is the opcode that defines what type of message it is and thus by checking the opcode the element knows which model to send the message to (since overlapping models are not allowed by protocol definition).

Security

Security in the Bluetooth Mesh protocol is handled in several steps. There are two different main types of keys that are used in order to produce a secure network: application keys and network keys. Encryption and decryption are made in the Network Layer as well as the Upper Transport Layer. At the Network Layer the messages are encrypted/decrypted in the context of the network the node is a part of with the network key. All the nodes in the same network uses the same network key in order to communicate with each other. After a message has been received in the correct context and authenticated it will be sent up to the next layer of the stack. Access Messages that are meant for models are again encrypted/decrypted in the Upper Transport Layer before being processed by the model/sent to the Lower Transport Layer. This is done with the help of an application key and the Access Messages specifies which key that is supposed to be used in order to decrypt the message. The application keys are also shared between the nodes.

In addition to the two types of keys, there is a third special key that is called Device Key. This key is an application key that is only shared between a node and the provisioner, it is used to secure communication between only them. It is the provisioner that provide this key as well as all the other keys, it also sees to it that they are regularly updated. Application keys are bound to the network keys, this to ensure that only application keys that are used in the context of the network is functioning.

The algorithm used throughout the protocol is the AES-CCM encryption. Each ciphertext is verified with a four octet MIC (message integrity code, same as message authentication code), or eight octet MIC if only encrypted at Network Layer

(which is the case for e.g. Transport Control Messages). After encryption in the Network Layer obfuscation is done with AES-ECB and XOR bitwise operations. This is not deemed safe encryption, but given that its purpose is simply to obstruct passive eavesdroppers it might be sufficient.

2.3.4 Provisioner

The provisioner is the role undertaken by the node that organizes the mesh network. As described in section 2.3.2 the provisioner is the node responsible for managing the network resources. In short it can be said to be the creator and maintainer of the network.

A device that is not a member of the mesh network is called an unprovisioned device. It is the provisioners job to make an unprovisioned device into a provisioned node (just simply called node). In order for this to be possible, the unprovisioned device need to be given the aforementioned network resources. Since the unprovisioned device does not have, for instance, a network key it cannot communicate directly to the Provisioner in the context of the mesh protocol. A separate provisioning protocol is used in order to bridge this gap. The unprovisioned device advertises its presence to the Provisioner, which in turn authenticates the device and allocates necessary network resource. These resources are delivered to the unprovisioned device and this in turn allows it to operate as a Bluetooth Mesh node.

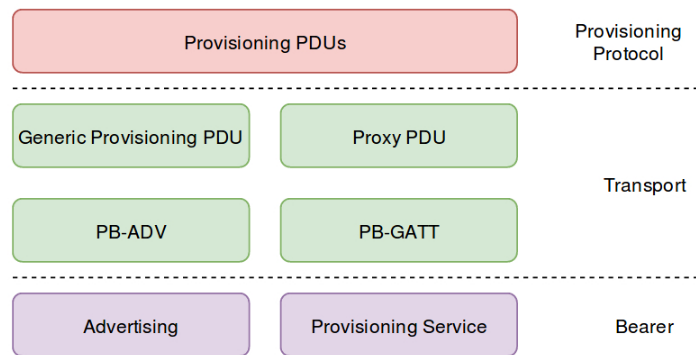

Figure 2.6: Provisioning Protocol Stack

Figure 2.6 shows how the provisioning protocol is structured. Provisioning PDUs are used in order to send relevant information between the provisioner and the unprovisioned device. The transport layer of the provisioning stack decides how the provisioning PDUs are supposed to be segmented into manageable pieces for the underlaying Bearer layer when transmitted, as well as reassembled when received.

How the provisioner is implemented is not strictly specified, different units could give different amount of freedoms and usability for this. However, since e.g. phones

are recommended to be used for this role it will most likely be very easy to set up a Bluetooth Mesh network.

Chapter 3

Implementation Considerations & Requirements

Using the theory that was presented in the previous chapter we will now use this chapter to observe what difficulties that might arise in an implementation that aims for interoperability.

3.1 Mesh Nodes

A functioning Bluetooth Mesh network that demonstrates its core concepts can be summarized by including the following four devices:

- LP node (with server capability)
- Friend enabled node
- Relay enabled node
- Client node

The simplest network possible that includes these could be explained by figure 3.1. Here a simple Client advertises a message with the LP node as destination. The Relay node is the only one in proximity to receive it, it does this and relays it to the Friend node. The Friend caches it for the LP node since they have established a friendship, and sends it upon reception of a Friend Poll message from the LP node.

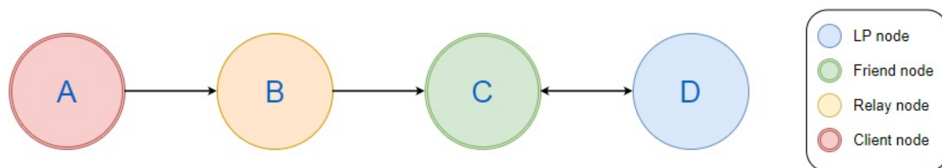

Figure 3.1: Demo Network

For a full implementation of the protocol a provisioner also needs to be implemented to set up and configure the network. This functionality is however outside

the scope of this thesis.

All nodes in the Mesh protocol acts as a multi-role broadcaster-observer, as described in 2.2.1. How the different SDKs function in order to create the roles are discussed in section 4.4. Besides this, some features require more advanced state machines than used in the BLE specification, some of these rely on timeouts. In order to be able to implement the protocol on top of existing BLE stack, depending on what role is implemented access to one or several of following functions are needed:

- Setting user defined advertising data
- Starting a custom advertising interval with timeouts
- Starting a custom scanning interval with timeouts
- Timers

We make notice of the fact that if manual stop of scanning and/or advertisement is required, due to e.g. lack of scanning/advertisement timeout functionality, then timers are required for that specific functionality as a substitution. The requirements on the suggested devices are summarized in table 3.1.

Table 3.1: Mesh Feature Requirements

Role	Timers	Set User Def. Adv.	Start Scan	Stop Scan	Start Adv.	Stop Adv.
Relay	N	N	Y	N	Y	Y
Client	N	Y	Y	N	Y	Y
Friend	Y	Y	Y	N	Y	Y
LP	Y/N*	Y	Y	Y	Y	Y

*If timeout functionality is missing for advertising and/or scanning procedures, and only manual stop is available, then this requirement must be fulfilled as well, otherwise not.

Out of the nodes required it is the LP feature that is of most interest to port, this since it is here most hardware restrictions lies. Friends are supposed to be run on more powerful units, which in general wont have the same restrictions, and therefor have greater chance of achieving interoperability. This since they may run on a real Operating System (OS) where a complete implementation can be imported. The OS libraries will then take care of hardware connections, meaning that the porting is already done, as long as the same programming language and version is used.

As can be seen from table 3.1 the roles with highest requirements is the nodes with the Friend Feature and LP feature. Aside from these demands in SDK functionality, the Friend feature is also the role that have the highest requirements on hardware, this since it needs to cache messages for potentially many LP nodes,

and thus requiring memory space and higher concurrency capabilities. The Friend node can never be a LP node and it is ideally thought of as being a device with high capabilities. Ordinarily it should not be constrained by limitations in the way a LP node is.

3.2 Compilers

Some issues that might arise on the SoC when porting includes how words (integers) are represented in the different processors running the application, while more obvious reasons could be just to get the code to compile and behave as expected. The latter would typically have to do with the different structures of the SDKs.

We identify the following potential issues in the SDKs:

- SDK access to the stack
  - Is it protected by some interface layer?
  - Are the side effects of the corresponding functions in two different SDKs the same?
- GAP profiles
  - Are they fully implemented in the SDK?
  - Are they designed to be event-driven and/or thread-driven?

The choice of programming language and compiler is a very important aspect. Software in embedded systems have traditionally been written in C but in later years C++ has begun to take over more and more mainly due to the improved capabilities of embedded devices. As described in [5], C++ is a very complex language that can produce different behavior depending on which compiler is used, even when conforming to ISO standards. Different hardware can have different processor architecture, which can lead to difficulties finding a common compiler for interoperable software. In this thesis, a combination of C and C++ was used to produce the code, while IAR Embedded Workbench was used in order to deal with the compiler issue. IAR Systems is a software company that produces development tools and compilers for different types of embedded systems. IAR supports the different SoCs that was used for the implementation. The hardware will be presented in chapter 4.

3.2.1 Toolchain

A software toolchain typically consists of a compiler, linker, libraries and a debugger. This means that even if the same compiler is used for all platforms, there might be differences in how the toolchains is configured for each SoC and SDK. Given the same compiler, other issues to be careful of would be different libraries enforced by the SDK.

3.3 Wrapping and General Hardware Limitations

Since a SoC does not run on an Operating System there are in general a number of standard library resources usually available for a desktop computer, running on e.g. Linux, MacOS, or Windows, that is not available on the SoC. Such resources could include threads, timers and printing. The replacement solutions functionality for these resources could vary a lot between two different SoCs, therefor it is important that these resources be wrapped in such a way that their wrapper functions are handled in a generic way, and its use is flexible. If the design is heavily dependent on the manner in which one of these resources functions interoperability of the protocol could be heavily jeopardized. However, keeping wrappers for some functionality is a good way to ensure internal functions to function properly across the different platforms.

The issue of memory could also be prominent. As have been mentioned before the Bluetooth Mesh protocol works as an extension on top of the BLE specification. A SoC manufactured for BLE is typically designed to be as power efficient as possible, meaning the hardware, including memory should be as minimal as possible. Extending the BLE protocol in an inefficient way could leave little memory left for applications. Furthermore, since its release under a Bluetooth specification in 2010 a lot of different manufacturers and developers have implemented the BLE stack in their own way. Depending on this implementation the stack could rely more or less on flash memory versus SRAM. The space left for either of these could be a limitation on the hardware.

Chapter 4

Hardware

In this thesis we have surveyed three different SoCs made available to us. They are all mature chips that work well with the BLE Specification. Even though they do this well they all have their own SDK implementing the BLE stack and does so with their own design and dependencies. In this chapter we explore their specifications, BLE capabilities, and stack interface closer.

4.1 Nordic Semiconductor nRF52832

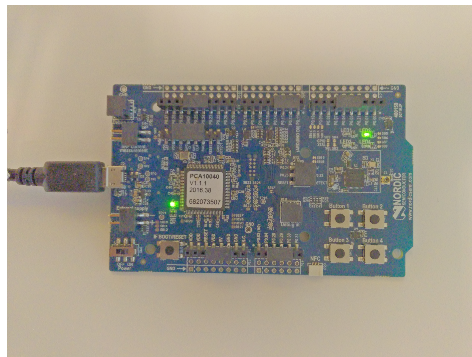

Figure 4.1: nRF52832 SoC

4.1.1 Features

- 32-bit ARM Cortex-M4F Processor
- 2.4 GHz bandwidth
- 512kB flash + 64kB RAM
- Bluetooth 5 support
- Parallel Bluetooth Scanning and Advertising

4.1.2 SDK

The nRF52832 SoC is supported by the nRF52 development kit. Software support for the nRF52832 SoC is split in two major parts: SoftDevices, which are complete wireless protocol stacks, and the nRF5 Software Development Kit (SDK). For concurrent programming it uses a Real-Time Operating System (RTOS) package called FreeRTOS, which provides thread-similar capabilities and real-time timers. It does however have its own hardware based timers that in some sense can run concurrently with callback functions.

The SoftDevice is used for Bluetooth communication and works as an interface between the SDK and hardware. The specific SoftDevice used in the project is called S132 [13].

One limitation with this chip is that it does not posses the necessary output parameters for the AES-CCM encryption that is required by Bluetooth Mesh. Instead a software solution is required that will take up space and energy.

4.2 Silicon Labs BGM121

Figure 4.2: BGM121 SoC

4.2.1 Features

- 32-bit ARM Cortex-M4 Processor
- 2.4 GHz bandwidth
- 512kB flash + 64kB RAM
- Bluetooth 5 support
- Parallel Bluetooth Scanning and Advertising

4.2.2 SDK

The SDK belonging to the BGM121 [12] module comes with a complete BLE stack and a powerful event-handler that delivers all possible BLE events through both a blocking and non-blocking wait. With high-priority timer callbacks and how it allows the developer to define his own events it can in many ways substitute an RTOS. As of time of writing/coding support for FreeRTOS (or any other) was not available on this product line, a huge downside as most other investigated SoC/SDK pairs in some manner could offer a concurrency solution in the form of threads. The event handler, with built in timers and user-defined-event capabilities, should be enough to implement the solution.

The structure of the BLE Stack implementation allows for easy extension and control, this in the form of "plug-ins". Predefined plug-ins defined as well as user-defined plug-ins are auto-generated in a graphical user interface when starting up the developing software that is delivered with the kit. This interface allows the developer or client to easily "drag-and-drop" between plug-ins for a BLE system. Picking just parts of the BLE stack is fully possible, thus defining the Bluetooth Mesh (or even all its layers each) as such a plug-in is fully possible. Adding the Mesh Stack on top of the existing functionality is not hard. However, large parts of the stack are unaccessible, meaning that to some extent the developer have to conform to the SDK.

Hardware accelerated AES-CCM encryption is available in this SoC.

4.3 Texas Instruments CC2650

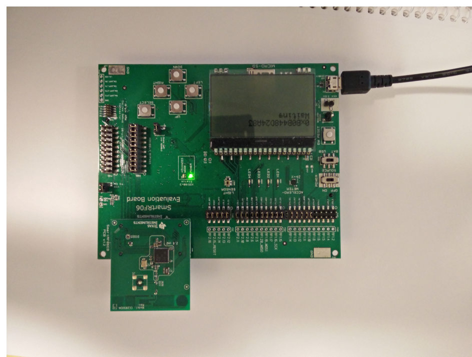

Figure 4.3: CC2650 SoC

4.3.1 Features

- 32-bit ARM Cortex-M3 Processor
- 48-MHz Clock Speed

- 2.4 GHz bandwidth
- 128KB of In-System Programmable Flash
- 20KB of Ultralow-Leakage SRAM

4.3.2 SDK

The CC2650 chip uses an "Ultra-Low Power Wireless Microcontroller Platform" that Texas Instruments (TI) calls SimpleLink[11]. It has the capability to use several different low power consuming wireless technologies including Bluetooth Low Energy. It implements the Bluetooth 4.2 stack and should thus be capable of handling Bluetooth Mesh. SimpleLink has an expressed purpose of giving 100% code portability between different devices that are using the SimpleLink platform.

TI release all their micro-controllers, including SimpleLink, with their own RTOS solution, TI-RTOS, which is a real time, pre-emtive, multithreaded OS. The SDK available for CC2650, and its corresponding Bluetooth Stack is heavily dependent on using TI-RTOS. Communication between application and BLE Stack happens through a software layer called Indirect Call (iCall), which is designed to handle the communication in thread-safe manner. It is possible to remove iCall and TI-RTOS if one wishes to access the BLE stack directly, but this requires a lot of work and removes the whole point of TI-RTOS [14], which is to streamline the application development process. iCall allows applications to register itself (in the form of a task/thread), and only registered tasks may access the BLE stack.

AES-CCM encryption is available for this SoC only in the form of a software package integrated into the SDK.

4.3.3 Summary

As can be concluded from the SDK descriptions the different platforms function quite differently. A summary of what possessed functionality is desired and/or needed by the different nodes can be seen in table 4.1.

Table 4.1: SDK Features

SoC	Timers	Threads	Blocking Events	non-Blocking Events
nRF52832	Callback	Y	Y	N
BGM121	Event	N	Y	Y
CC2650	Event	Y	Y	N

Summary of required functionality, as described in chapter 2 and its availability can be seen in table 4.2.

Discrepancies in architecture can be seen mainly in how the Silicon Labs SDK needs to follow a strictly event-driven design, without concurrency neither in the form of threads nor timer callbacks. Instead if offers a more powerful event handler.

Table 4.2: SDK Mesh Feature Functionality

SoC	Set User Def. Adv. Data	Start/Stop Scan	Start/Stop Adv.
nRF52832	Y	Y	Y
BGM121	Y	Y, but no timeout	Y
CC2650	Y	Y	Y

4.4 Role Creation

Since all Bluetooth Mesh nodes need to implement both the observer and broadcaster role, exactly how this is done in the different SDKs is of vital importance. It more or less defines what is portable in the total mesh implementation and what is not. This section will hence be dedicated to investigating these differences.

4.4.1 Nordic Semiconductor nRF52832

In nRF52832 with the S132 SoftDevice there are no clear distinction between the observer and broadcaster roles. It can be defined whether the device running the SoftDevice should be considered a peripheral or central device but no further restrictions apply on the broadcaster/observer roles since they can run concurrently with each other. The SoftDevice can even be configured in a manner that lets the device be all four roles simultaneously. This implies almost no restrictions on the implementation of the Bluetooth Mesh since start/stop of advertising/scanning is all that is needed.

4.4.2 Silicon Labs BGM121

Role configuration in the BGM121 is very similar to that of the nRF52832. It offers concurrent broadcaster, observer, central, and peripheral roles, while just as in the nRF52832 there is no clear distinction from a configuration point of view. The only thing the developer needs to take into consideration is to use the appropriate parameters when making a call to start scanning or advertising, as these functions requires specific advertising/scanning modes. This can be seen as

an on-the-fly configuration, and as mesh always uses the same roles this implies almost no restrictions on further mesh implementations.

4.4.3 Texas Instruments CC2650

In the CC2650 the user has to specify which roles the device should perform and how these roles should interact with the application. Events are generated according to the specified roles and the parameters given at the initialization. How these events are handled is then up to the user after the initialization is done. How to implement the four roles is explained by the the SDK guide, but as mentioned how they interact with the application is up to the user.

The design is made so that one thread should be designated to run the iCall interface. A second thread should then be configured to receive and handle GAP events, and in turn send these to a third thread running the developer-defined application. Events are generated by reception of GAP messages, scanning/advertising timeouts, timer timeouts, etc. It is thus a highly event driven design, with a default concurrency solution that offers great control and delegation of tasks.

4.4.4 Summary

An issue we already can identify is that the implementation of the GAP roles differs from platform to platform in a meaningful way. The important aspects of porting nodes between platforms depends upon thread-event driven design, timer functionality, and GAP profile implementations. These aspects are summarized for the different platforms in 4.3.

Table 4.3: Platform Designs

SoC	Design	Timers	GAP integration
nRF52832	Event with optional threads	Callback	Fully
BGM121	Event	Event	Semi
CC2650	Thread-Event	Callback & Event	None

All three SDKs are event-based, but exactly how differs quite a lot. On the Nordic Semiconductor SoC all GAP events have their own blocking event handler, while e.g. buttons have another. Furthermore, timers use callbacks to do what they need to do.

In the Silicon Labs SDK all events are handled in the same event handler, both blocking and non-blocking. The events, delivering data for every kind of event

that happens on the chip gives a very intuitive and easy way of designing a LE device. It does however force the developer to conform to this architecture, and it is highly dependent on library function hidden to the developer.

In the TI SDK the GAP profiles are not pre-implemented, instead the SDK is designed so that the programmer have to do this, which is tedious and not very intuitive, but comes with the benefit of great control. Due to the iCall architecture it is hard for other SDKs to imitate its behavior, while the reverse might be easier.

Chapter 5

Modeling

With the analysis made on the requirements discussed in chapter 3 and the hardware discussion in chapter 4, we now examine how to model an interoperable solution in this chapter. A proposed solution in accordance with the perceived issues will also be presented.

As the primary objective is to explore interoperability issues in the protocol with regards to SDKs and SoCs, the main focus will be put into the protocol software. The aim is to make this portable to different SoCs and SDKs, instead of focusing on application software. Little effort will thus be put into design of the application.

5.1 Design

To create a design based on the discussion made in chapter 3 we are lead to realize that the protocol cannot just simply be abstracted to the highest desirable level and ported to the different hardwares. A component based style seems appropriate so as to isolate the functionalities in several different components, and then try to port these. A good starting point would be to divide the Bluetooth Mesh Protocol into two main parts, the Protocol Stack and the nodes functionality. Both of these parts needs to be implemented in order to get a functioning network. The stack is of course used by all nodes internally, but it is the same stack that is being used no matter the node, only with more or less required functionality.

Why this division is regarded as essential is not only based on the plausibility of this abstraction, but also on the nature of the features, as described in section 2.3.2. These features are supposed to be an intrinsic part of the protocol, which are activated depending on the functionality a certain node is supposed to have. Some of the features may require access to some sort of timer or similar hardware capabilities. To implement them inside the stack could thus lead to some serious porting difficulties and the division seems crucial for an interoperable design. In other words our approach will focus on the architecture of the software and make a division into components according to perceived difficulties in accordance of [4].

This is not the only way to design it, as e.g. the surveyed SDKs (see chapter 4) often integrate all layers of the BLE stack into the whole SDK. In the TI

SDK the stack is layered with an access interface, which provides its own benefits. The SoftDevice in the Nordic Semiconductor SDK instead uses C structs that are checked for validity and errors by underlying libraries and connecting the stack to the transceiver for Received Signal Strength Indication (RSSI) in ways not extractable to the developer.

5.1.1 Mesh Stack

The Mesh Stack encodes and decodes the messages that are needed for communication. In order to make the implementation as easy to port as possible, all layers of the Protocol Stack that lies above "BLE Core Specification" should be implemented as a standalone stack to be used as its own component. All SDK functions should be wrapped and presented only as input on a call to encode or decode on the stack. Furthermore, as some functionality of the stack is only used by certain features these parts can be set to be defined only when configuration is made for that feature to be enabled.

This separation and these wrappings aims to eliminate direct and integrated dependencies toward a certain SoC, while minimizing any porting problems for the stack and any internal callback function. The only dependencies the stack will have outside of itself after doing this separation is input parameters, advertising data output, and encryption. However, both the nRF52832 and CC2650 SoC lacks any hardware accelerated encryption. To solve this issue a standalone encryption library is used, which in turn eliminates platform dependencies.

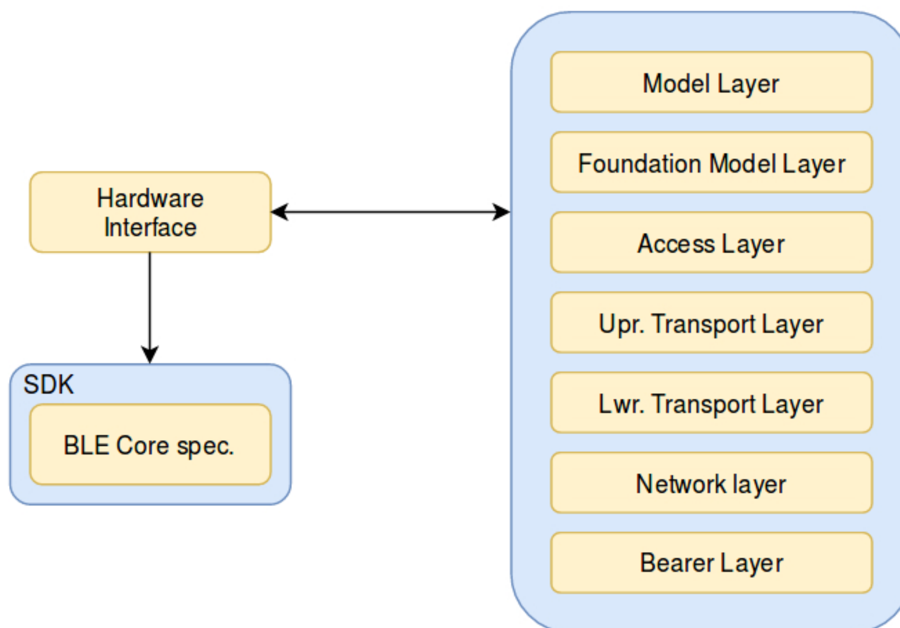

Figure 5.1: Design Suggestion

Making these abstractions is absolutely essential for keeping a stack interoperable over the different platforms, even if it puts larger responsibility on the developer and potentially lowers the efficiency of the stack.

The stack and these feature-components will be connected through a hardware interface that also will handle the specific calls to the BLE stack in the SDKs. This Hardware Abstraction layer is designed to improve the codes portability[5], and works as a middle-ware between the different SDKs. It will at the least work as the lowest common denominator in the process of abstracting functionality. An overview of our suggested stack solution can be seen in Figure 5.1.

On a note, one other consideration that makes the stack not fully interoperable is the Model Layer. The models are responsible for the required behavior of the node being performed. When a message is received by the SDK in the BLE stack and fetched by the Hardware Interface in order to be sent along to the mesh stack for processing the message will eventually arrive at the Model Layer. In the model some form of communication with the hardware is essential since the state of the node may be needed to change in accordance with the received message. For instance a received ON message will need a light to turn on, that functionality is not possible to be implemented in a generic way due to the simple fact that lights on the different hardwares can handled differently.

5.1.2 Functionality Components Design

As was discussed in 4.4 the architecture of the different SDKs is quite distinctive from each other, especially when it comes to how the GAP roles are integrated into the SDK, as well as how it communicates with the BLE stack. This is the largest issue when designing the components that are supposed to handle the functionality, as every node must be both an observer and a broadcaster, which means that nodes needs to at least at some point scan for incoming GAP messages. Received messages are in turn sent to the stack where they are decoded and handled properly. If an advertising response is required for the received message the stack will call upon the Hardware Interface to do so, and at the same time if the internal state of the node needs to be changed the stack will set the required state variables through the Hardware Interface.

The most primitive case is the relay node that does nothing but scan for GAP messages and then discards or re-broadcast them depending on the TTL value.

In addition to handling incoming messages and sending these to the stack, both the LP and Friend node needs a state machine to let them know if they are befriended or not. The states in this machine are based upon what amount of advertisements has been sent and/or received in a certain amount of time. The LP node also needs to be the initiator of advertisements and not only to react upon their reception.

Finally we have the Client node that needs to be able to initiate an advertisement upon request of the user, this could be e.g. when pressing a button. In this case a

handler listening for this event is required, our suggestion is to put it into a thread.

Hereafter comes a deeper analysis of the features, which will be used in correlation to the analyzed platforms in order to make our design. This is done in an iterative manner, trying to expand more advanced functionality on the simpler roles, aiming to keep a design feasible for all platforms.

5.1.3 A Simple Relay Node

A good starting point is by looking at how to design a simple relay node, as this node uses logic that needs to be implemented by all nodes, namely scanning, advertising, and making calls on the stack. The prerequisite for both scanning and advertising is that both are made in observer and broadcaster GAP role modes. We remind ourselves of the discussion made in 4.4 of the different SDKs to see how this simple functionality can be implemented:

- The nRF52832 allows for these simple features quite trivially. The s132 Softdevice in the SDK is pre-configured to simultaneously act as all GAP roles. The only thing the developer needs to do is call on the appropriate library functions to act out the role in question.
- The BGM121 also allows for these simple features quite trivially. Role configurations are made on the fly when wanting to scan or advertise.
- The CC2650 needs to be pre-configured for the different GAP-roles. Besides this an implementation is needed by the developer in the form of a GAP thread that takes care of the different GAP events. A lot more work is required than in the previous cases, but same functionality can be achieved without compromise.

From this summary we conclude that for the simple task of relaying both the nRF52832 and the BGM121 SoC seems to offer a platform to build a generic solution, whereas the TI in turn needs a solution conformed to the design of the underlying GAP and iCall threads.

To instead conform the Nordic Semiconductor and Silicon Labs solutions to TI is a bad choice for a number of reasons, the first one would be that the increased layer that is not necessary would increase memory usage and power consumption, both of which is highly undesirable. The second one is that the Silicon Labs SDK does not offer any concurrency solution, meaning that such a solution would be half-made, and in turn also highly edited and adapted towards Silicon Labs. The third is that even if this layer is created, it would work as a non-necessary simulation of the TI solution, running with no dependency towards a protective BLE stack access layer, as TI does with its iCall thread.

Next we take a look on how GAP messages are received in the different SDKs, and how events are accessed there. The interested reader can find more detailed signatures in appendix A.

nRF52832

The Nordic Semiconductor SDK makes the developer register a function to the Softdevice with a call to `softdevice_ble_evt_handler_set`. The parameter of type `ble_evt_t` that is forwarded to this function holds the GAP messages received. They can also hold other messages from the BLE stack, such as GATT. The event is fetched from an event queue and will run in the context of the thread. The function call to fetch the event is blocking.

BGM121

In the Silicon Labs SDK events are fetched directly from an event buffer in the manner described in listing A.2. These events include all events that exists in the SDK, from a triggered button, a timer timeout or a GAP message. The idea is then to branch on the event type and handle all possible cases in the context of the main function just after the call to `gecko_wait_event()`.

CC2650

By design the application and GAP are divided into two threads, with a third underlying iCall thread that communicates with the stack. Since iCall is non-optional on the CC2650 this means that using TI-RTOS is neither. The design of the SDK and the event handling is slightly different from previous platforms. The developer writes his own GAP thread and registers the thread to iCall with the function `ICall_registerApp`. The semaphore associated with the thread unlocks the blocking function `ICall_wait` when an event is received inside iCall. At the point the developer can fetch the message inside his GAP thread with `ICall_fetchServiceMsg`. Although it is possible for the programmer to then put all logic inside this thread, further profile files in the SDK is designed so that its beneficiary to use a third Application Thread that receives messages from the GAP thread. One of such benefits are that the programmer easily can work around the blocking function `ICall_wait` in the application thread if needed, others are that the application does not need to handle timer timeouts.

Summary

So far it would be possible to design a Nordic Semiconductor and a Silicon Labs solution that are very similar to each other. The TI solution could have the same logic, but this needs then to be factored out and put into its own thread. Disregarding this extra implementation work on the TI platform, the API of the Abstraction Layer is sufficient to make a portable component that can be plugged into each SDK, consisting only of relay function that takes the GAP message as input.

5.1.4 Extending to a Simple Client Node

A Client node needs the same basic functionality as a Relay node but with the relaying capability being optional. Besides this, for the scope of this thesis, it also

requires the functionality to generate an Access Message. In the proposed demo in this thesis this message would represent a request to toggle a LED light on a LE unit. The only extended requirement on this node is thus something to trigger the generation of this message, which could be a runtime initialization, a timer or a button. Button seems the most reasonable and is available on all hardwares, it is signaled through events of various sorts. On all surveyed SoCs this functionality exists and can be seemingly used without problems.

For both Silicon Labs and TI this event is delivered through the ordinary event handler, and the solution just needs a minor extension for both of these chips. However, the Nordic Semiconductor chip needs its own extension in the form of a button handler (or maybe extended event). The abstracted function to generate this message can be used across all three platforms without any issues, but the structure of where to call on it differs.

The problem can be described by looking with the pseudo code in Figure 5.2:

```
byte* event = wait_event();

switch (event->header) {
    case GAP:
        if (!for_me(decode(event->data), my_address) && IS_RELAY)
            relay(event->data);
        break;
    case BUTTON:
        send_access_msg();
        break;
    default:
        break;
}
```

Figure 5.2: Example of a Bluetooth Mesh Network Topology

In this code the relay feature is called from the abstraction layer with the same API signature for all platforms, namely `relay`. For all platforms the GAP event can be fetched in a similar manner, and it is thus easy to plug the function into the corresponding places in all architectures. However, when adding the functionality of a Client the problem arises. Both Silicon Labs and TI include all SDK events in the same event handler, meaning that the pseudo code structure in Figure 5.2 could be used. This would allow even further abstractions, namely just sending the event to the Abstraction Layer and letting a generic function handle the event. In the case of Nordic Semiconductors this is not possible since two different event handlers are used for the two cases. One way of solving this problem would be to implement an extra abstraction layer just for Nordic Semiconductors that puts both events in the same queue, but this creates unnecessary complexity and code size, which are both undesirable[6]. Relying on this kind of design would also enforce further porting to conform to the same idea, which could prove to get

more and more tedious. Therefor we wish to keep the abstraction of just the two functions `relay` and `send_access_message`, putting them both in their proper places in the corresponding platforms.

5.1.5 Extending to a Simple LP Node

A LP node needs the basic functionality of a Relay node, which is to send and receive GAP messages. In addition it also needs to hold a state machine that gets to react on incoming GAP messages and timeouts. Its behavior is described in Figure 5.3.

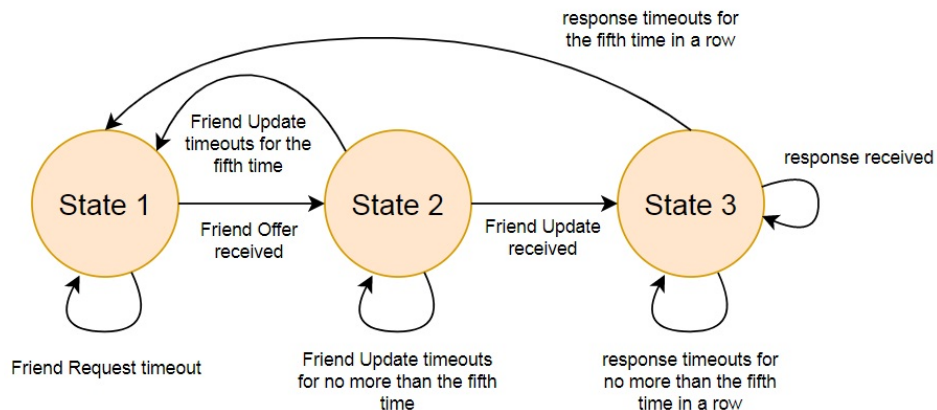

Figure 5.3: LE Node state machine

The intuitive way of handling this kind of behavior would be to do it in its own thread, with messages forwarded from the GAP Message event handler. This since it is dependent on timeouts and has a processing depth that needs time to evaluate, which could jeopardize concurrency if in the same scope (a new message is received when still handling the first one). This design is natural for both Nordic Semiconductor and TI, but for Silicon Labs it is not feasible due to the lack of concurrency solutions. Here instead the state machine needs to be extracted and placed just below the event-handling switch in the main method. This solution does not work with Nordic Semiconductor SDK for two reasons: its advertising have a minimum timeout of one second (timeouts of $100ms$ required) and its event handler does not handle timer timeout events. It must thus rely on timer callbacks to set signaling variables instead. This means that the Nordic Semiconductor platform will have to implement the state machine in its own thread, as timeouts indeed are a requirement for state changes (as can seen in Figure 5.3).

The TI SDK allows for both of the solutions mentioned for Nordic Semiconductor and Silicon Labs. This since timers can set signaling variables through a callback function and/or generate an event that triggers the event handler. As threads also

are present it can chose to let this functionality to be handled in its own scope, just as for the Nordic Semiconductor platform. Even if the ideal way would have been to wrap a state machine thread in the Hardware Interface, and just call on its creation during initialization, this is not feasible. It is however possible to wrap the whole state machine in its own function that will be called after the event handler, or in its own thread. The best possible abstraction then becomes functions in the Hardware Interface, however some custom placement of these seems to be required.

5.1.6 Extending to a Simple Friend Node

The Friend, just as the LP node, needs the same functionality as a Relay node, but with the extension of a state machine. The Friend state machine can be seen in Figure 5.4.

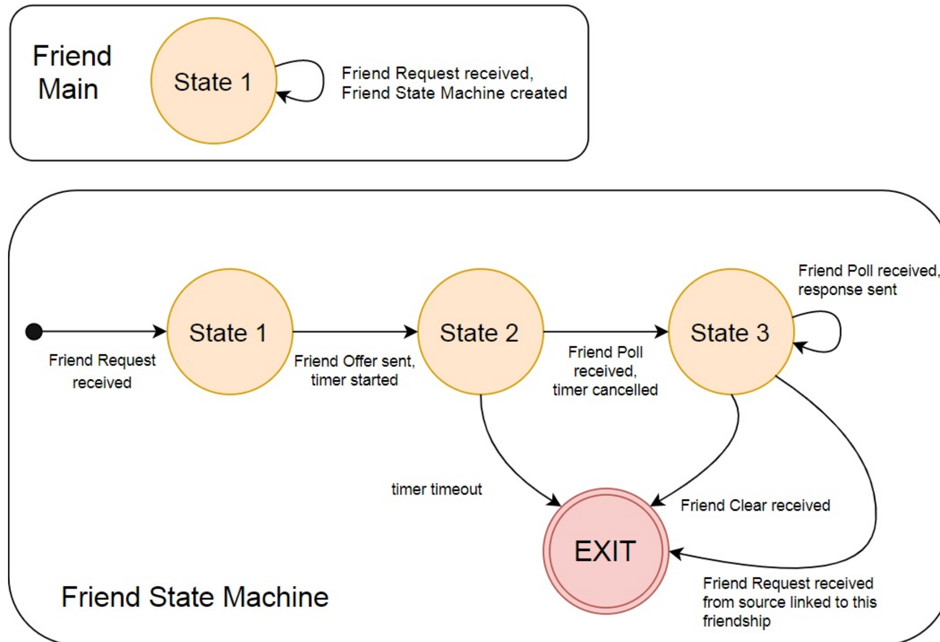

Figure 5.4: Friend Node state machine

In addition it must also have timers independent from scanning/advertising timeouts, this because it is supposed to scan all the time. Timeouts are directed to signal a timeout of the reception of the friendship-confirming Friend Poll message. It could also potentially need to be able to handle several friendships with LP nodes at the same time, meaning concurrency indeed is the best design. In a real applicable Mesh Network this node would probably be represented by a more powerful and static device, which has a constant power supply, but having a SoC with a power cable connected as a Friend Node in a network is not an unfeasible idea. No great focus on implementing the Friend node on more than one SoC will

be given since the functionality is primarily meant for more capable devices than the surveyed SoCs.

Our proposed solution is similar to that of the LP component, but with timers that callback to notify failed friendship.

5.1.7 Summary

With the case of the simple Relay it would be possible to implement a complete abstracted solution that works on all platforms, where porting is required only in the Hardware Interface. Such a solution only needs to be a function that takes as input a GAP event that it can branch on. The only thing needed on the respective chip would be to call this function upon GAP message reception. TI still needs a lot of extra work though, but this is a limitation of the SDK architecture. Even if implementing an ordinary non-mesh LP node this extra work would be required in comparison to the other platforms.

With the expansion of more complex features it is apparent that the discrepancy of functionality that is required for the components grow larger. This is not an issue for the former BLE Core Spec. that only sends static data during advertising, and uses GATT for connections. However, for mesh friendships timers are a new mandatory thing, and how they are handled across platforms differs. The results is that even the two previous similar SDK architectures now have grown too far apart to use such an easy portable solution. It is of course still solvable without an excessive amount of effort.

5.2 Proofs of Concepts

To prove that the implementation of the protocol holds over the set of SoCs a few use-cases were constructed as a proof-of-concept. They aim to extend each other with more functionality and porting, each showing the kept interoperability.

5.2.1 Use-case 1: Nordic Semiconductor Friendship w/ Client

The first use-case is used for the initial implementation on the first hardware, which is the SoC from Nordic Semiconductor explained in 4.1. It is used to prove the functionality of the protocol and implementation.

By using only two Nordic Semiconductor nodes we will be able to prove that the Friend feature and the mesh LP feature works, with their internal state machines. In addition a Client will be built into the Friend node. The model used in this use-case is that of a light with a switch. The LP node offers a LED light which is accessed as its primary element.

The Client offers a button switch which when pressed upon will send a message to toggle the light. This is instantly sent to the stack which in turn will send it to

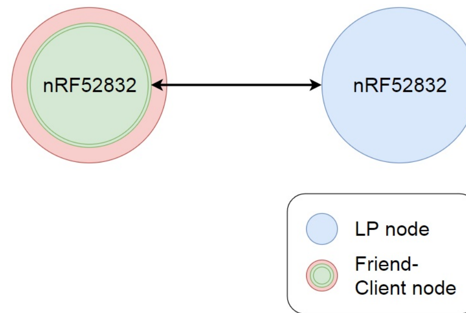

Figure 5.5: Implementation Proof of Concept Network

the Friends cache. From the Friend cache it will be delivered to the LP node after a Friend Poll message has been received.

5.2.2 Use-case 2: Mixed Hardware Bluetooth Mesh Network

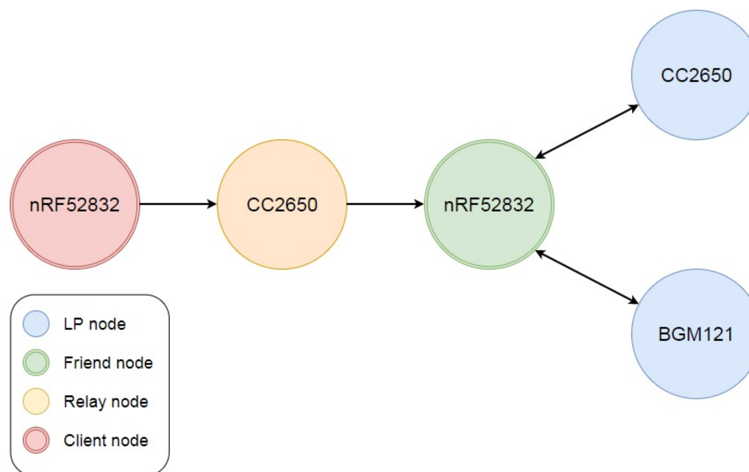

Figure 5.6: Interoperable Proof of Concept Network

This use-case involves all the relevant parts of a functioning and provisioned Bluetooth Mesh network. Topologically the biggest difference from the previous use-case is that the Relay node have been added, that it includes an additional LP node, and that the Client have been extracted to its own node. This will firstly prove that the Relay feature works, secondly that the Friend can cache messages it receives through its scanning process, and thirdly that the Friend is capable of storing and sorting messages for different nodes.

This Use-case also takes different Hardwares into account. As can be seen in Figure 5.6, no two neighboring nodes are of the same SoC. This aims to prove

communication interoperability between different SoC. Furthermore neither of the BLE nodes are of Nordic Semiconductor, instead a ported solution to TI and Silicon Labs is used. This aims to prove that a ported solution does indeed work across all surveyed SoC.

5.2.3 Use-case 3: Every Role on Every Chips

The aim of the final use-case is to try to detect any more interoperability issues that might not yet have been foreseen. It spans over several iterations where all features and node-roles are ported at least once to every SoC. It was mentioned in section 5.1.6 that no great effort will be put on trying to port the Friend node due to the fact that it is supposed to be implemented on a device with greater capabilities than the Low Power Nodes. However, to test if it is possible can shine some light on previously undetected issues. To what extent this use-case will be dealt with is up to time constraints.

Chapter 6

Results & Evaluation

With the help of the modeling considerations presented in chapter 5, a solution was implemented on the different SoCs. This chapter will focus on how well this went and try to draw some conclusions whether or not a generic solution is a feasible and worthwhile endeavor when Bluetooth Mesh capabilities is meant to be brought into different hardwares.

Interoperability is a quite difficult thing to evaluate since there are no clearly defined metrics to measure. One metric that can be used to give some sort of indication and support for the claims made are e.g. SRAM and Flash Memory analysis. The theoretical case of a software update on the protocol can be applied as well: if the protocol is extended, can this be implemented in the components only or does it need extensions on all platforms? The most important aspect however, is whether there exist some design solution that can be applied to all platforms, with some porting required. If reinvention of software component design is required, then that software component is not interoperable.

6.1 Use-cases

The use-cases, described in section 5.2, provided a way to iteratively grow the network and its functionality. By having this approach it was possible to, in each step, test to see if the desired functionality was achieved, as well as to evaluate interoperability aspects as the system grew. How, more precisely, the process of porting each node was handled will be dealt with in upcoming sections, but a quick overview of which use-case deemed possible and not will be provided here.

Implementation of the first use-case did not encounter any interoperability or porting issues, this of course due to the fact that the same hardware, the nRF52832 SoC, was used for all of the features. A confirmation that the implementation of the LP and Friend features and Client capabilities were functioning was achieved, and the porting portion of the project could commence.

Use-case two shined a light on the main issues that will be discussed in the upcoming sections of this chapter. A network of the kind that Figure 5.6 describes was indeed possible to achieve, but the amount of work required to do so was not

proportional to that of use-case one.

It was the third use-case that clearly highlighted the difficulties with the goals of the implementation. Every hardware is not suitable for every role, and to try and design the software for this is not deemed desirable. Evaluation of the use-case yielded that it is not feasible to achieve total interoperability due to the amount of re-invention required, in correlation to time constraints. The conclusion of this points towards poor portability due to design as well as functionality differences in the SDKs.

6.2 Component Implementation

With the quick overview of what did and did not work according to the use-cases, a more in depth analysis of why the results yielded what they did follows.

6.2.1 Protocol Stack

Due to the abstractions made into components, as described in chapter 5, the protocol stack proved to be highly interoperable. Almost no amount of code required to be rewritten when all the SDKs functionalities had been stripped off. The ease of portability of this component is, in addition to the stripping of the SDKs functionalities, in part an effect of the common IAR compiler used, which lead to minimization of compile time errors. Since the cryptography was dealt with in a software component and not via hardware capabilities issues arising from hardware demands could be erased.

The division made brought great ease into isolating issues arising from other components, and made node functionality implementation easier across all platforms. Our conclusion is that the protocol stack for decoding and encoding messages is fully interoperable and portable with the proposed design across the surveyed SDKs and SoCs. Should different compilers have been used then more work would have been expected to port the stack, however, this would be a porting issue only. Issues that for other platforms could jeopardize interoperability would only be an affect of BLE Core Spec. implementations in an SDK, i.e. if it would not be possible to isolate functionality into the middle-ware.

6.2.2 Relay Node

The Relay node holds one of the most basic features in the mesh network. The authors vision of an ideal implementation would have been to for each platform to wrap the necessary functions, as well as the thread functions available in each SDKs. Developers could then simply use the interoperable solution as an implemented API, so to in the main function of the application create the relay feature with a function call to something as simple as `create_relay_feature(...)`. This function could be responsible for setting up all necessary functionality and to run the relay feature automatically.

Such a solution was not possible across all platforms, mainly because the Silicon Labs SDK was lacking concurrency support. An additional issue with this kind of solution was the architecture of the TI SDK. Such a solution on TI would require the developer to use a number of pre-configured files, as well as a template main file. It is however deemed possible to do, but the common abstraction in Nordic Semiconductor and TI would be separated with a lot of additional resources and caution for TI.

For these reasons the lowest common denominator was to use the middle-ware functions described in chapter 5. Each platform would thus do their necessary implementation of a regular BLE application (configuring main function and threads for GAP roles) and then simply use the generic Relay functions in our middleware in the GAP event handler of said platform, just as when using each SDK as intended for regular BLE use.

With all of this in mind we deem the solution of the Relay feature to be both interoperable and portable. However, due to limitations in abstractability, scalability is limited. The amount of work required for using the solution of a mesh Relay feature lies solely on the amount of work required when using each SDK properly.

6.2.3 Client Node

The Client nodes functionality is in many ways as basic as the relay feature. Complexity in it lies solely on the application at hand, and limitations comes primarily from the hardware being used (if the SoC does not have buttons it is impossible to implement a light switch based on a button).

Functionality of the Client node is based on something to trigger the generation of a message, and then to send this message. Because of the differences in the SDKs on how events are handled (based on what kind of events they are), as well as the discrepancies of the SDKs discussed in 6.2.2, the lowest common denominator is on the same level as for the Relay feature. This functionality could be used by all platforms, requiring only as much work as for a regular BLE application to implement. A Nordic Semiconductor Client for example needs an additional event handler, which due to blocking functionality needs to be put into its own thread to prevent deadlocks. A Silicon Labs Client cannot do this, nor does it need to.

6.2.4 LE Node

As was discussed in chapter 5, the issues that arose due to lack of a common ground in some sort of RTOS [5] was prominent, and it forced the design to deal with the more serious issues of the design differences in events. What events were reported on can be summarized in table 6.1.

Table 6.1: Scanning and Advertising functionality across the SDKs

SoC	Scan timeout	Scan event	Adv. timeout	Adv. event	Timer timeout
nRF52832	Y	Y	Y, min. of 1 sec.	Y	N
BGM121	N	N	Y, in number of intervals.	Y	Y
CC2650	Y	Y	Y, min. of 1 sec.	Y	Y

On the Nordic Semiconductor SoC, due to a blocking event handler that only handled GAP events, and the limitation of timeout time, the state machine was required to be put in its own thread so that it could react in real time whenever a timer callback stopped the advertising or changed a variable.

On the Silicon Labs SoC, due to lacking RTOS support and the presence of an event handler that unlocked and branched on timer timeouts, the state machine instead could and had to be put right after the event-handler branching. If wishing to simulate a timer callback, this could have been done in the event-handling switch, this in order to extract the state machine as its own function while letting shared variables be hidden in a LP feature component file. Advertising and scanning timeout limitations, as seen in table 6.1, was not an issue as they could be substituted with timer timeouts.

TI, as before, allowed for more versatility, but with the cost of more work. On one hand it provided the capability of timeout for both scanning and advertising, while at the same time offering events for these features. This provided a very simple an intuitive way of handling all BLE events at the same place. It also offered functionality for timer timeouts, such that they either could be registered to an ordinary callback function or generate an event that would be handled in the main event handler. Texas could thus implement a very similar design to that of the Nordic Semiconductor and/or the Silicon Labs design. However, in both cases differences did arise, e.g. in regard to Silicon Labs it would be the presence of threads, while in relation to Nordic Semiconductor the difference in thread functionality (heap/stack based, more on this in section 6.5). In either case a lot of more work is required to set up a basic node, and a lot of more understanding is required by any developer who wishes to use the platform. This does in part defeat the purpose of an interoperable solution.

Even though it was possible to port this functionality across all platforms, the results were in some cases almost that of complete re-invention. The constraints held by the Silicon Labs architecture forced state machines into the main function, leaving critical function calls visible to any developer. This means that unless us-

ing pre-configured files the developer would be forced to partially implement the LP feature state machine. Nevertheless, the solution was portable but can not be considered fully interoperable, not as it is described in this thesis.

6.2.5 Friend Node

Implementation and design of the Friend feature had the same issues as described for the LP node in section 6.2.4. In addition, the complexity of a Friend, possible being able to have several friendships(restricted only by developer implementation) and creating a new state machine for each of these, proved to be unsuitable for the Silicon Labs platform. To solve the issue without concurrency support would require a lot of data structures to fetch the currently processed friendship, each requiring its own timers with their own IDs. Both timers and GAP messages would have to fetch the ID of the currently processed friendship. The authors believe such a solution could be made, but it would be of much less intuitive design than that used for the Nordic Semiconductor platform. As event handlers across the surveyed kits would react to timer timeouts so differently, and due to the lack of common RTOS support, the difference in forced design for the different kits would differ so much that a single interoperable solution across the platforms could not be deemed possible. Using this imagined Silicon Lab solution on the Nordic Semiconductor SoC would not have been possible for the same reasons described in section 6.2.4.

Beside this the TI CC2650 was not regarded as suitable at all to make this implementation, in part due to the limited amount of memory available on the chip, but also as it only possessed a single transceiver, meaning it could only transmit or receive at a single time, not both. These constraints made all devices but the nRF52832 unsuitable for a Friend node, and its feature was thus not ported. Interoperability was not analyzed further both due to these reasons, but also as the Friend is conceptualized as a more powerful unit than the investigated SoC. Interoperability of the Friend feature across the surveyed LE SoCs had thus less value than e.g. the LP feature.

6.3 Abstraction Layer

The abstraction layer worked as a nerve system for the components, connecting their interfaces and trying to minimize their dependency on the platform being run.

6.3.1 Hardware Interface

Issues in regard to wrappers, i.e. the middle-ware functions, in the Abstraction Layer was prominent, but not unsurpassable. These kind of issues arose primarily due to different function signatures as a result of underlying design patterns not visible to the developer. For example, when developing the initial protocol for Nordic Semiconductor a wrapper was made for the function to make a system call

to the SoC in order to begin sending advertising messages. Its signature ended up looking like this:

```
start_adv(uint16_t mesh_interval, uint16_t mesh_timeout)
```

where both parameters were needed as input for Nordic Semiconductor SDK function to start advertisement, and the first parameter were taken in milliseconds, while the second were taken in seconds. When later porting the function to Silicon Labs the timeout parameter instead was required to be an integer that represented the number of advertisement intervals to perform before timing out. Furthermore, in the Nordic Semiconductor case the timeout parameter could be set to zero in order to represent "no timeout", this was not possible in the Silicon Labs case, leading to internal porting issues. In the Silicon Labs case this option did not exist as the architecture of the SDK would force an event-driven programming style. Another more serious issue was that the Nordic Semiconductor SDK did not allow a timeout in under one second, thus forcing a timer callback to shut down advertisement when needed, something which was not an issue on Silicon Labs. TI default configuration was the same as the one in Nordic Semiconductor, but the SDK allowed for timers to easily generate timeout events in less than one second.

These kind of issues made some behavior of the wrapped functions behave differently depending on what hardware were used, and while not an issue for the more basic features like Relay or Client, it would increase non-interoperability of a single wrapped design for other features, such as LE.

Similar issues arose for scanning, in this case timeout was a required parameter in the Nordic Semiconductor SDK, but not possible in the Silicon Labs'. Here a timer was needed for Silicon Labs, which reported events in the same event handler as GAP messages. This functionality was not possible on the Nordic Semiconductor Softdevice.

TI uses initializations for scan durations, and then requires explicitly changing them if required. Nordic Semiconductor and Silicon Labs instead always takes interval as parameter when calling scan to start. Furthermore it only allows for a scanning time (not window or interval). This architecture is instead based on events, that is, when one such scanning time is finished an event is created and the developer decides what should happen.

Overall, basic functionality for the protocol stack and Relay feature did not cause any issues, but for more advanced features all of these issues had to be taken care of, meaning that even if a generic design for state machines had been possible, every state machine port had to be adjusted for these cases.

Furthermore, to accommodate the different requirements in the different SDKs, as discussed in this section, signatures would have parameters to satisfy everyone, leading to in some cases having at least one that worked only as a dummy for a

certain SDK. While not in any way an interoperability issue, it could indeed be seen as smelly code, especially in regard to scalability.

6.3.2 Toolchains

TI, even though having support for IAR development environment, did not use its pure toolchain. Instead of using IAR implementation of standard library memory allocation functions, libraries provided by the SDK implemented its own memory allocation. This was done through the iCall interface and was crucial for how applications are created on the TI chip.

6.4 Memory Analysis

We here provide a quick overview of the memory usage on the different SoCs. Memory usage was measured through compiler generated map-files, which provides an analysis of read-only code and data, read-write data as well as assembly label placements. Cuts of the map-file outputs for the different platforms can be seen in Figures 6.1, 6.2 and 6.3. A summary of Flash Memory usage on the different devices can be seen in table 6.2. We can see that in all cases the direct memory

Table 6.2: Mesh Implementation Flash Memory Usage

SoC	Total Flash	Remaining Flash	Mesh Usage
nRF52832	512kB	438kB	23kB
BGM121	512kB	326kB	18kB
CC2650	128kB	81kB	16kB

usage due to our implementation is quite low, in relation to total amount available and unused memory this is especially true for the nRF52832 and BGM121 SoCs. Even though the mesh solution uses the least amount of memory on the CC2650 SoC, this amount still accounts for 12.5% of the total flash memory of the chip. Had all of this memory been freed for applications the amount of remaining flash available would increase with almost 20%.

For the more limited memory, the SRAM, a summary of usage can be seen in table 6.3. For all SoCs the implementation uses quite little memory. It is however noteworthy to mention that on a limited SoC, such as CC2650, as little as 3kB still takes up a lot of the total amount. This amount is 15% of the total SRAM, and freeing it for application would result in an increase of a staggering 37.5%.

The results of this overview promotes a few notes to the overall discussion of the results. In the case of BGM121 it can be seen that the remaining SRAM, as well as the indirect SDK impact on our implementation's SRAM and Flash memory usage seems to make the Silicon Labs SDK very efficient. It indeed comes with

```
Module                              ro code  ro data  rw data
------                              -------  -------  -------
  :
heap_1.o                                116            16 392
  :
-------------------------------------------------------------
Total:                               56 806    2 663   19 401
  :
Linker created                                    48   18 852
-----------------------------------------------------------------
Grand Total:                         70 500    2 770   39 029
```

Figure 6.1: Memory Usage in nRF52832 SoC

```
    Module                              ro code  ro data  rw data
    ------                              -------  -------  -------
C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v1.1\protocol\
    binbootloader.o                                  796
    binstack.o                                   118 319
    -------------------------------------------------------------
    Total:                                       119 115
      :
    -------------------------------------------------------------
    Total:                               29 952    3 436    5 838
      :
    Linker created                                15 620    4 568
-----------------------------------------------------------------
    Grand Total:                         47 210  139 179   10 956
```

Figure 6.2: Memory Usage in BGM121 SoC

```
    Gaps                                     47       10        4
    Linker created                          528      324    1 432
--------------------------------------------------------------------------
    Grand Total:                         44 165    3 139   12 406       64
```

Figure 6.3: Memory Usage in CC2650 SoC

the disadvantage of no RTOS support, and should an RTOS have been used the memory usage might have spiked in comparison. It is possible to notice from analyzing nRF52832 SRAM usage in Figure 6.1 that the FreeRTOS named heap_1.o amounts for an incredible large amount of the SRAM usage.

For the CC2650 SoC memory is indeed an issue, and a generic solution requires common ground which will require additional memory usage. Depending on the application needed on the CC2650 it can be questionable whether or not a generic solution actually is a feasible idea, a specific TI solution would most probably have a much greater potential of minimizing SRAM drainage. Had a newer TI chip been surveyed memory had probably not been an issue at all.

Table 6.3: Mesh Implementation SRAM Usage

SoC	Total SRAM	Remaining SRAM	Mesh Usage
nRF52832	64kB	25kB	4kB
BGM121	64kB	53kB	0.3kB
CC2650	20kB	8kB	3kB

6.5 RTOS

As was recommended by [5] an RTOS is an important part of creating viable software to be used on different platforms. In this thesis the surveyed platforms did not at all offer a common RTOS, in fact Silicon Labs did not yet support any RTOS at all. As can be concluded from the design analysis and the implementation results an RTOS as a common ground seems to have been able to go a long way to create a truly interoperable solution. Due to the importance of an RTOS, and with a future release of Micrium-RTOS and FreeRTOS for Silicon Labs, a short analysis of the differences between them and how to bridge the gap of their designs is presented here.

FreeRTOS Tasks require only stack-depth as an argument, and will take care of allocating and deallocating this heap-memory itself. This stands in contrast to TI-RTOS, which is designed for every task to receive stack-allocated memory. The effects of these two methods are:

- Heap-allocated Task Stack
  - Simplicity in creating the thread from anywhere. Does not need global storage duration memory allocated in e.g. a main-function file.
  - Termination of the task must be controlled.
- Stack-allocated Task Stack
  - Termination control can be relaxed. Tasks callback-function may terminate naturally.
  - Memory must be pre-allocated and stored in a context that lives at least as long as the task.

Finally, the callback functions for all the different RTOS examined had different function signatures. These could of course be hidden and preconfigured for the mesh solution, but since a single RTOS could not (and will not in a foreseeable future) be applied to all platforms, caution must still be taken by the developer, as behavior for the different platforms differs. This means that even if a single design were possible, some adjustments of behavior would apply to the developer depending on application.

6.6 Provisioning

Although the lack of implementation of the provisioning part some conclusions can be drawn regarding the provisioning from the other aspects of the protocol implementation. The provisioning would, just as the Friend and LP nodes, have required timer functions in order to fulfill a proper implementation. This would lead to interoperability issues in the provisioning component just like those observed in the LP and Friend components. A separation of the stack and functionality could have been done in the same manner as was done in chapter 5 in order to make the provisioning stack, presented in section 2.3.4, interoperable.

None of the surveyed SoCs would have been suitable to be a provisioner since it is supposed to be a more powerful device such as a smart phone. Since such a device is not constrained in the same way by either specific SDK design nor hardware capabilities it is much easier to create an interoperable solution. Due to specification leaving it up to the developer to decide exactly how provisioning is done (e.g. through an intuitive drag-and-drop smart phone app that anyone can use, or through expert based configuration) there is also less value to analyze this for the scope of this thesis.

A final note on the provisioning is that the provisioning in a fully implemented Bluetooth Mesh network also effects the other nodes in the network. The Foundation Models (section 2.3.1) would handle the nodes provisioning and other highly necessary network functionalities. To add this onto the suggested solution would certainly bring further problems, not in the least in form of memory management. As this already is an issue to some degree for the TI SoC the provisioning part would indeed increase issues.

6.7 Scalability

The small discrepancies in each SDK certainly affected design choices as more features needed to be implemented. The missing common ground in form of similar event handlers and/or RTOS availability on all platforms forced solutions to reinvent some design patterns across the porting process. Individually the solutions on each platform might be scalable, but across them as a group it is likely that as more complex functionality is added, especially such referring to the issues discussed in this chapter, the amounts of re-invention required increases. This could clearly be seen from chapter 5, where whenever a more advanced feature was to be added the discrepancies of the SDKs grew more serious and forced more compromises, only to in the end result in re-inventions. Along with the results seen in 6.3. These differences are not trivial and arises from several levels of the SDKs, it is only fair to assume that should the Mesh protocol be extended, as is expected, scaling of the implementation is in a poor state, at least in the feature components.

Chapter 7

Conclusions

This chapter will contain some final thoughts and conclusions about the project, how to further the project, and Bluetooth Mesh's future prospects in general.

7.1 Discussion

It was presented in chapter 2 that Bluetooth Mesh nodes can have a variety of different functionalities (features), some of which can be turned on or off. A component based architecture proved to be appropriate when implementing the protocol since this easily allows tweaks and improvements in features without jeopardizing the underlying stack. However, it did not prove to be very interoperable. Every surveyed SoC had different ways of handling a BLE implementation and in general they had quite different SDK structures. This might not come as a big surprise since every hardware is specialized according to what is deemed to be most important to the manufacturer.

As can be seen from the results (chapter 6), several components needed reimplementation due to the large discrepancy in the design. Solutions for all platforms was possible to develop, so we can conclude a solution is portable, but with large amounts of work, and indeed not completely interoperable. Even if a generic solution should be possible to implement in a wider sense, issues discussed across this thesis, such as RTOS, would still provide a large obstacle to form a seamlessly interoperable solution. Each different RTOS needs to be understood, even when wrapped and hidden in some API. Furthermore, as explained, even if this could be applied to all platforms the same amount of work is still needed for each SDK as if an ordinary BLE device was to be implemented. Because of this, we cannot promote a great deal of benefits for a generic solution in the surveyed cases. However, if a common ground would have existed across all platforms, such as suggested by [5] among others, then the value and potential of such a solution would be huge. This is were we come to the same conclusion as [4], the importance of learning how to design software in such a way that they become abstractable into interoperable components could be among the most important aspects of a developers education.

It is therefor our conclusion that, based on the resources used in this thesis, it is not very feasible to try to create a general solution to apply on several differ-

ent constrained SoC/SDK pairs, not unless the mentioned common ground exists. Even if it does exists, designs could still have too large discrepancy to find an effective solution, and the SoCs might not be as efficient as if the solutions are tailored to each SoC. Since one of the key features is to bring BLE devices into a mesh network, this efficiency problem is non-trivial. Memory is also an issue when it comes to creating generic and component based software; some overhead is unavoidable and a SoC that is manufactured with LE in mind does not always have a great amount of memory available for the developer. The TI CC2650 is a good example of this where memory usage was high in relation to total amount. Depending on the complexity of an overlaying application on the CC2650 it could possibly prove not to be feasible at all to use the suggested solution.

If reminding oneself of the designs possible and discussed in this thesis, then even if the Nordic Semiconductor SDK did provide a non-blocking event handler, allowing a state machine to be placed just after the event handling switch, this design would still make the lowest common denominator quite low. At some point an interoperable solution becomes too smelly, ugly, non-readable, and too large. Just getting things to work is not the only interesting thing, as the reason for such a solution is in large part to require less time in understanding and studying several different SDKs. If a single solution becomes so complex that for someone new to enter the project requires more time to understand the software the whole point of the solution fails.

7.2 Future Work

There are several steps that could be taken in order to further this project. Adding the provisioning and more complex models would be two obvious improvements. However, the authors do not deem this to be the best way to further the use of this new technology. In the final months of this project a Bluetooth Mesh solution was added to both the Nordic Semiconductor SDK as well as Silicon Labs SDK. This in of itself does not invalidate the utility of the goal of this project since a Bluetooth Mesh solution that is interoperable could be applied to SoCs that do not get a software update with Bluetooth Mesh. A solution that is built by application developers could also be highly specialized which could be beneficial. As discussed in section 7.1, more problems than benefits have been observed though and it seems that more work on this path would not be productive. Instead the authors believe that surveying interoperability between different SoCs and SDKs when a known common ground does exist could be of great value. As have been discussed the only issues that arose was not only due the lack of thing such as a common RTOS, and it would thus be of great interest to survey how much it actually does affect. Had all our SoCs had a FreeRTOS port, would our conclusion have been different? Since FreeRTOS supports all of the processors for the surveyed SoCs it would be a matter of creating the FreeRTOS ports, which itself requires some time.

Another survey that could yield valuable results would be to investigate if and with how much work it would be possible to create a common event handler that

allows components to be completely abstractable. Constraints foiling this could be memory usage and SDK functionality locked away from the developer.

With that said, the future of Bluetooth Mesh seems to be a great improvement in the IoT field. It is easy to envision a future where Bluetooth Mesh networks are set up in every home. The already enormous reach of the Bluetooth technology makes it possible to be the defining universal IoT standard, but the field is in constant state of advancement so to speculate about the future in the long run is very hard to do with such a rapidly moving technology. One thing to be certain about is that Bluetooth Mesh will certainly be big and will be the next big step in IoT technology. Only time will tell how long it can hold its position as front runner since the next big thing might be right around the corner.

References

[1] J. Rivera and R. Meulen, *Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020*, 2013, `http://www.gartner.com/newsroom/id/2636073`, |Online; accessed 11-Sep-2017|

[2] Bluetooth Special Interest Group, *Our History*, `https://www.bluetooth.com/about-us/our-history`, |Online; accessed 11-Sep-2017|

[3] B. Sedov, A. Syschikov, and V. Ivanova, "Technology and Design Tools for Portable Software Development for Embedded Systems", in the *Proceedings of 16th Conference of Open Innovations Association FRUCT*, 978-1-4799-6226-6, pp. 86-93, Oulu, Finland, Oct 2014

[4] N. Medvidovic, R. F. Gamble, and D. S. Rosenblum, "Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms" in *Fourth International Software Architecture Workshop*, pp 77-83, Limerick, Ireland, June 2000

[5] G. Obiltschnig, *Designing and Building Portable Systems in C++*, Applied Informatics, 2007

[6] B. Govindarajalu, *Embedded Device Architecture Software Logic and Memory in SoCs*, 2014

[7] G. Fabron, *Bluetooth Technology 101*, 2016, `http://www.tomshardware.com/reviews/bluetooth-technology-101,4464.html#p1`, |Online; accessed 11-Sep-2017|

[8] Bluetooth Special Interest Group, *Bluetooth Specification Version 4.2*, 2014

[9] Bluetooth Special Interest Group, *Mesh Bluetooth Specification Version 0.9*, 2016

[10] Bluetooth Special Interest Group, *Mesh Model Specification Version 0.9*, 2016

[11] Texas Instruments, *CC2640 and CC2650 SimpleLink Bluetooth low energy Software Stack 2.2.1 Developer's Guide*, 2016

[12] Silicon Labs, *UG136: Silicon Labs Bluetooth C Application Developer's Guide*

[13] Nordic Semiconductors, *SoftDevice SpecificationS132 SoftDevice v3.0*, 2016

[14] *Texas Instruments support forum.* URL: https://e2e.ti.com/support/wireless_connectivity/bluetooth_low_energy/f/538/t/545493 [Online; accessed 9-Oct-2017]

Appendix A

SDK Signatures

The following are listings from the SDKs used for the nRF52832, BGM121, and CC2650 SoC:

```
typedef void(*ble_evt_handler_t)(ble_evt_t *p_ble_evt)

/**@brief Function for registering for BLE events.
 *
 * @details The application should use this function to register
 * for receiving BLE events from the SoftDevice. If the application
 * does not call this function, then any BLE event that may be
 * generated by the SoftDevice will NOT be fetched. Once the
 * application has registered for the events, it is not possible to
 * cancel the registration. However, it is possible to register a
 * different function for handling the events at any point of time.
 *
 * @param[in] ble_evt_handler Function to be called for each
 * received BLE event.
 *
 * @retval NRF_SUCCESS Successful registration.
 * @retval NRF_ERROR_NULL Null pointer provided as input.
 */
uint32_t softdevice_ble_evt_handler_set(ble_evt_handler_t
    ↪ ble_evt_handler);

/*
 * Blocks until event_queue is non-empty, then executes callback
 * function registered with softdevice_ble_evt_handler_set.
*/
uint32_t intern_softidevice_events_execute();
```

Listing A.1: Nordic Semiconductor SDK Signatures

```
/**
 * Blocks until new event arrives which requires processing by user
 * application.
```

```
 *
 * @return pointer to received event
 */
struct gecko_cmd_packet* gecko_wait_event(void);

/**
 * Same as gecko_wait_event but does not block if no events
 * waiting, instead returns NULL
 *
 * @return pointer to received event or NULL if no event waiting
 */
struct gecko_cmd_packet* gecko_peek_event(void);
```

Listing A.2: Silicon Labs SDK Signatures

```
/**
 * Registers an application.
 * Note that this function must be called from the thread
 * from which ICall_wait() function will be called.
 *
 * @param entity pointer to a variable to store entity id assigned
 * to the application.
 * @param msgsem pointer to a variable to store the synchronous
     ↪ object handle
 * associated with the calling thread.
 * @return @ref ICALL_ERRNO_SUCCESS when successful.<br>
 * @ref ICALL_ERRNO_NO_RESOURCE when ran out of resource.
 */
static ICall_Errno ICall_registerApp(ICall_EntityID *entity,
                                        ICall_SyncHandle *msgSyncHdl
                                            ↪ );

/**
 * Waits for a signal to the semaphore associated with the calling
     ↪ thread.
 *
 * Note that the semaphore associated with a thread is signaled
 * when a message is queued to the message receive queue of the
     ↪ thread
 * or when ICall_signal() function is called onto the semaphore.
 *
 * @param milliseconds timeout period in milliseconds.
 * @return @ref ICALL_ERRNO_SUCCESS when the semaphore is signaled
     ↪ .<br>
 * @ref ICALL_ERRNO_TIMEOUT when designated timeout period
 * has passed since the call of the function without
 * the semaphore being signaled.
 */
```

```
static ICall_Errno ICall_wait(uint_fast32_t milliseconds);


/**
 * Retrieves a message received at the message queue
 * associated with the calling thread.
 *
 * Note that this function should be used by an application
 * which does not expect any message from non-server entity.
 *
 * @param src pointer to a variable to store the service id
 * of the registered server which sent the retrieved
 * message
 * @param dest pointer to a variable to store the entity id
 * of the destination of the message.
 * @param msg pointer to a pointer variable to store the
 * starting address of the message body being
 * retrieved.
 * @return @ref ICALL_ERRNO_SUCCESS when the operation was
     ↪ successful
 * and a message was retrieved.<br>
 * @ref ICALL_ERRNO_NOMSG when there is no queued message
 * at the moment.<br>
 * @ref ICALL_ERRNO_CORRUPT_MSG when a message queued in
 * front of the thread's receive queue was not sent by
 * a server. Note that in this case, the message is
 * not retrieved but thrown away.<br>
 * @ref ICALL_ERRNO_UNKNOWN_THREAD when this function is
 * called from a thread which has not registered
 * an entity, either through ICall_enrollService()
 * or through ICall_registerApp().
 */
ICall_Errno
ICall_fetchServiceMsg(ICall_ServiceEnum *src,
                      ICall_EntityID *dest,
                      void **msg);
```

Listing A.3: TI SDK Signatures