Thomas Fischer BSc

Design and Implementation of a Secure Personal Assistant Device with BLE and NFC

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.-Prof. Dipl.-Ing. Dr.techn. Christian Steger
Institute for Technical Informatics

Advisor

Dipl.-Ing. Mihai Tudosie
Infineon Technologies AG

Graz, April 2016

AFFIDAVIT

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

....

Date Signature

Abstract

This thesis describes the design and implementation of a secure personal assistant device which is capable of communicating with an Android smartphone using the Bluetooth Low Energy (BLE) and Near Field Communication (NFC) interfaces. The device can be used as mobile temperature sensor station, allowing an user to view sensor values, transmitted over a mid-range distance via the BLE interface to the smartphone. Quick and user friendly device paring is supported using the NFC interface.

The document starts with an introduction to the wireless interfaces BLE and NFC. An overview of the state of the art technologies used in security controllers is provided. The next chapter is dedicated to the design of the hardware and software. Software components include a Bluetooth Low Energy software stack and a client for the Transport Layer Security (TLS) protocol to allow secure communication with the device. An Android App is designed to demonstrate communication with the personal assistant device. The implementation chapter starts with an overview of the development flow, describes important aspects of the implementation and concludes with the testing of the hardware and software. Finally a conclusion is drawn, summarizing the most important results.

Kurzfassung

Diese Masterarbeit beschreibt den Entwurf und Implementierung eines persönlichen Assistenzgeräts, welches in der Lage ist, mit einem Android Smartphone über ein Near Field Communication (NFC) und Bluetooth Low Energy (BLE) Interface zu kommunizieren. Das Gerät kann als mobile Sensorstation benutzt werden, welche dem Benutzer erlaubt, Sensorwerte, welche über eine mittlere Distanz über das BLE Interface übertragen werden, auf einer Smartphone App anzuzeigen. Einfache benutzerfreundliche Kopplung der Geräte wird über das NFC Interface unterstützt.

Das Dokument beginnt mit einer Einführung in die Interfaces BLE und NFC. Es folgt ein Überblick über den aktuellen Stand der Technik in Security Controllern. Das nächste Kapitel widmet sich der sich dem Design der Hardware und Software. Zu den Software Komponenten gehört ein Bluetooth Low Energy Software Stack und ein Client für das Transport Layer Security (TLS) Protokoll, welcher sichere Kommunikation mit dem Gerät erlaubt. Es wird eine Android App entworfen, welche Kommunikation mit dem Gerät demonstriert. Das Implementierungskapitel beginnt mit einer Beschreibung des Entwicklungsflusses, gefolgt von wichtigen Aspekten der Implementierung und schließt mit dem Testen der Hardware und Software ab. Am Ende wird eine Schlussfolgerung gezogen, welche die wichtigsten Resultate zusammenfasst.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The most commonly used wirelss technologies include Bluetooth and NFC. Most smartphones contain a module for Bluetooth and many support NFC. The Bluetooth Low Energy (BLE) standard is interesting because it allows devices to be operated with a small coint cell battery for many years. Providing a larger transmission range than NFC it allows to support many interesting use cases as described in the following chapter. For a student of computer science it is interesting to understand how the software contained in a BLE device is designed and implemented. By studying the standard many questions about the design are answered but only after implementing the software in person a full-blown understanding is acquired. The ideal goal would be to implement the hardware and software of a whole device by oneself but this seems impossible considering the complexity of such a system. So the focus will be on the software which is related to the wireless standards Bluetooth and NFC.

Since all data transfered is send over the air it is important to provide means to protect the data from eavesdropping, beeing altered or faked by others. The key to solve this problem is called cryptography which describes how to provide confidentiality, authentication and integrity. The most important standard used is called the Transport Layer Security (TLS) protocol which is used in the world wide web. Every internet banking application uses TLS to secure the communication between the web browser and the server. By not just implementing this standard but also focusing on impementation specific security aspects much knowledge can be aquired.

Combining the topics wireless technologies and information security leads to an interesting project an master thesis which should be worthwhile reading.

1.2 Overview

Infineon Technologies AG is a provider of secure microcontrollers which are suitable for cryptographic functions and storing confident information. These controllers provide certain protection mechanisms against logical and physical attacks to prevent extraction of the stored information. During this thesis options to integrate a secure element into an application which requires the capabilities of a security controller and the BLE interface are evaluated.

This thesis will guide through the necessary steps to design and implement a personal assistant device to demonstrate the capabilities of an Infineon security controller to act as application controller in a device which provides security related services combined with wireless interfaces (BLE and Near Field Communication (NFC)) for communication. One aspect of this thesis is the implementation of a BLE software stack to control a BLE transceiver as well as implementing other interface drivers like Inter-Integrated Circuit (I2C) to access external sensors and NFC to allow configuration of the device.

The designed device will be capable to serve in two different usage scenarios. In the first use case the device will be used as external sensor station. The device can be configured via an App on an Android smartphone over the NFC interface. Once the device is activated, the user can view sensor values on the App which are continuously transfered via a BLE link. Security is provided in the sense that the link is authenticated and encrypted which implies that no external listener should be able to intercept and decrypt the sensors values. The second use case is about using the device as a door watchdog which periodically sends authenticated and encrypted messages to the smartphone which triggers an alarm if the door is opened or the signal is lost. There are countless other possible use cases which can easily be served by slightly modifying the device. The focus will be on the implementation of the interface drivers (I2C, NFC, and BLE) and the cryptographic services which provide authentication, confidentiality and integrity of the data transfered over the air.

In the end the constructed device shall be evaluated and possible optimizations shall be discussed.

Chapter 2

State of the art

This chapter will give an introduction to the Bluetooth and NFC standards and their applications. The available technologies used in security controllers will be summarized and possible attacks and countermeasures discussed.

2.1 Bluetooth Low Energy

There are two good books which can help to acquire an understanding of the Bluetooth technology, the first [18] gives a general overview of the Bluetooth stack and the second [19] is a guide intended for system engineers who are implementing the standard. More detailed information is available in the Bluetooth standard [3].

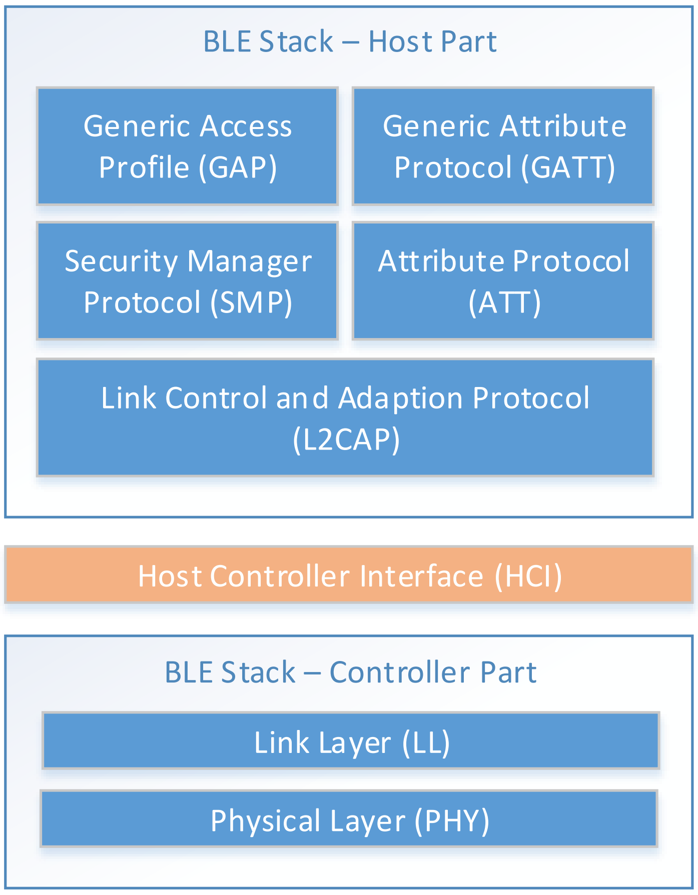
Figure 2.1 gives an overview of the BLE stack. The stack is divided into two main parts. The host and the controller part. This separation is important because it affects possible system architectures.

Bluetooth Low Energy is a completely new standard which is working different than classic Bluetooth. The physical transport uses the 2.4 Ghz frequency spectrum for is intended for industrial, scientific and medical applications (ISM band). The used spectrum is divided in 40 channels, where three of them are used as so called advertising channels. When a device wants to broadcast information it uses these three channels.

The link layer is placed on top of this physical transport. It has the purpose to create logical connections between devices and provide these channels to higher layers. Three basic modes of operation are supported by the link layer. The first mode is the advertising mode. In this mode the device broadcasts advertising packets on one of the three advertising channels. This allows unidirectional communication with other devices. The second mode of operation is the scanning mode. In this mode the device listens for broadcast packets on the advertising channels. Besides from receiving the data in the advertising packets this also has the purpose to find devices which are suitable for a bidirectional connection. This leads to the third mode of operation, the connection mode. If a device sends special advertising packets indicating it supports a connection it is possible to create a link layer connection with this device.

Once the link layer connection is established between two devices the Link Control and Adaption layer (L2CAP) is providing service multiplexing on top of this link. This means that data of multiple protocols can be send over one link layer connection and the L2CAP takes care of differentiating them.

On top of L2CAP there are usually at least to protocols and layers supported. One layer above is the Generic Access Profile (GAP) which responsible to provide a user interface to allow handling connections between devices. It uses the Security Manager Protocol (SMP) to perform secure pairing between devices. The second layer above is the Generic Attribute Profile (GATT) which is basically a structured database containing the information the device want to provide to other devices. This stored information can be accessed using the Attribute Protocol (ATT).



BLE Stack – Host Part
Generic Access Profile (GAP)
Generic Attribute Protocol (GATT)
Security Manager Protocol (SMP)
Attribute Protocol (ATT)
Link Control and Adaption Protocol (L2CAP)
Host Controller Interface (HCI)
BLE Stack – Controller Part
Link Layer (LL)
Physical Layer (PHY)

Figure 2.1: Bluetooth Low Energy Stack

2.1.1 Applications of BLE

2.1.1.1 Mobile sensor stations

Bluetooth is integrated in many health care devices. In [21] many example applications in that area are described like a heart rate sensor or a finger pulse oximeter. A system architecture based on the products from Nordic Semiconductor and Texas Instruments is proposed. Both companies provide Micro Controller Unit (MCU) packages containing a whole Bluetooth solution.

It is very common to user smartphones based on Android and iOS (Apple) to extract data from the devices as seen in [24]. This paper contains the system architecture of a blood pressure monitor which is read out using an iPhone app.

All these examples demonstrate the capability to operate a Bluetooth Low Energy device using small batteries over a long period of time. This low energy consumption is a big advantage compared to WiFi/WLAN. The disadvantage is obviously the lower data transmission rate.

The system architecture of these devices is similar to the personal assistant device. Both act as mobile sensor station (e.g. heart rate sensor) and contain a micro controller with a Bluetooth subsystem to transmit the data to a smartphone. The difference lies mainly in the provided security level. The personal assistant device in implemented in a high security controller with protection against active and passive attacks.

2.1.1.2 BLE beacons

One common type of application is using a BLE device as a beacon. This means only a transmitter is required and no receiver which reduces the costs of production. These beacons can broadcast arbitrary data which are application depended. Besides from that it is possible to approximate the distance between a beacon and a receiving device by analyzing the Received Signal Strength Indicator (RSSI). These two capabilities are combined to use BLE in public transportation applications [17].

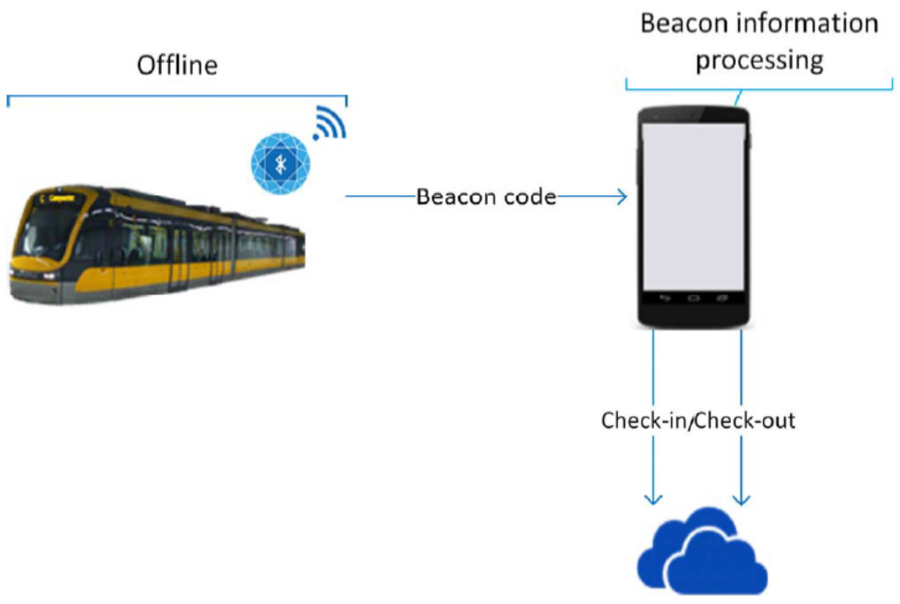


Figure 2.2: BLE Beacon in Public Transportation (taken from [17])

Figure 2.2 demonstrates the setup when using a BLE beacon for ticket management in public transportation. A BLE beacon is placed in a train which periodically emits advertising packets containing an identification number of the used transport. The user possesses a smartphone with BLE capabilities and has an app installed which connects to the service provider. The user does not need to manually select which route is taken because the id number from the beacon is available as well as the RSSI. Using the RSSI the app can determine when the user leaves the train. The service provider can combine this acquired information on company servers with the time schedule and GPS position of the train to calculate a route. The user just has to install and setup the app once. After that the beacon signal can be used to wake up the app whenever needed. This leads to a very user-friendly approach where no interaction is required in daily use.

As stated in [17] the BLE interface is capable of providing sufficient reaction times which for this application if the BLE connection parameters (timing intervals) are chosen properly.

Another interesting application for BLE beacons is demonstrated in [20] where a student registration system at an university is implemented. Each student possess a student id card which is scanned by a NFC capable smartphone using an app. This scanned data together with a magic number received from a BLE token in class room is used to perform a registration. Since the magic number is only broadcasted in the class room only present students can register.

Apple's iBeacon is a well known example for BLE beacon devices [16]. Figure 2.3 shows how the proximity information can be used to provide the customers information about new products, track his location and enable user friendly payment when leaving through an exit way.

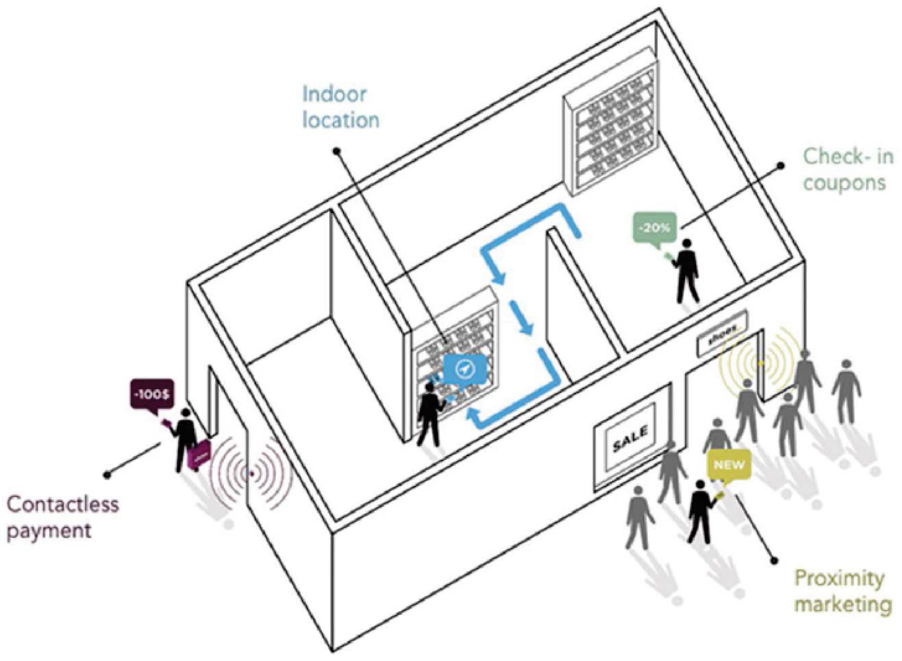


Figure 2.3: BLE Beacons in an Apple Store (taken from [16])

The personal assistant device to be created in this project can also be used as beacon which implies supporting many use cases where a beacons are used. The capabilities of the device to be designed exceed those of a simple beacon because a transceiver (on not just a transmitter) is integrated and the implemented BLE stack provides much more features like bidirectional connections.

2.1.1.3 IPv6 over BLE

Allowing BLE to carry Internet Protocol (IP) packets leads to interesting applications. Devices can be integrated in the Internet of Things (IoT) and top of this network encryption protocols like the Transport Layer Security (TLS) can be implemented. Figure 2.4 shows this can be build upon the basic layers of the Bluetooth stack. The Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) protocols are provided for the application and are using the Internet Prococol (IP) as transport. The IP protocol has a very large Maximum Transmission Unit (MTU) and therefore an adaption layer is necessary whose main purpose is data compression. A first implementation was done in [7]. This paper describes how the shown extension is implemented on top of the Linux Bluetooth stack BlueZ.

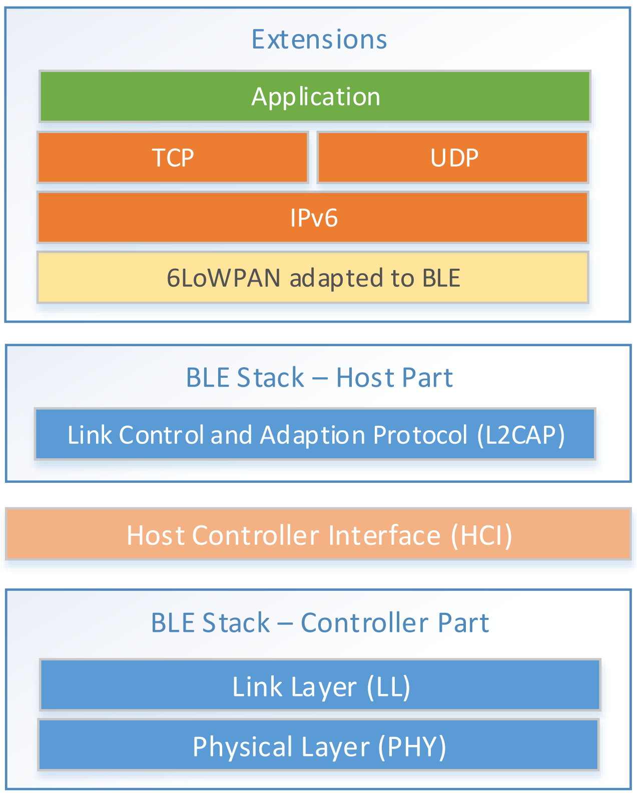


Figure 2.4: IPv6 over Bluetooth Low Energy

The advantage of implementing this standard is being compatible with the existing infrastructure. On the other hand the IP stack introduces a huge overhead regarding the size of the transmitted packets because additional large headers are required. The work of [6] suggest skipping the IP stack and replacing it with a lightweight more optimized protocol for this type of applications.

An application worthwhile to mention for IPv6 over BLE is audio data streaming as described in [11]. This use cases is also an interesting test case to monitor and measure the performance and energy consumption of a Bluetooth device because of the high transmission load.

2.1.2 Bluetooth Low Energy (BLE) Solutions

Multiple companies around the world provide BLE solutions which can be adapted for applications. The Bluetooth stack can be divided into two main parts. The host part which consists of the higher layers of the stack and the controller which contains the lower layers (physical and link layer). This leads to the possibility to split a Bluetooth solution into two physical separate parts. There are many different Bluetooth solutions on the market but all can be assigned to one of the two following categories.

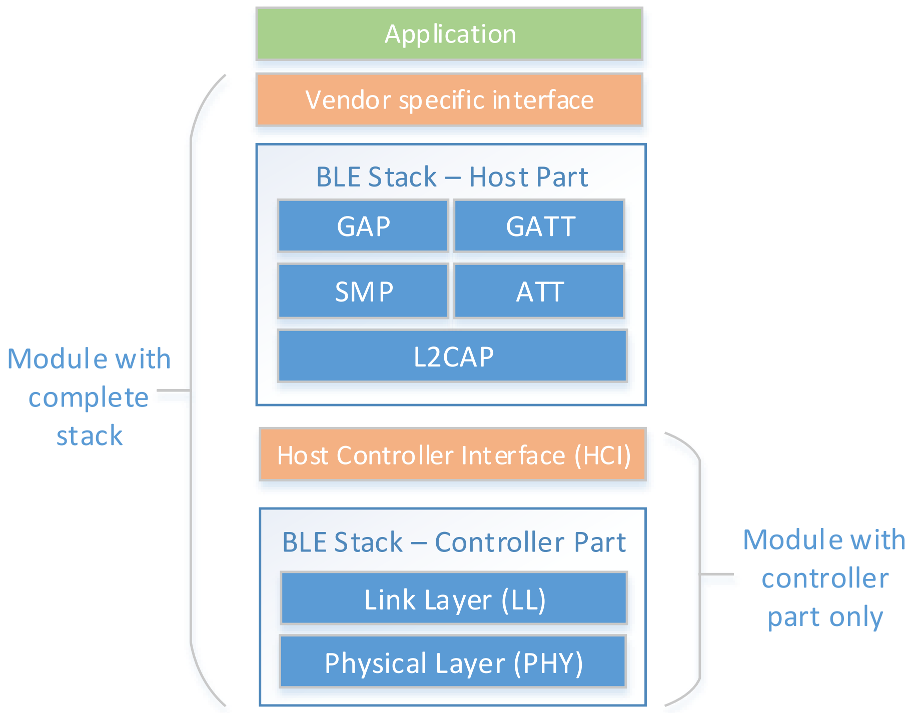

Application
Vendor specific interface
BLE Stack – Host Part
GAP
GATT
SMP
ATT
L2CAP
Host Controller Interface (HCI)
BLE Stack – Controller Part
Link Layer (LL)
Physical Layer (PHY)
Module with complete stack
Module with controller part only

Figure 2.5: Bluetooth module types

2.1.2.1 Modules with complete host and controller part

There are modules which contain the host and controller part of the Bluetooth stack. In this case typically a BLE stack is provided in form of a software library and the user can write application code which makes use of this library. The advantage of this solution is that the vendor supplies the whole Bluetooth stack and no extra implementation is necessary. Software is considered to be intellectual property of the corresponding company and is usually provided only in binary form to the customer. This makes adaptations and optimizations very hard and leads to an inflexible approach. For example if optimization regarding power consumption is required and the source code of the software is not available then only certain parameters can be adapted. For many applications this flexibility may be enough but from an academic point of view it is desirable to get more insight and understand the used technologies.

2.1.2.2 Modules with controller part only

The second option is the use of a module which contains a BLE controller (transceiver) and provides an interface for an external application controller. For instance the transceiver could be connected via Universal Asynchronous Receiver Transmitter (UART) or Serial Peripheral Interface (SPI) interface to an application controller. In this case the transceiver runs only the lower (time-critical) layers of the BLE stack (which consist of the physical transport and the link layer). The functionality of the BLE Link Layer is logically exposed to the application controller as the standardized Host Controller Interface (HCI). The upper layers (often referred as host part) of the BLE stack must be implemented on the application controller.

2.1.2.3 Targeted solution

Currently there exists no Infineon micro-controller with a BLE interface so an external BLE module is required. For this project a module with only the controller part is chosen. A custom implementation of the host part of the BLE stack allows optimizations regarding the required processing power and memory consumption which also implies lower energy consumption. These higher layers do not have critical timing requirements which allows the execution of other software simultaneously on the chip.

2.2 Near Field Communication (NFC)

The Near Field Communication (NFC) standard is based on the Radio Frequency Interface (RFI) which is used to provide short range communication links. There are two functional types of devices, readers and cards. Readers are the active component, generating a electromagnetic field while the card is mostly passive. The typical range is around ten centimeters however this highly depends on the used hardware, antenna and transmission power. One big advantage compared to BLE is the capability to power a device over the electromagnetic field of the reader. The reader is sending data by modulating the field. Since the card draws power from the field it can modulate the power consumption and therefore also send data.

Infineon provides multiple chips containing an RFI module which can be used as NFC card. One example is the Infineon SLE70 chip which is used during this project as application controller.

According to [8] about 30 percent of the smartphones sold in 2015 contain a NFC chip. This is certainly below the distribution of BLE but this number is still growing.

2.2.1 Applications of NFC

Applications often intersect with BLE, for example in public transportations [8]. An NFC tag can be used as ticket to check-in at gateways. Similar use cases exist related to the hotel industry. Tags can be used to access buildings and for payment. One of the most prominent use cases is the integration of NFC in credit cards and banking cards. MasterCard and Visa are providing fast and user-friendly payments via NFC.

2.2.1.1 NFC used for Bluetooth device pairing

The Bluetooth standard supports device paring by using an out-of-band channel like NFC. Using this alternate channel it is possible to transmit high entropy data which can be used to authenticate devices. In this project we will use the NFC interface for device pairing considering the fact that NFC is a low range communication and that man-in-the-middle attacks are likely to be detected by the user. However additional security protocols are implemented on top of NFC to prevent eavesdropping and authentication will be supported.

2.3 Security controllers

The security of NFC capable systems is a popular research topic. The simplest form of attacks to a system are logical attacks. Weaknesses in the communication protocol and interface are used to extract information. An example of a logical attack would be the use of an interface in way it is not indented to use. The input of invalid data could cause unhanded errors which could eventually reveal secret information. This is probably the easiest way because this can be done remotely. Therefore it is mandatory to design secure interface drivers and protocols.

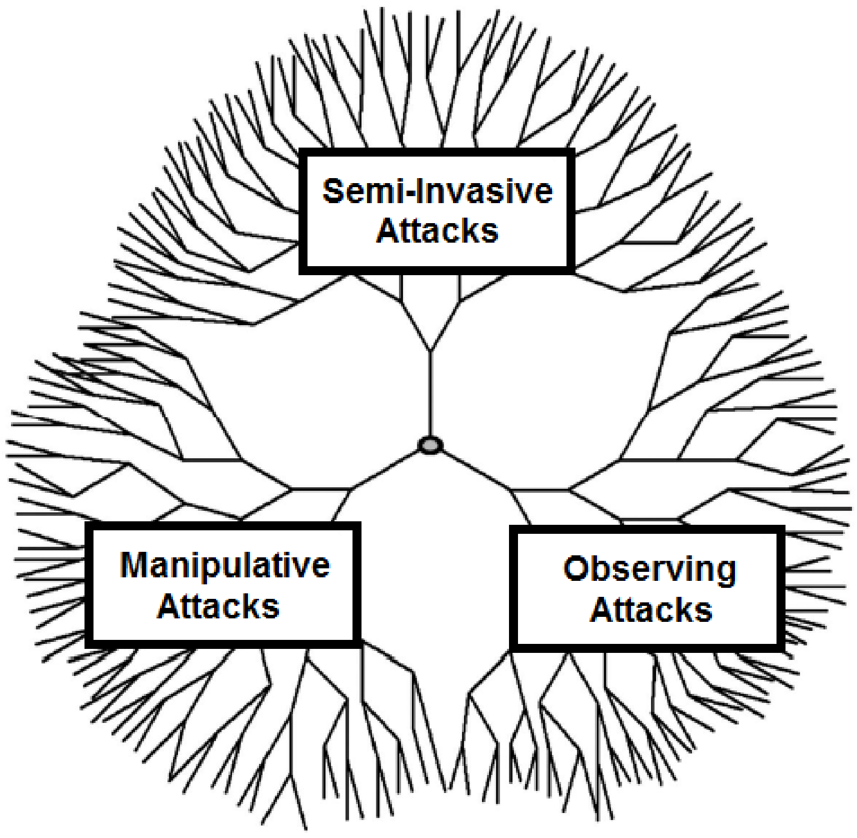


Figure 2.6: Attack tree (taken from [14])

2.3.1 Manipulative attacks

In [14] an overview of the most important types of attacks is presented. Manipulative attacks include all operations where the physical structure of the chip is altered. This can be done by performing micro-surgery using an focused-ion-beam (FIB) workstation. Also fine needles can be placed on signal lines to extract information. This category of attacks is the most complex to perform but it turns out cheaper attacks are often also sufficient to extract information from the chip.

2.3.2 Observing attacks

Observing attacks are passive attacks where the chip itself is not altered and only inputs and outputs are measured. Typically side-channel information like the power consumption or the execution time of a command are used to extract secret information from the device. For example the a straight-forward implementation of the RSA encryption algorithm using the square and multiply method would yield side channel timing information because the execution time of this algorithm depends on the key bits used. An example for an attack where the power side-channel is used it where a processor is used that shows different power consumption levels for different instructions. This allows mapping a measured power trace to the instructions executed. More advanced attacks in this category include differential power analysis (DPA) and electromagnetic analysis (EMA).

2.3.3 Semi-Invasive attacks

Semi-Invasive attacks are attacks where the functionality of a chip is disturbed by external influences like power spikes or light emissions. For example a PIN check of a NFC smartcard could be disturbed by a power spike on the supply voltage. Implementing counter-measures against one of these attacks is possible but might not protect against similar attacks. For example integrating a light sensor might detect against light emissions but not other emissions like radiation with alpha particles. An other example is the monitoring of the supply voltage of the chip to detect spikes. If the spike would be applied only locally to one module the detection might also fail. Electromagnetic induction attacks are an other example for attacks that can be applied locally to a certain module of the chip. Thermal attacks use the properties of components like memory to operate only in a certain temperature range. If these ranges are exceeded malfunctions happen and data is corrupted. If some memory modules are exposed to very low temperatures the memory starts to "freeze-in" even of the power supply is off. One advanced type of attack utilizes so called thermally induced voltage alteration (TIVA) devices which cause local irritation trough the backside of the chip with using a infrared laser. This type of light is not detected by light sensors which means more advanced protection is required.

2.3.4 Countermeasures

Facing all these types of possible attacks, security countermeasures must be implemented to protect the chip and its data. Common methods to protect the memory are parity bits, error correction codes (ECC) and mathematical error detection codes (EDC). The simplest form of parity bits only tells if the bit sum of a value is even or odd and will protect only in fifty percent of multi-bit errors against manipulations. The more advanced EDC method is standard in todays security controllers. The memory and bus must be protected against being read out by attackers after disassembling the chip. This problem can be solved by encrypting the contents of the memory and the data on the bus. It is also state of the art to work with encrypted data in the CPU, this is also often referred as full data path encryption. Detecting faults induced by an attacker can be done by implementing a dual-core CPU where both cores check each others results and compare them, resulting in the security reset if the results are different. At last it should be mentioned that if co-processors and other peripherals are used the data path protection must be extended to cover also these elements.

Chapter 3

Design

This chapter describes the use cases, requirements and the system architecture of the device to be constructed.

3.1 Use cases

There are many possible use cases for an embedded device with NFC and BLE capabilities. For demonstration purposes two applications are chosen.

3.1.1 Mobile temperature sensor station

The device should include a temperature sensor which does a measurement in a periodic time interval. The user should be able to view this data on smartphone using an App as shown in figure 3.1.

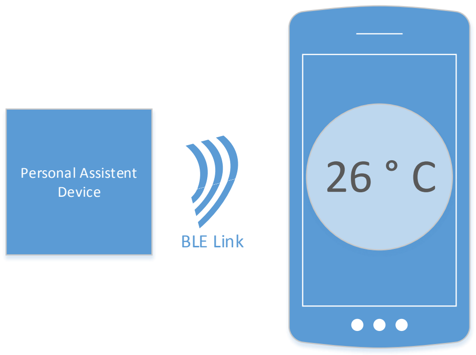


Figure 3.1: Use case mobile temperature sensor station

The usage procedure is as follows.

1. Place the device in a position of interest
2. Move a smartphone (with NFC interface) in close proximity of the device to activate it and perform pairing
3. Review the measurement values on the smartphone app
4. Stop the application on the phone

3.1.2 Door watchdog

By including a reed sensor in the device it is possible to place to device on a door frame and detect when the door is opened (see figure 3.2). The system sends encrypted, signed messages periodically to the smartphone as long as the door remains closed. Once the door opens or the device will stop sending these messages and the smartphone will activate an alarm. Should the device be deactivated for any reason the alarm will also be triggered.

The usage procedure is as follows.

1. Place the device on the door frame, mount a magnet on the door, the reed sensor must be in closed state
2. Use the phone with the NFC interface to activate the device
3. If the door is opened, an alarm will occur on the phone
4. Else stop the application on the phone

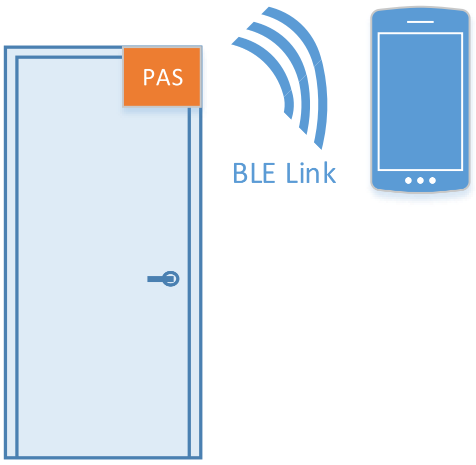


Figure 3.2: Use case 2: door watchdog

3.2 Requirements

The use cases lead to the following requirements.

- The device must provide an BLE interface to provide the acquired data.
- A temperature sensor must be integrated to perform measurements.
- A reed sensor must be integrated detect state changes of a door (or similar structure).
- Activation and pairing between the device and a smartphone should be possible using the NFC interface.

- The system should only be used by authorized users.
- Data transfered over the wireless interfaces should be protected. This implies data must be encrypted to provide confidentiality and message authentication codes must be used to ensure authenticity and integrity.

3.3 System overview

The personal assistant device to be constructed is an embedded system with no display or keyboard. These missing I/O capabilities are delegated to a smartphone based on Android which is used to display application data generated by the system. To achieve this two wireless interfaces are implemented, Bluetooth Low Energy (BLE) and Near Field Communication (NFC). The BLE interface is intended to carry application data over a medium range distances to be visualized by a smartphone app. The NFC interface is used perform pairing between the embedded system and the smartphone which includes configuring it and setting up the BLE connection.

Figure 3.3 presents the physical view of the system components including the interaction with a BLE and NFC capable smartphone.

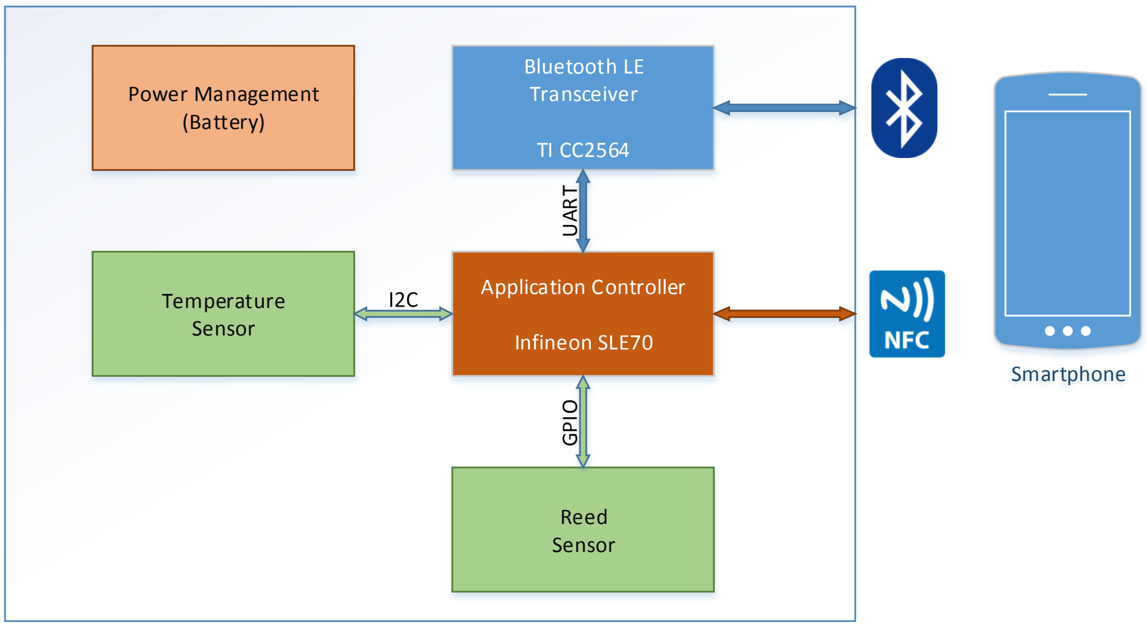


Figure 3.3: System overview

3.4 System components

3.4.1 Bluetooth Low Energy Transceiver

The BLE transceiver module is connected to the application via a 4-wire UART interface supporting hardware and software flow control. In this project the Texas Instruments CC2564 module has been chosen. It provides the standardized Host Controller Interface (HCI) as defined in the BLE specification.

3.4.2 Temperature sensor

A temperature sensor is connected via I2C interface to the application controller. The Atmel AT30TS74 device supports measurements in range from -55 up to +125 degrees with high accuracy. It offers a simple interface with five special function registers (see table 3.1) which can be accessed using a simple communication protocol.

Register	Address	Read/Write
Pointer Register	n/a	W
Temperature Register	0x00	R
Configuration Register	0x01	R/W
T_{LOW} Limit Register	0x02	R/W
T_{HIGH} Limit Register	0x03	R/W

Table 3.1: Atmel AT30TS74 Registers

The pointer register is used to select the current active register which can be written or read in the next command. The temperature register allows reading the last measured temperature value of the sensor. The configuration register allows to alter the resolution of the sensor values in bits and other properties. The two limit registers can be used the setup an interrupt on an alert line connected to the application controller. This feature can be enabled and disabled in the configuration register. Using this feature is is possible to detect temperature changes without actively polling data from the sensor. This can be used save energy and resources on the application controller because it can stay idle if no changes occur.

3.4.3 Reed sensor

A reed sensor is connected to a General Purpose Input Output (GPIO) pin of the application controller. This sensor is similar to a switch. By placing a magnet close to the sensor the state can be altered. The application controller is detecting voltage edge transitions on the corresponding GPIO pin. By setting up an interrupt service routine the state changes are propagated to the application.

3.4.4 Power Management

A battery allows powering the device in absence of an external power supply. Alternatively the device can be powered over an Universal Serial Bus (USB) cable or the NFC interface. The SLE70 chip is capable of operating in a dynamic voltage range. If the USB interface is used for flashing the software then a 5V supply is needed. Otherwise lower voltage levels like a 3V battery are sufficient. The TI CC2564 chip requires a voltage level around 3V. The 5V supply from the USB connector exceeds the maximum voltage ratings stated in the data sheet. This implies a voltage regulator is required if the device is powered with 5V from USB.

3.4.5 Application Controller

An Infineon SLE70 secure microcontroller is used as application controller. It runs a multi-tasking operating system capable of dealing with data from multiple interfaces concurrently. The NFC interface can serve for communication and as power source. Initially the device is deactivated and draws no energy from the battery. The user has to activate the NFC interface on the phone (card reader mode) and move the phone near the device. The electromagnetic field (from the phone) will then power the application controller within the device over the NFC antenna. Once the user is authenticated the device can be activated and the application started. This means the BLE module is powered on and transmission of application data begins. The power is then supplied by an battery in the device.

Figure 3.4 shows the development view consisting of the software modules in the SLE70 controller. The green modules are responsible of acquiring data from the sensors. The blue modules are related to Bluetooth and the orange to NFC. The application modules are described in the following paragraphs.

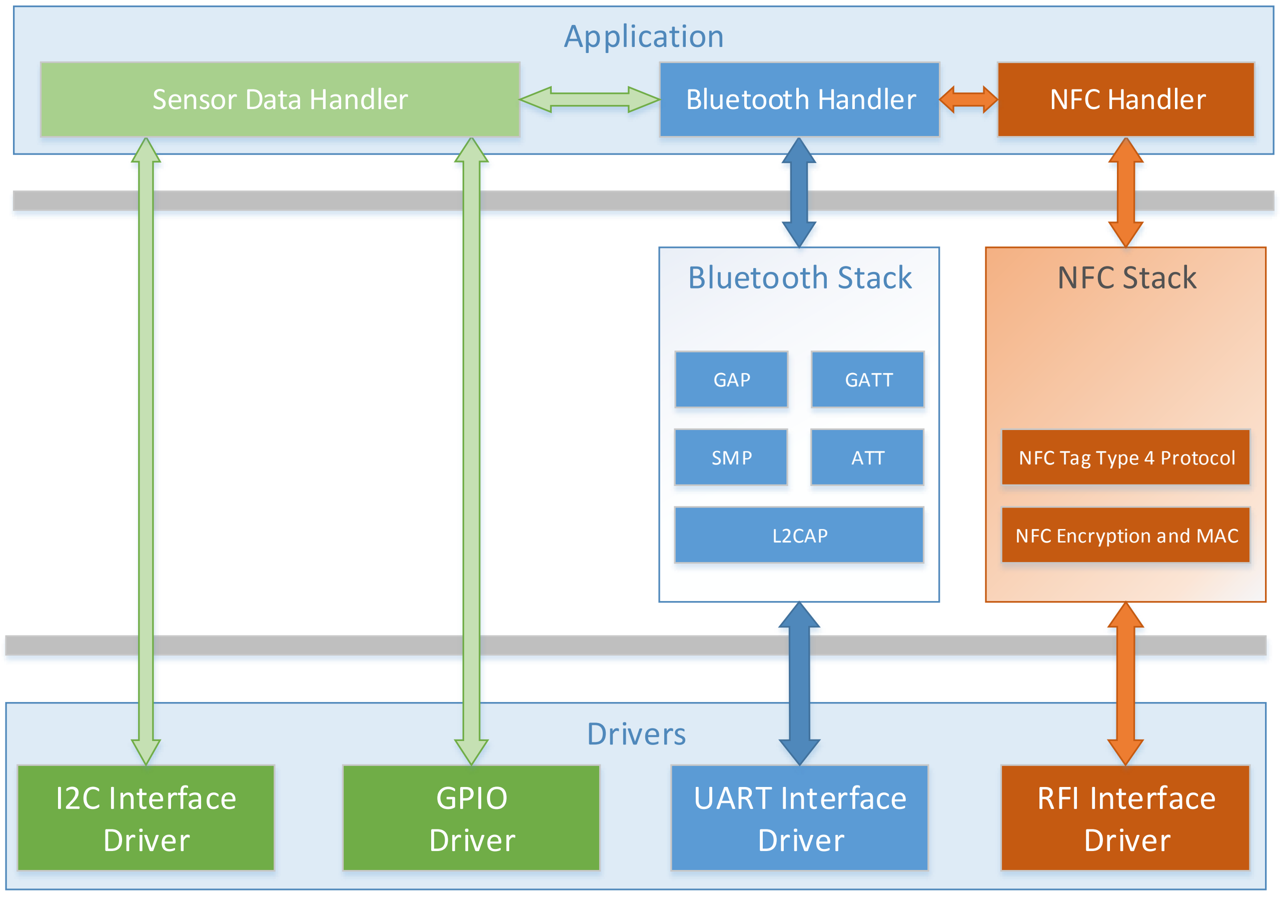


Figure 3.4: Development View - SLE70 application controller software

3.4.5.1 Application modules

The application modules are the represent the highest layer in figure 3.4. Each of them represents a task in the operating system of the application controller.

3.4.5.1.1 Sensor Data Handler

The Sensor Data Handler is a task which periodically acquires sensor readings and stores the values in memory. Whenever the sensor values changes the new values are propagated to the Bluetooth Handler. Values from the temperature sensor are acquired by using the I2C driver and the state of the reed sensor is checked using the GPIO driver.

3.4.5.1.2 Bluetooth Handler

The Bluetooth Handler has the purpose to initialize the Bluetooth stack and process events received from the it. In addition it provides an interface for the Sensor Data Handler and NFC Handler.

3.4.5.1.3 NFC Handler

The NFC Handler task has the purpose to process commands coming from the NFC interface. It contains a command interpreter which checks the format of each command and executes it afterwards. The supported commands include the following actions.

- Enable/Disable the BLE interface
- Activate the BLE advertising
- Retrieve a randomly generated key for BLE pairing

3.4.5.2 Protocol stacks

The protocol stacks represent the middle layer in figure 3.4. The design includes a stack for BLE and NFC.

3.4.5.2.1 BLE stack

The BLE stack contains all layers mandatory by the Bluetooth standards which includes the Link Control and Adaption Protocol (L2CAP), Security Manager Protocol (SMP), Generic Access Profile (GAP), Attribute Protocol (ATT) and Generic Attribute Profile (GATT). The functionality and interfaces of the stack is well defined in the standard with one exception. Designing the interface to the application or user is left to the implementating engineer. The following paragraphs will summarize the interface functions required.

3.4.5.2.1.1 L2CAP interface

The L2CAP layer is the lowest layer of the host part of the stack and little interaction with the user is required except the creation of the L2CAP LE Credit Based Channels. The following functions are needed.

- Create/Close a L2CAP LE Credit Based Channel (channel id as parameter)
- Transmit an L2CAP packet (connection handle and packet data pointer as parameter)

3.4.5.2.1.2 GAP interface to set controller properties

An interface (see the next code listening) is provided to allow the user or the application to read and change device specific properties. In addition functions are provided to reset the controller, configure the received events (by setting event masks) and alter vendor specific properties like the transmission power.

```
UINT16 gap_address_get(UINT8* bluetooth_address);
UINT16 gap_address_set(UINT8* bluetooth_address);
UINT16 gap_device_name_get(UINT8* device_name,
                           UINT16 max_length);
UINT16 gap_device_name_set(UINT8* device_name);
UINT16 gap_passkey_get(UINT8* passkey, UINT16 max_length);
UINT16 gap_passkey_set(UINT8* passkey, UINT16 length);
```

Bluetooth Device Address

A 6 byte value identifying a Bluetooth device. This value should be unique. The functions gap_address_get and gap_address_set are used to manage this property.

Bluetooth Device Name

A character string intended to give the device a user friendly name which is displayed in the user interface when discovering devices. The length of this value is limited to 248 bytes. If Unicode characters are used which are encoded using more then one byte then the maximum string length might be less. The functions gap_device_name_get and gap_device_name_set are used to manage this property.

Bluetooth Passkey (PIN)

An alpha numerical value which can be used to derive keys used in the Security Manager. This value might be pre-shared, entered by the user or exchange over an out-of-band interface like NFC. The functions gap_passkey_get and gap_passkey_set are used to manage this property.

3.4.5.2.1.3 Interface to manage connections

A device can assume the following roles in BLE:

- Observer
- Broadcaster
- Peripheral
- Central

An Observer listens for undirected advertising packets which are emitted by a broadcaster. A Central is a device which can create connections to one or more Peripheral devices. The roles are supported by a special packet format, the advertising packet format.

Advertising packet format

An advertising packet consists of a set of Advertising Data (AD) structures. Each AD structure consists of one byte length, indicating the size of the data (payload) and the data part. The data part consists an AD type and an arbitrary number of AD data. See figure 3.5 for a graphical representation. The Bluetooth Core Specification Supplement (CSS) specifies a set of AD types which are needed for basic GAP functionality.

The most important of the specified AD types is the Flags type which is consist of one

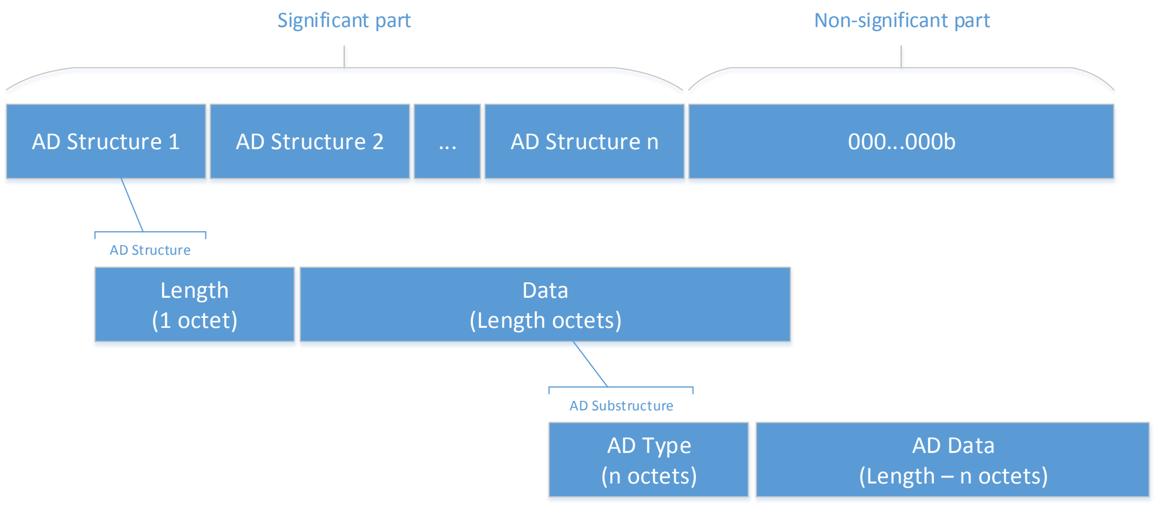


Figure 3.5: BLE Advertising data

byte payload to carry the following flags.

- Limited Discoverable Mode
- General Discoverable Mode
- BR/EDR Not Supported
- Simultaneous LE and BR/EDR to Same Device capable (Controller)
- Simultaneous LE and BR/EDR to Same Device capable (Host)

If Bluetooth Low Energy is used only then the last three flags related to BR/EDR can always be set to zero and ignored on reception. The usage of the first two flags related to discover ability will be discussed in the following sections. The Bluetooth standard defines multiple operational modes and procedures.

Broadcast mode and observation procedure

A device which assumes the role of an broadcaster configures its controller to send undirected advertising packets. The Limited/General Discoverable Mode flags have to be set to zero in this mode.

The observation procedure means that a device does active or passive scanning for advertising packets. Passive scanning means just list listen and receive the packets without sending anything. Active scanning means analyzing the packets the send scan request where possible and then process the scan response. With this technique the payload size of advertising packets can be doubled. Data that often changes should be put in the advertising packets and data that changes less often should be placed in the scan response packets. Bluetooth controllers offer an option to filter duplicate advertising packets from the same origin. If dynamic is placed in advertising data then this feature must be deactivated.

The following interface is provided to support the Broadcaster and Observer role. The Broadcaster role can be entered by calling the gap_scan_enable function. The data of the advertising packets can be set using the gap_advertising_set_data function. The gap_scan_enable and gap_scan_disable can be used to enter and leave the Observer role.

```
void gap_advertising_start();
void gap_advertising_stop();
void gap_advertising_set_data(UINT8* data);
void gap_scan_enable();
void gap_scan_disable();
```

3.4.5.2.2 NFC stack

The NFC protocol stack supports the Tag Type 4 standard which is described in [1], a document from the NFC Forum. This standard defines a data structure and an interface to access it. In addition a custom security layer with encryption and message checksums is integrated.

3.4.5.3 Interface drivers

Several interface drivers are required to interact with other hardware components. Drivers are platform-specific while the remaining modules are portable to other systems.

3.4.5.3.1 I2C driver

Since the temperature sensor is connection via an I2C bus to the application controller an I2C driver is required. The driver must provide functions to initialize the driver, send and receive a data frame.

3.4.5.3.2 GPIO driver

This driver is required to retrieve the status of the reed sensor. The driver must provide an interface to check the voltage level of a GPIO pin which can be high or low.

3.4.5.3.3 UART driver

In order to communicate with the Bluetooth transceiver a UART driver is needed. The interface of this driver must provide functions to initialize the connection, send and receive a data frame.

3.4.5.3.4 RFI driver

The Radio Frequency Interface (RFI) driver is the base for the NFC protocol. This is the most complex of all integrated drivers and is provided by Infineon as application note. This reference driver will be used in the project.

3.4.6 Android App

The Android app allows the user to communicate with the Personal Assistant Device via the BLE and NFC interfaces. Two tabs are provided, one for each interface. The NFC tab provides control elements to activate the device, retrieve its Bluetooth address (pairing) and allows creating a Bluetooth Low Energy connection with it. The BLE tab shows the connection status and the sensor values from the device.

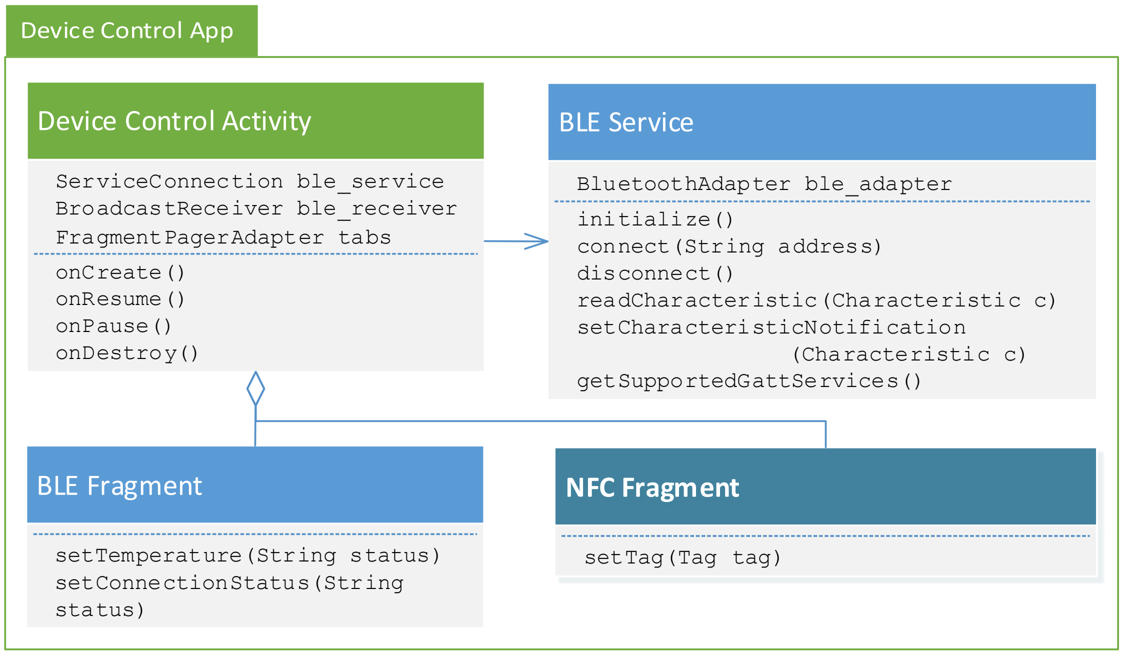

Device Control App
Device Control Activity
ServiceConnection ble_service
BroadcastReceiver ble_receiver
FragmentPagerAdapter tabs
onCreate()
onResume()
onPause()
onDestroy()
BLE Service
BluetoothAdapter ble_adapter
initialize()
connect(String address)
disconnect()
readCharacteristic(Characteristic c)
setCharacteristicNotification (Characteristic c)
getSupportedGattServices()
BLE Fragment
setTemperature(String status)
setConnectionStatus(String status)
NFC Fragment
setTag(Tag tag)

Figure 3.6: Logical View - Android App UML class diagram

As seen in figure 3.6 the App consists of a main activity called ”Device Control Activity”. This activity contains a FragmentPagerAdapter containing to tabs, the BLE fragment and the NFC fragment. In addition there exists a BLE Service which is managing the Bluetooth Adapter of the system. The activity is bonded to that service, has the ability to call functions using the service interface and is able to receive notifications from the service. For this purpose the activity has the ServiceConnection member which allows creating a bond with the service and the BroadcastReceiver which allows receiving events from the service. The functions onCreate, onResume, onPause and onDestroy are called by the Android system to manage the life-cycle of the app. The onCreate method is called when the activity is created by the system because the user started it or moved the phone to the personal assistant device which triggered the detection of the NFC interface connection. The onPause function is called whenever the activity looses focus and some other app is on top of this app. onResume is the inverse event when the activity regains the focus. At last the onDestroy method is called when the user or system ends the app. It is used to clean up used resources like the service connection.

The BLE fragment contains to public methods which allows the activity to set the current information values in the user interface once updates from the service arrive. The NFC fragment contains only a method setTag which is called by the activity when it is started by a NFC intend which means the phone is near the device. Then a reference to the tag is passed to the fragment, allowing it to communicate with it.

There are a set of functions in the BLE service class with allow to access the GATT server of the personal assistant device. The initialize method enables the Bluetooth adapter and stores a reference in the member ble_adapter. The connect method allows to create a connection with a device with given Bluetooth device address. The disconnect method ends an open connection. The getSupportedGattServices function allows retrieving a list of all available services on a connected device. After using this function the readCharacteristic method can be used to retrieve data from a service like the temperature values from the sensor in this project.

3.5 System interaction

This section shows the systems interaction when using the device as indicated in the use cases. A process view diagram should assist in understanding how the components interact with each other.

3.5.1 Device activation via NFC

Figure 3.7 visualizes the device activation process. The Android smartphone sends a command via the NFC interface to the SLE70 application controller. This command is processed by the NFC Handler task. If the command is valid the BLE Handler interface is used to trigger the corresponding functionality in the Generic Access Profile (GAP) layer. First the function "init_controller" in the GAP is called which results in an Host Controller Interface (HCI) command which is send over the UART driver to the BLE controller. Then the "enableAdvertising" command is send. Once both commands are executed by the BLE controller the device is ready for a BLE connection which is initialized by the smartphone.

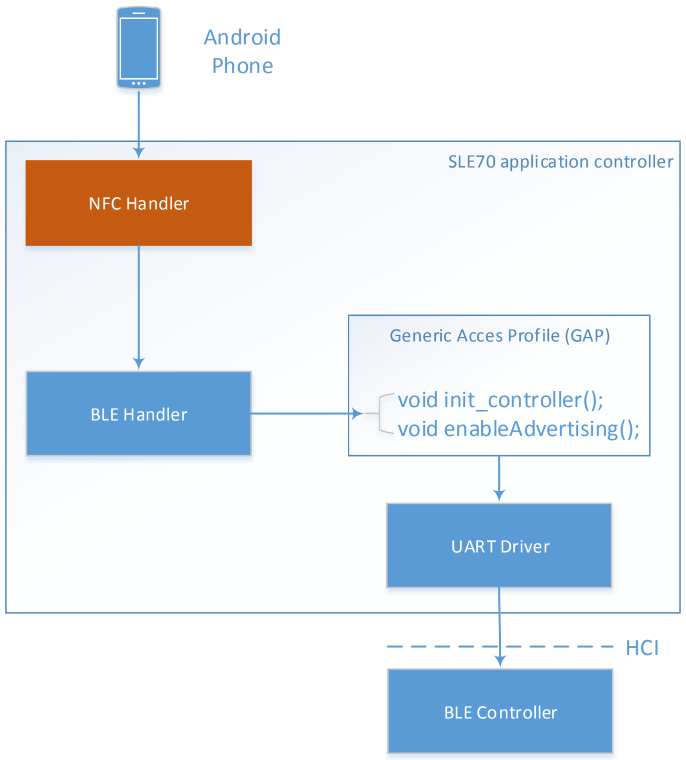


Figure 3.7: Process View - Device activation via NFC

Chapter 4

Implementation

This chapter describes the implementation of the software of the application controller as well as the implementation of the hardware and the development of the Android App.

4.1 Development Flow

This section introduces the tools used to create the software necessary to operate the personal assistant device.

4.1.1 Application Controller Software

The software is based on the Infineon Dual Sim application note. This framework provides a cooperative multi-tasking operating system with drivers for the UART and RFI communication. Multiple tasks can be in running state at the same time and a scheduler is responsible for choosing the next task to be executed in the processor.

There are two fundamental different ways to define at which time the scheduler is triggered to choose the next task. This operating system is a cooperative multitasking system. The scheduler switches to to next task after a task is finished and yields which means passing over the control to the scheduler. In contrary in a preemptive system an timer interrupt is used to stop a running task to trigger the scheduling of the next task. The advantage of a cooperative system is that shared resources do not have to be protected by synchronization methods like a mutex or semaphore.

Each task in the operating system requires a stack for local variables. In addition the stack is used to backup the processors registers before a task switch is done and to restore them when the task is scheduled again.

The Keil standard libraries provides a heap memory implementation. A static memory area of sufficient size will be used as heap memory. A wrapper module around this functionality is required in order to link heap segments to tasks. This is required because tasks can be stopped at any time when the corresponding interface is disconnected and then heap segments have to be freed to prevent memory leaks.

4.1.1.1 IDE and Tools

Figure 4.1 illustrates the flow. The Keil Integrated Development Environment (IDE) is used to develop the software for the SLE70 application controller. In the IDE the C header and source files are created using an integrated text editor. Once the toolchain is started the preprocessor includes all referenced header files and processes all preprocessor macros. Then the compiler creates an object file for each source file. Finally all object files are put together by the linker. The postlocator takes care of the platform specific memory mapping and the final result is a HEX file, a text file containing the the code bytes formated as hexadecimal values. This HEX file can be downloaded (flashed) on a SLE70 controller using the tool Infineon SmartcardManager. Alternatively the HEX file can be used in the SLE70 simulator to run tests. This testing is limited because the behavior of the hardware modules is simplified and the timings are different than on the real hardware. However is is possible to test at least some software functions with prepared input arguments (function parameters).

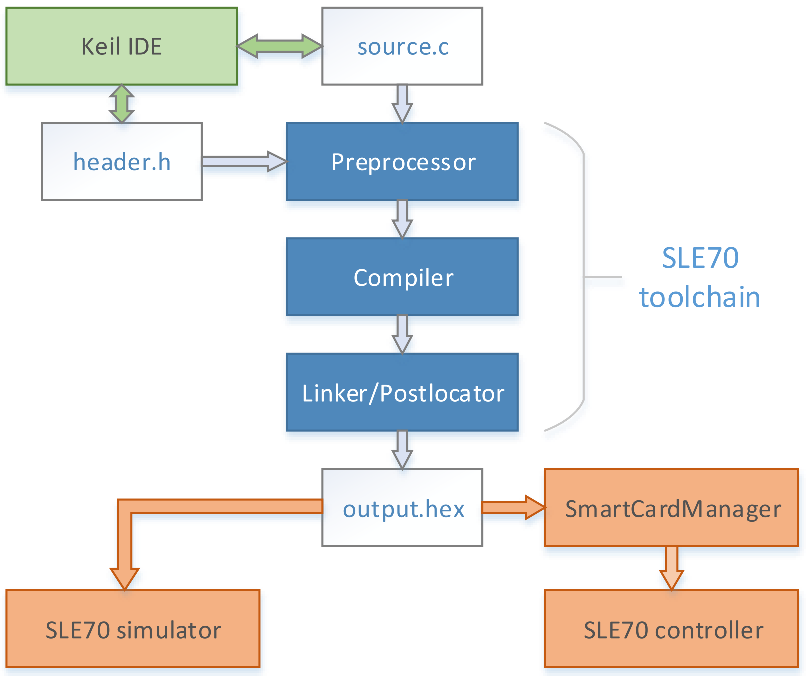


Figure 4.1: SLE70 toolchain

4.1.1.2 Bluetooth Stack development

The stack is designed to run on multiple platforms. During writing this thesis the Infineon SLE70 platform and Microsoft Windows are supported. The support of Windows makes developing and debugging much easier. At first the stack was developed and tested on Windows which allowed debugging using the functionality of Microsoft Visual Studio. After verifying the functionality the software was ported to an Infineon security controller where less debugging options are available.

Multi-platform capability is accomplished by using preprocessor macros to have different implementations for platform specific functions (like HCI communication or thread synchronization). When compiling the sources files the target platform is detected by using compiler defined preprocessor macros. For example the macro '_WIN32' is defined when using the Visual Studio compiler on Windows.

One challenge when developing for multiple platforms is to take care the endianess of the processor. Intel processors running Windows are using litte-endian word storage while

Infineon SLE70 controllers are using big-endian storage. Whenever multi-byte values are accessed it is important to take this difference into account.

Figure 4.2 shows how the stack was developed and functionally tested. First the stack was development as part of a Win32 C Application on Microsoft Windows. A Win32 window form is used show a transmission log to the testing user and provide control elements for the BLE stack functions. Since the PC has just USB connectors a USB to UART converter is needed. A FTDI FT230X dongle is used for that purpose. That way the PC can be connected to the TI CC25664 BLE controller. After the code has been proven to work, the stack was ported to the SLE70 controller. A task is now replacing the Windows GUI.

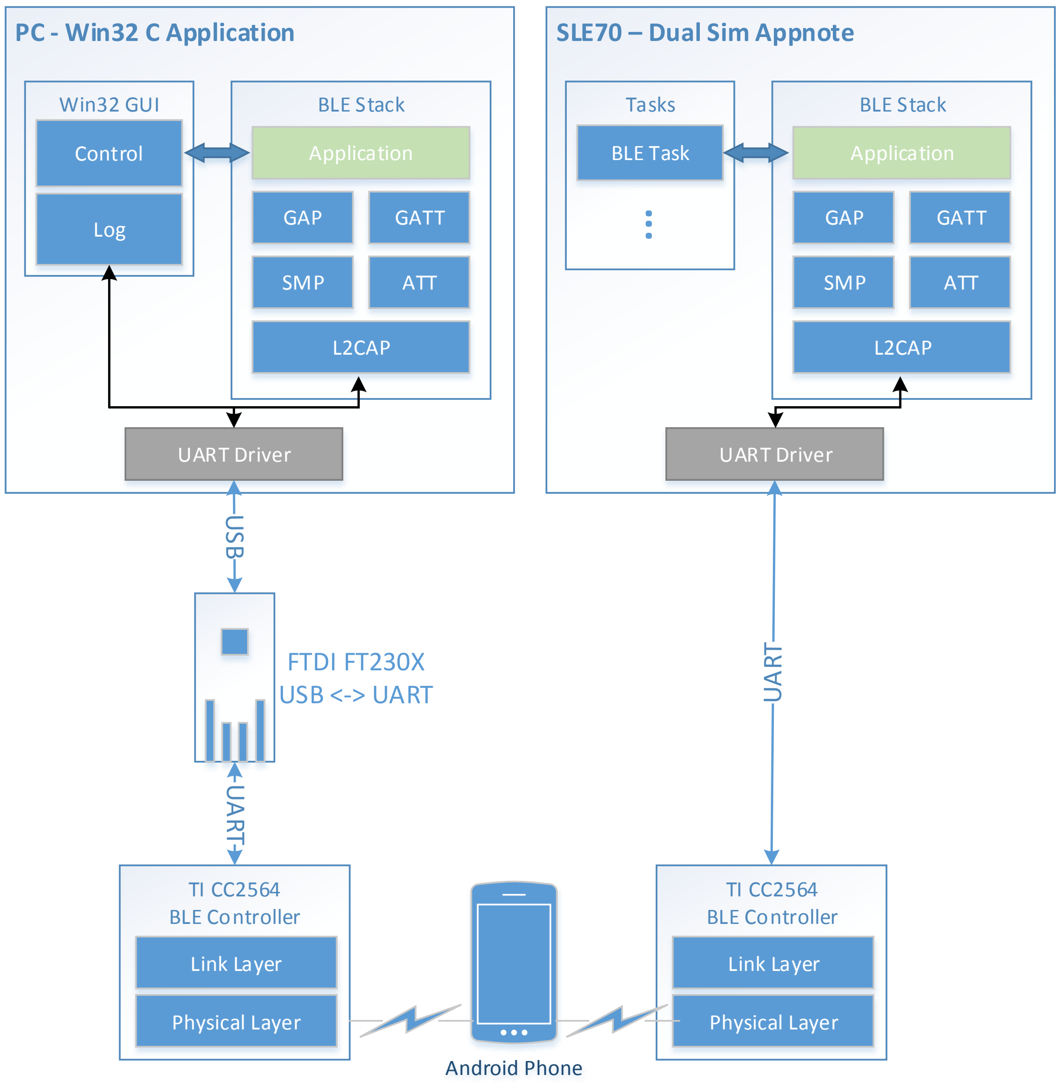


Figure 4.2: BLE stack development overview

4.1.2 Android App Tools

The Android Studio IDE was used to develop the Android App. For testing a LG (Google) Nexus 5 phone was used. Using this IDE is the standard aproach recommended by Google and plenty of documentation is available on the Google Android Developer Website.

4.2 Software Implementation

4.2.1 Bluetooth Low Energy Stack

This chapter describes the architecture of the BLE stack implemented on the personal assistant device. All layers in the host part of the stack will be explained in detail.

4.2.1.1 L2CAP Layer Implementation

The L2CAP layer is the lowest layer in the host part of the BLE stack. It accesses the functionality of the Link Layer by using the Host Controller Interface (HCI). The host and the controller are either in the same device or they are located in separate devices and connected via a interface like UART or USB. In this project two separate chips are used and they are connected via UART interface.

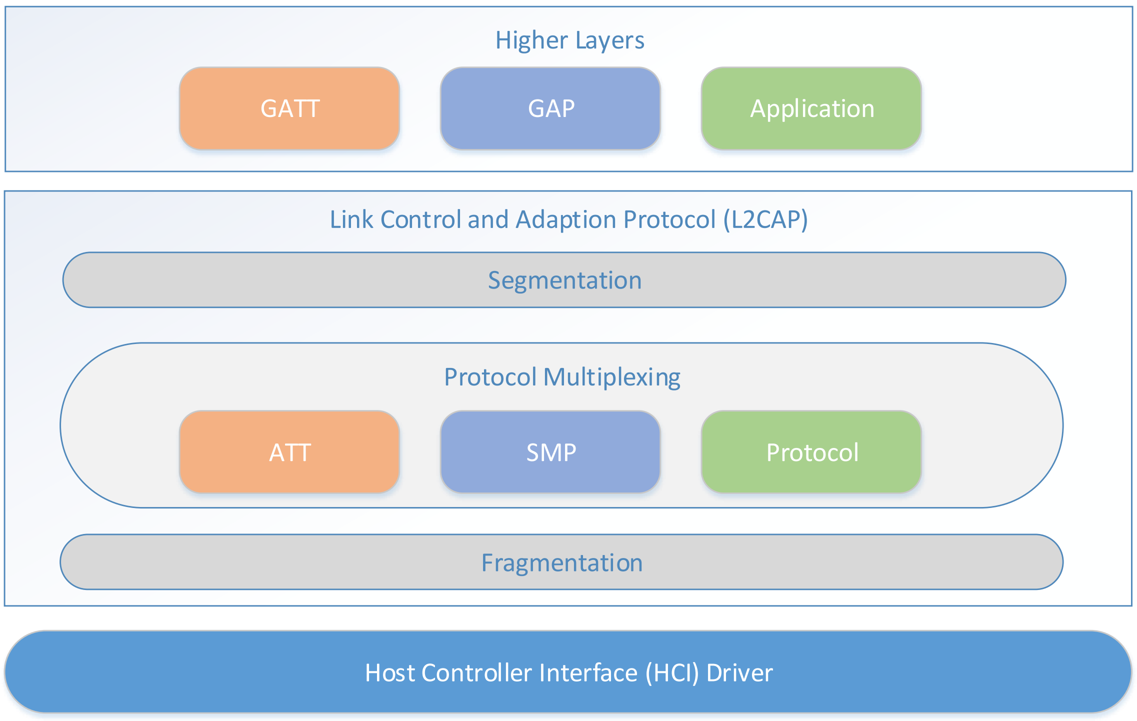


Figure 4.3: L2CAP Overview

Figure 4.3 shows how the L2CAP layer interfaces with other layers. Higher layers submit packets to be transmitted over a L2CAP channel and receive notifications once incoming packets on a selected channel are available. An interface is provided to create additional L2CAP channels.

4.2.1.1.1 Command and event handling

The L2CAP layer is designed to send HCI commands to the BLE controller and handle events received from the controller. Higher layers like the GAP layer can use the L2CAP layers interface to alter the state of the controller. For example the GAP layer can use the L2CAP interface to send a command which puts the controller in advertising mode. When the command is complete or an error has occurred the controller will send back an event to the host over the HCI which will be processed by the L2CAP layer and forwarded to higher layers if appropriate.

4.2.1.1.2 Channel multiplexing

Once a connection between two Bluetooth controllers has been established using the Link Layer raw data in form of packets can be exchanged. In practice multiple protocols are relayed over such a link. For BLE it is common to support at least the ATT and SMP. The L2CAP layer takes care of multiplexing protocols over a single logical link. For this purpose the concept of L2CAP channels is introduced. A L2CAP channel is a virtual channel of an logical link to an remote device established by the Link Layer. There can be an arbitrary number of L2CAP channels during the lifetime of a logical link. Each L2CAP channel has an assigned Channel Identification Number (CID). The BLE standard defines that three L2CAP channels always exists on each physical link, namely the ATT channel, the L2CAP signaling channel and the SMP channel. If needed the application controlling the BLE stack can create more channels.

The following table gives an overview over all possible channel identifiers according to the Bluetooth standard.

CID	Description	Remarks
0x0000	Null identifier	not allowed
0x0001-0x0003	Reserved	used only by BR/EDR controllers
0x0004	Attribute Protocol	used by the GATT layer
0x0005	LE Signaling Channel	used to send L2CAP commands
0x0006	Security Manager Protocol	used by GAP to perform pairing
0x0007-0x001F	Reserved	
0x0020-0x003E	Assigned numbers	IANA assigned protocols
0x003F	Reserved	
0x0040-0x007F	Dynamically allocated	LE credit based channels
0x0080-0xFFFF	Reserved	

Table 4.1: L2CAP Channel IDs for LE

Each channel has a defined mode of operation. The standard defines two modes of operation which are used in BLE. The Basic L2CAP Mode and the LE Credit Based Flow Control Mode.

4.2.1.1.2.1 Basic L2CAP Mode

This mode is the default mode of operation for all channels. The three always available channels are using this mode. Figure 4.4 shows how a data frame in this mode is defined. The first field "Length" refers to the length of the information payload of the packet. It is followed by the Channel ID (CID) and at last the payload follows.

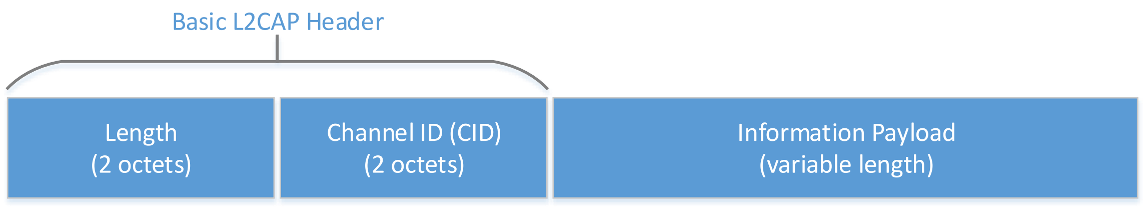

Basic L2CAP Header
Length (2 octets)
Channel ID (CID) (2 octets)
Information Payload (variable length)

Figure 4.4: L2CAP Basic Frame

4.2.1.1.2.2 LE Credit Based Flow Control Mode

A channel using this mode can be established by using the L2CAP Signaling Protocol. This mode offers flow control to prevent the receive buffer of a device being overflowed. Whenever there space is has become free in the receive buffer of an endpoint, an L2CAP Signaling Protocol packet can be sent to allow the reception of new packets.

Figure 4.5 shows visualizes a frame in this mode of operation. The difference to the Basic L2CAP mode is that the first frame of a message from a higher layer contains the "SDU Length" field which tells the receiving side how long the message will be overall. This allows a message to be distributed over multiple packets if it exceeds the size of a single frame.

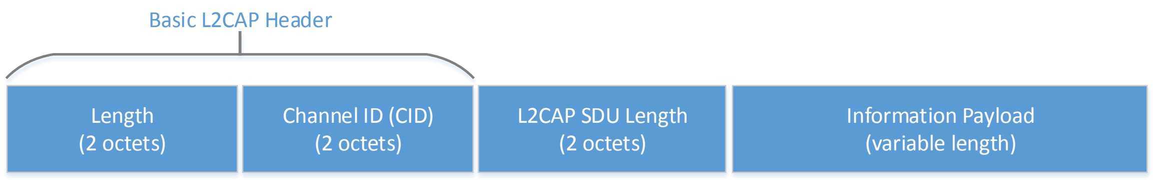

Basic L2CAP Header
Length (2 octets)
Channel ID (CID) (2 octets)
L2CAP SDU Length (2 octets)
Information Payload (variable length)

Figure 4.5: L2CAP LE Frame

4.2.1.1.3 LE Signaling Channel

The LE Signaling Channel is always exists for each link layer connection. It is intended to exchange L2CAP management commands. For BLE there are two main use cases for this channel. The first case is the "Connection Parameter Update" procedure and the second, more important use case is the creation, termination and flow control of LE Credit Based channels. Table 4.2 summarizes all possible commands relevant to BLE. The following paragraphs will discuss packet structures and procedures in more detail.

4.2.1.1.3.1 Managing LE Credit Based channels

The commands necessary to create, manage and close LE Credit Based channels will be described in the following lines. All commands consist of "Code" octet which identifies the command, an "Identifier" field which must be an unique number for each request and a "Length" field defining how much data follows.

Code	Description
0x01	Command Reject
0x06	Attribute Protocol
0x07	LE Signaling Channel
0x12	Connection Parameter Update request
0x13	Connection Parameter Update response
0x14	LE Credit Based Connection request
0x15	LE Credit Based Connection response
0x16	LE Flow Control Credit

Table 4.2: L2CAP Signaling Command Codes

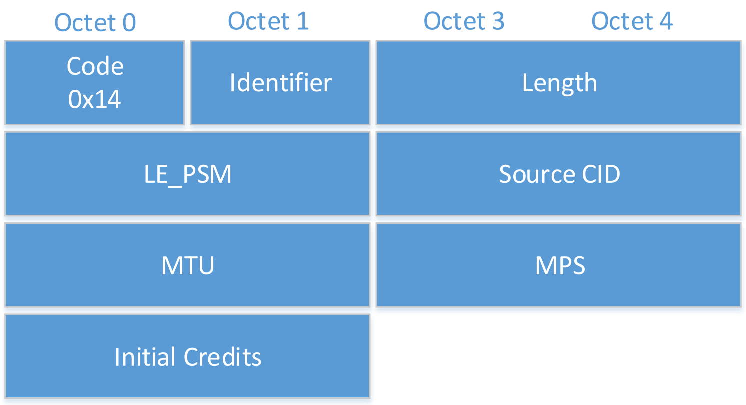


Figure 4.6: L2CAP LE Credit Based Connection Request

LE Credit Based Connection Request

The LE Credit Based Connection Request is send to request the creation of a new LE Credit Based L2CAP channel which can be used by a higher layer or application to transfer data with flow control. The packet is visualized in figure 4.6. The parameter Low Energy Protocol Service Multiplexer (LE PSM) defines for which service the channel will be used. The Source CID parameter indicates will channel id in the L2CAP layer will be assigned to this channel. The Maximum Transmission Unit (MTU) parameter can limit the maximum size of each transmitted packet over the channel. A minimum MTU of 23 octets shall be supported on BLE links. The Protocol Data Unit (PDU) field limits the maximum length of a frame received from a higher layer. A minimum PDU of 23 octets shall be supported. If the L2CAP implementation supports segmentation of packets than higher values up to the size of an 2 octet value (65533) are possible. At last the "Initial Credit" value determines how much packets can be stored in the L2CAP reception buffer for this channel.

LE Credit Based Connection Response

The LE Credit Based Connection Response is send as answer to a connection request packet. The Destination CID field indicates the channel ID of the L2CAP channel in the remote device. The MTU, PDU and Initial Credit fields have the same meaning as in the connection request packet but are corresponding to the remote side. Finally the "Result" field defines if the request was accepted or not. Table 4.3 lists possible success and error cases.

Code	Description
0x0000	Connection successful
0x0001	Reserved
0x0002	Connection refused – LE_PSM not supported
0x0003	Reserved
0x0004	Connection refused – no resources available
0x0005	Connection refused – insufficient authentication
0x0006	Connection refused – insufficient authorization
0x0007	Connection refused – insufficient encryption key size
0x0008	Connection refused – insufficient encryption
0x0009	Connection refused - invalid Source CID
0x000A	Connection refused - source CID already allocated
0x000B-0x0FFFF	Reserved

Table 4.3: L2CAP Signaling Command Codes

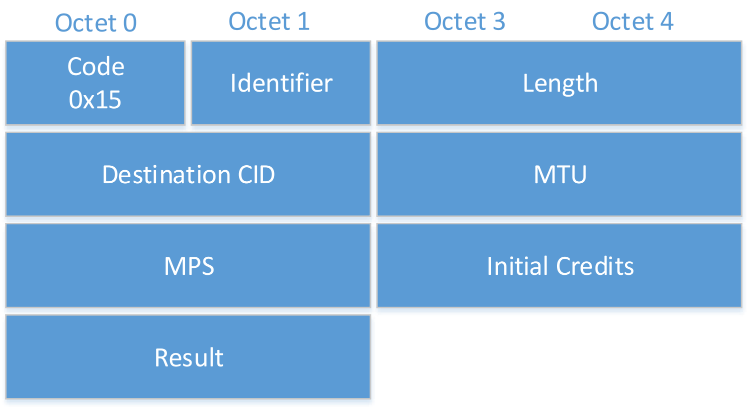


Figure 4.7: L2CAP LE Credit Based Connection Response

LE Flow Control Credit

This packet is send to grant the remote side more credits to continue sending new packages. Usually this is send whenever the free space in the L2CAP reception buffer of this channel has increased. The parameter CID refers to the corresponding channel id and the Credit parameter defines the number of packages which could be received additionally since the last credit grant.

Disconnection Request

The Disconnect Request packet is send to request the termination of a chosen L2CAP channel. Not every channel can be closed. There are some channels like the L2CAP Signaling Channel that are always available. Figure 4.9 shows the structure of this frame.

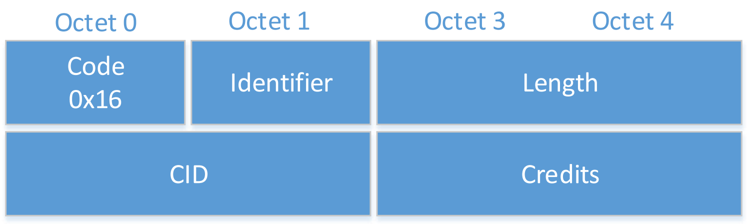


Figure 4.8: L2CAP LE Flow Control Credit

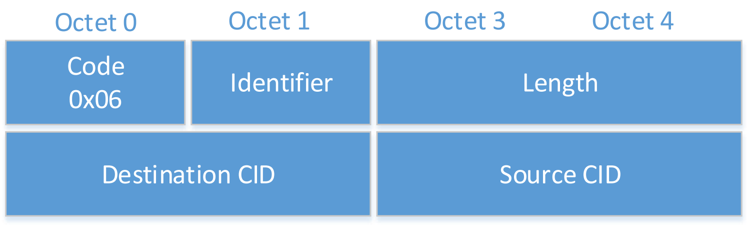


Figure 4.9: L2CAP Disconnect Request

Disconnection Response

The Disconnect Response packet is send to acknowledge the termination of a channel. If the disconnect request fails an other error message, the Command Reject packet is send. Figure 4.10 shows the structure of this frame.

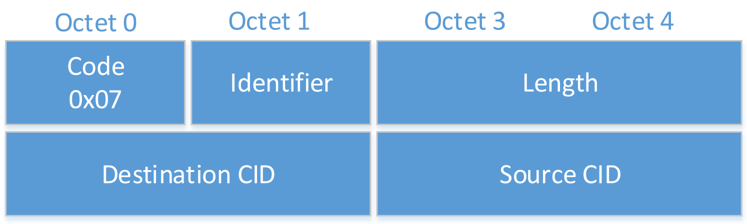


Figure 4.10: L2CAP Disconnect Response

Command Reject

This packet is send whenever an received command can not be processed or acknowledged. Figure 4.11 shows the structure of this frame.

4.2.1.1.3.2 Connection Parameter Update procedure

This procedure is used as fall-back method to adapt the link layer parameters for a connection if the primary link layer procedure is not supported by one or both controllers.

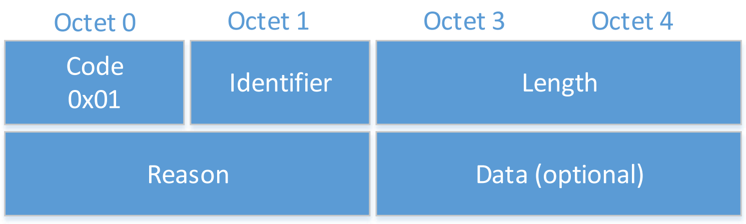


Figure 4.11: L2CAP Command Reject

L2CAP Connection Parameter Update Request

This packet as seen in figure 4.12 is send to request an update of the link layer connection parameters of an established connection. The "Inverval Min" and "Interval Max" fields refer to the minimal and maximal supported connection interval of a device. The connection interval defines a periodic timer interval at which the peripheral device in the link layer connection wakes up and listens for packets coming from the central device. The "Slave latency" is the number of connection intervals the peripheral device can stay sleeping to save energy before the central will disconnect the link. The "Timeout Multiplier" defines the supervision timeout which tells how long the connection can stay idle without sending any packet after connecting.

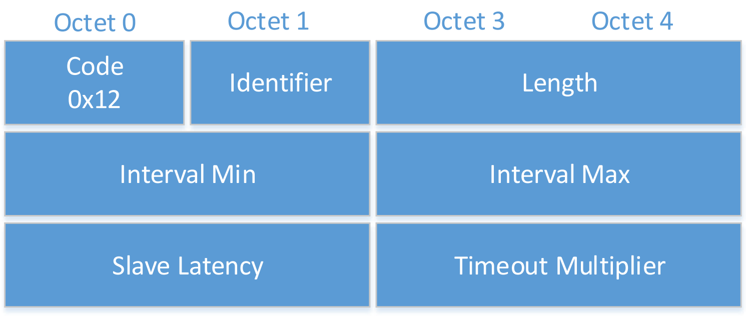


Figure 4.12: L2CAP Connection Parameter Update Request

Connection Parameter Update Response

The response packet as seen in figure 4.13 contains a "Result" field with a binary value to determine if the request has been accepted.

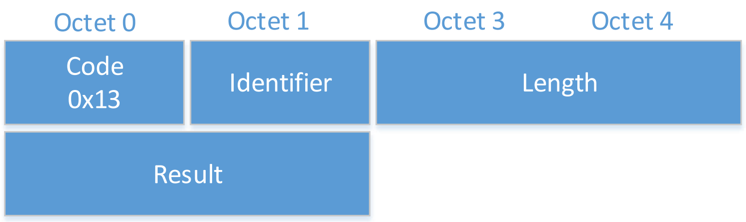


Figure 4.13: L2CAP Connection Parameter Update Response

4.2.1.2 GAP Implementation

The GAP layer acts as interface for the user or application to control the functions of the the Bluetooth device. The interface provides functions to get and set controller specific parameters like the Bluetooth device address and other properties. Most important, functionality is provided to connect to devices and activate security features like encryption by using the Security Manager Protocol (SMP).

4.2.1.2.1 Security features

In BLE there are two supported security modes. LE Security mode 1 offers encryption and LE Security mode 2 offers data signing (Message Authentication Codes). Only one mode can be used at a time. This leads to a much weaker security compared to other protocols like TLS where encryption and MAC are used combined.

Each security mode can be used in different levels. The levels define if and how authentication is done. Authentication is necessary to provide protection against man-in-the-middle attacks and unauthorized access to devices. Bluetooth is often used in medical devices which may attached to the patients body. These devices are often vital and no unauthorized user should be able to access the device or alter its functionality. Considering such use cases there are high security requirements. LE Security mode 1 provides the following levels according to the standard:

1. No security (No authentication and no encryption)
2. Unauthenticated paring with encryption
3. Authenticated paring with encryption
4. Authenticated LE Secure Connections pairing with encryption

Authentication is done by using a out-of-band (OOB) channel to transfer a key. If devices have a display it is common to ask the user to manually enter a paraphrase or key shown at one device into the other. Alternatively NFC can be used to transfer a key over a short distance. All methods assume that the second channel is secure.

In LE security mode 1 level 4 the Elliptic Curve Diffie-Hellman (ECDH) protocol is used for key exchange using the NIST P256 curve. This protocol provides forward-security which means even if the OOB information is leaked the encryption key remains secure if the paring was successful and there was no man-in-the-middle attack. LE Security Mode 2 provides the following levels of security according to the standard:

1. Unauthenticated paring with data signing

2. Authenticated paring with data signing

If data signing is enabled, each packet is appended with a signature consisting of a counter and Message Authentication Code (MAC) as illustrated in figure 4.14. The MAC requires a key which is referred as Connection Signature Resolving Key (CSRK). The count is also part of the MAC input to ensure that sending multiple times the same message will not result in the same signature.

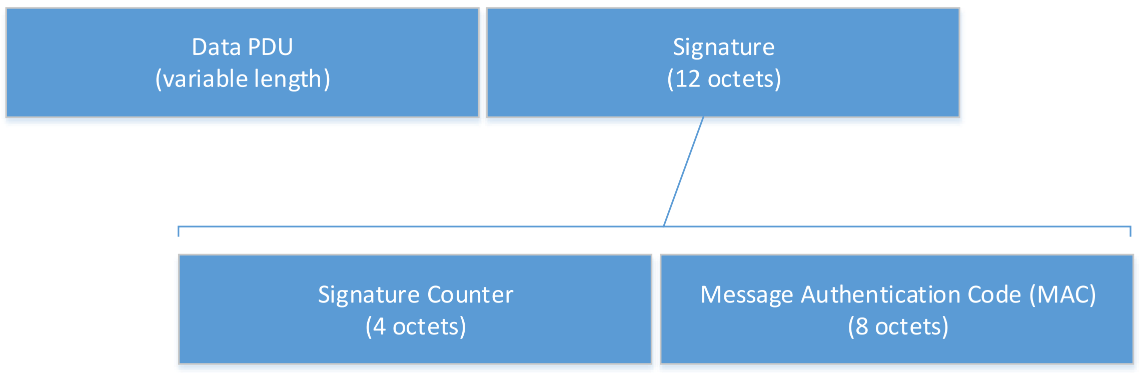


Figure 4.14: BLE Security mode 2 data signing

Before any security mode can be enabled a pairing or bonding process has to be completed. The term pairing refers to a temporal binding between to devices while bonding means a permanent binding. A permanent bond means that the security features can be re-enabled after reconnecting the devices. This also implies that a persistent memory is required the store the security keys for a long time period.

4.2.1.2.2 Paring using the Security Manager protocol

Figure 4.15 gives an overview of the paring procedure. Once a link layer connection between two devices has been established the L2CAP layer provides a automatically created channel for the SMP. By sending message over this channel the paring procedure is performed.

In phase 1 the capabilities of the devices are exchanges. This includes weather a device I/O capabilities like a keyboard or a display and if OOB information (for example from NFC) is available.

Phase 2 has the purpose to generate the Long Term Key (LTK) for LE Secure Connections.

After phase 2 has been completed the new key can be used to enable encryption on the link layer. This is done using the HCI to configure the controller to use the key from now on.

Phase 3 is optional to exchange additional keys like Identity Resolving Key (IRK).

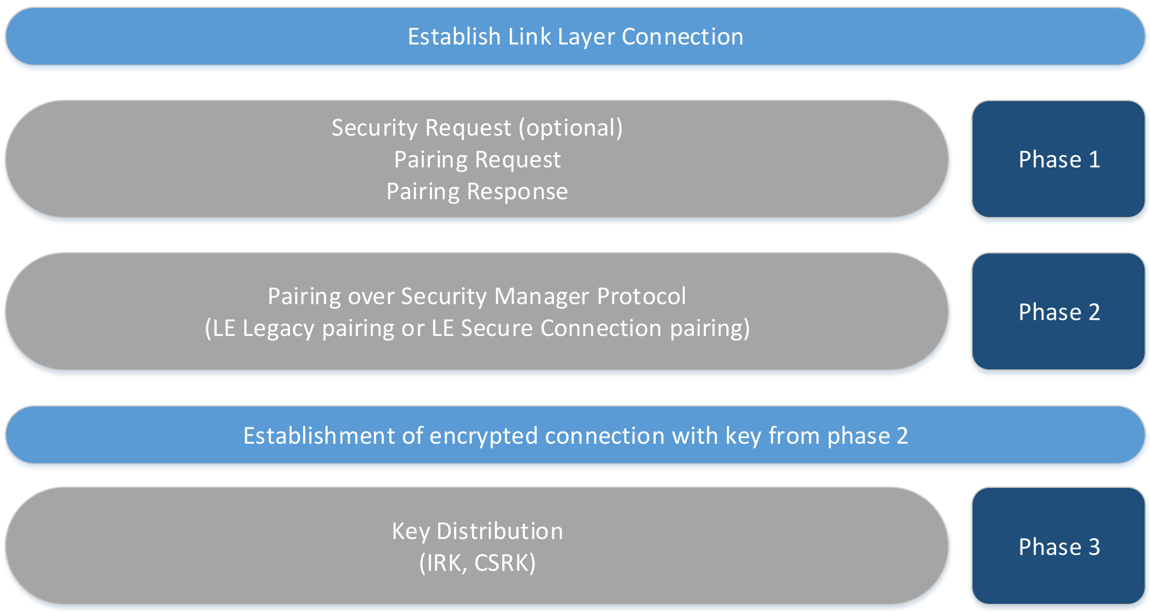

Establish Link Layer Connection
Security Request (optional)
Pairing Request
Pairing Response
Phase 1
Pairing over Security Manager Protocol
(LE Legacy pairing or LE Secure Connection pairing)
Phase 2
Establishment of encrypted connection with key from phase 2
Key Distribution
(IRK, CSRK)
Phase 3

Figure 4.15: BLE pairing overview

4.2.1.3 GATT Server Implementation

The Generic Attribute Profile (GATT) Server consists of a database conforming to a certain structure. By complying to the standard it is possible to make devices interpolatable usable. The GATT database contains a so called profile, consisting of one or more services, each of them containing characteristics. Each characteristic contains data in a certain format, for example an integer or a string. The GATT server provides an external interface for GATT clients to access the data stored in these data structures. The protocol to perform queries on the server is called Attribute Protocol (ATT). In addition there is an local interface to extend and modify the content inside the server which can be used by the user or application to setup and maintain the GATT servers database.

4.2.1.3.1 Data structure

The GATT servers data structure is based on the ATT data structures. Each attribute contains the fields handle, type, value and permission. The handle is an unique number that can be used to reference an attribute. Typically the handle numbers are assigned in increasing order to attributes. The type contains an Universally Unique Identifier (UUID) number, which can be used to filter a list of attributes when performing a search. The value field field can have arbitrary length depending on the attribute type. This must be taken into account when implementing attribute structures in the C programming language. The permission field defines if an attribute is readable, writable or both.

Figure 4.16 shows how data is logically organized inside a GATT server. A GATT server has at least one active profile, consisting of multiple services, which again consist of service includes and characteristics. Each characteristic contains at least a value and some property flags. Optionally multiple descriptors can be part of a characteristics. The following section will describe the meaning and usage of these data structures in more detail.

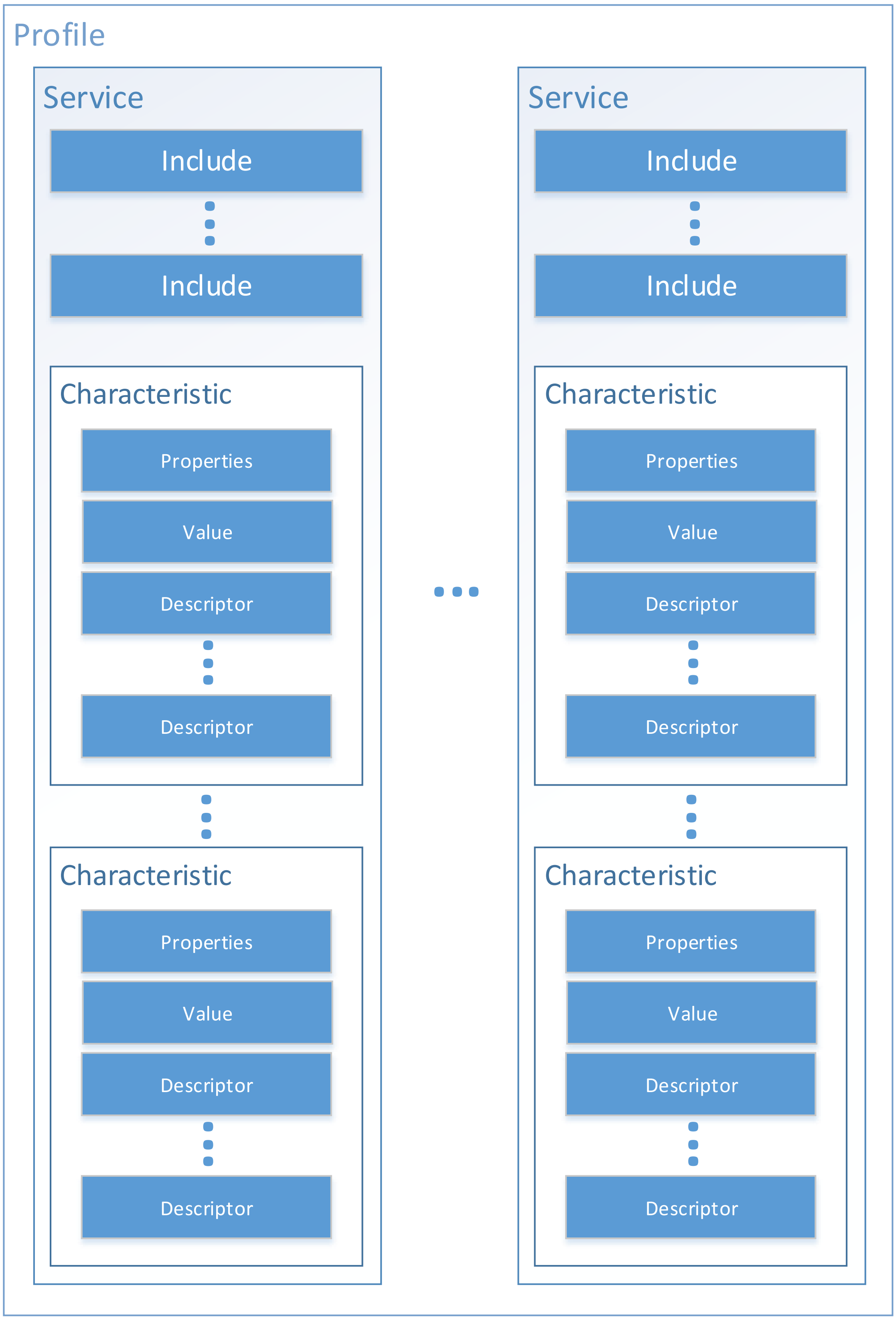


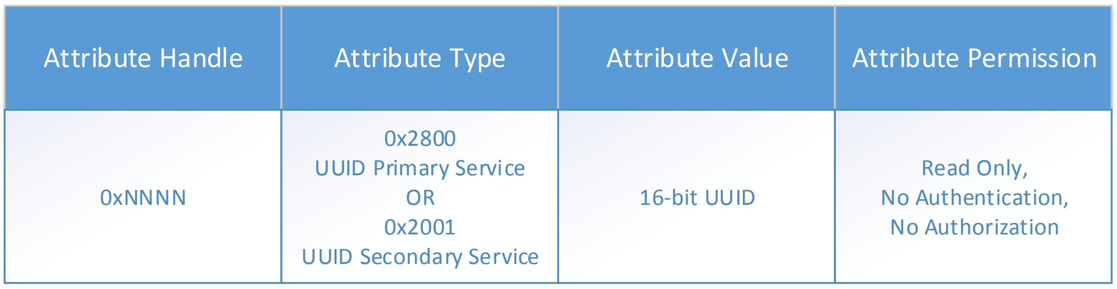
Figure 4.16: BLE GATT Profile

4.2.1.3.1.1 Profile

The profile is the highest element in the GATT servers hierarchy. In theory a GATT server could allow the switching between multiple profiles, however in practice it is common that devices only support one profile. A profile is a container for one or more services. This grouping element is assumed to exist always and no concrete data structure is implemented.

4.2.1.3.1.2 Service

A service is a container element which can contain "service includes" and arbitrary many characteristics. There is a distinction between primary and secondary services. A primary service can be discovered using the GATT server interface by doing a "Service Discovery" procedure. A secondary service is usually included by other services to extend the functionality of an existing service. From a technical point of view both types of services have the same data structure and differ only in the attribute UUID and logical usage context.



Attribute Handle	Attribute Type	Attribute Value	Attribute Permission
0xNNNN	0x2800 UUID Primary Service OR 0x2001 UUID Secondary Service	16-bit UUID	Read Only, No Authentication, No Authorization

Figure 4.17: BLE GATT Service

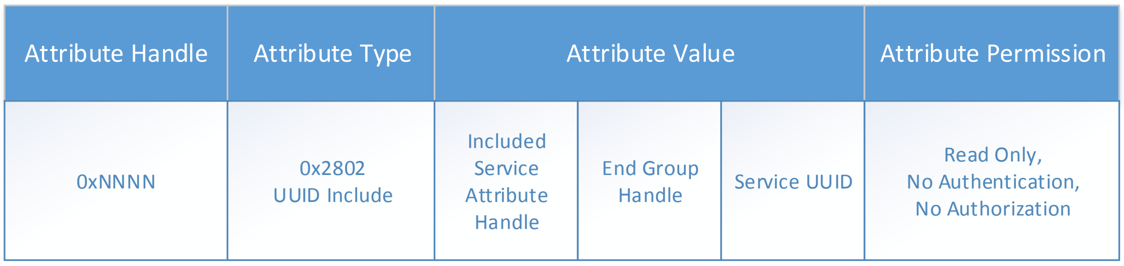
The definition of the GATT service can be directly translated in a C struct. All structs are added to a linked list to allow iterating over them. Additional members as the end_handle and size are required to optimize the process of iterating through all available service structures.

```
typedef struct {
        UINT16 handle;
        UINT16 type;
        UINT16 value;
        UINT8  permission;

        UINT16 end_handle;
        UINT16 size;
} GATT_SERVICE;
```

4.2.1.3.1.3 Service Includes

Each service can include a list of other services in its definitions that may be useful in the same context. This feature is optional.

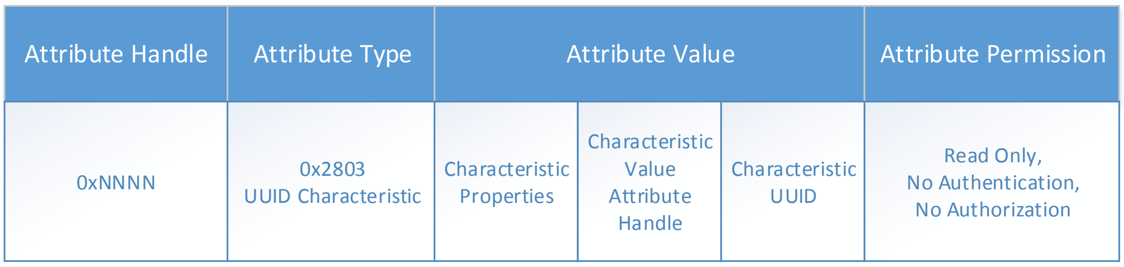


Attribute Handle	Attribute Type	Attribute Value			Attribute Permission
0xNNNN	0x2802 UUID Include	Included Service Attribute Handle	End Group Handle	Service UUID	Read Only, No Authentication, No Authorization

Figure 4.18: BLE GATT Include

4.2.1.3.1.4 Characteristics

A characteristic contains an UUID number to specify its type, a value, properties and optionally an arbitrary number of descriptors.



Attribute Handle	Attribute Type	Attribute Value			Attribute Permission
0xNNNN	0x2803 UUID Characteristic	Characteristic Properties	Characteristic Value Attribute Handle	Characteristic UUID	Read Only, No Authentication, No Authorization

Figure 4.19: BLE GATT Characteristic

Table 4.4 gives an overview of all possible defined descriptor property values. Broadcast means that a characteristics value can be included in advertising packets (see the GAP layer). By using the read and write properties it is possible to define data sources and data sinks which can be used as command interpreter (data sink) and command response point (data source). The command response characteristic may also have the Notify or Indicate property set. If the Notify property is set, a message may be send to the GATT client each time the characteristic value changes (if activated in the corresponding client characteristic configuration descriptor). The Indicate property is similar but the GATT client has to acknowledge each indication (and the next indication is delayed until the current has been acknowledged). The Indicate and Notify properties can be set independently by each GATT client. This increases the memory consumption and implementation effort because for each GATT client the state of each characteristic properties must be stored. If device bonding via the Security Manager is supported then this information must be stored even after a client disconnects the physical link or the device restarts which implies a persistent memory must be used. The Authenticated Signed Write property indicates that data signing defined in LE Security Mode 2 can be used to write the characteristics value. The "Extended Properties" flag means that an "Extended Properties Characteristic Descriptor" exists within the characteristic and contains additional information.

Each characteristic can contain an arbitrary number of descriptors which provide additional information about the characteristic value. The standard defines a set of descrip-

Properties	Value
Broadcast	0x01
Read	0x02
Write Without Response	0x04
Write	0x08
Notify	0x10
Indicate	0x20
Authenticated Signed Write	0x40
Extended Properties	0x80

Table 4.4: GATT Characteristic Properties

tors which can be used by any application. It is possible for the user to define custom application specific descriptors. The next section describes the descriptors defined in the Bluetooth standard.

Extended Properties Characteristic Descriptor

This descriptor is used the extend the characteristic property bit-field with additional properties as seen in table 4.5. One property is the Reliable Write ability which enables support for a procedure where written data is echoed back to the client and the client must acknowledge it before it becomes valid. If the Writable Auxiliaries field is set active then the a Characteristic User Description descriptor exists and is writable.

Properties	Value
Reliable Write	0x0001
Writable Auxiliaries	0x0002
Reserved	0xFFFC

Table 4.5: GATT Characteristic Extended Properties

Characteristic User Description Descriptor

This descriptor contains an UTF-8 encoded string which can be shown on the GATT clients display to describe the meaning of a characteristic.

Client Characteristic Configuration Descriptor

This descriptor is special because each GATT client client has its own instance of the descriptor. The bit-field can be used to enable Notification or Indication for the corresponding characteristic value. The setting of each client lasts as long as the device bonding (via the Security Manger).

Properties	Value
Notification	0x0001
Indication	0x0002
Reserved	0xFFF4

Table 4.6: GATT Client Characteristic Configuration Descriptor

Server Characteristic Configuration Descriptor

This descriptor exists only once and has the same instance for all GATT clients. It can be used to enable or disable the broadcast of the corresponding characteristics value in advertising packets as defined in table 4.7.

Properties	Value
Broadcast	0x0001
Reserved	0xFFF2

Table 4.7: GATT Server Characteristic Configuration Descriptor

Characteristic Representation Format Descriptor

The purpose of this optional descriptor is to tell the GATT client how to interpret the characteristics value. For example the value could be interpreted as signed 16-bit integer or as 32-bit floating point number. It is also possible to define an exponent for integer type values which will be multiplied to the value by the GATT clients.

Characteristic Aggregate Format Descriptor

If an characteristics value is a aggregation of multiple values then multiple characteristic representation format descriptors might exists. This descriptor contains an ordered list to tell the clients in which orders the format descriptors are connected to the aggregated value parts. This is also useful because it allows to reuse format descriptors from other characteristics (which saves memory space).

Translation to a C struct

The characteristics definition can be translated to the following C struct. Because each characteristic has exactly one value handle, this can be integrated in the same struct.

```
typedef struct {
        UINT16 handle;
        UINT16 handle_value;
        UINT16 handle_description;
        UINT8  properties;
        UINT16 characteristic_uuid;
        UINT8* value;
        UINT16 value_size;
        UINT8* user_description;
        UINT16 user_description_size;
} GATT_CHARACTERISTIC;
```

4.2.1.3.2 Interface to GATT clients

GATT clients can access the GATT servers database using the Attribute Protocol (ATT). This protocol consists of a set of commands described in this section.

4.2.1.3.2.1 Read By Group Type Command

This command it mainly used for the discovery of services of a GATT server. Figure 4.20 shows the structure of the command. The request consists of the Opcode (0x10 for the request and 0x11 for the response), the starting attribute handle from which the search in the GATT database should start, the end handle (which can be set to 0xFFFF if the search should be exhausting) and the group type UUID (set to the UUID for primary service for service discovery).

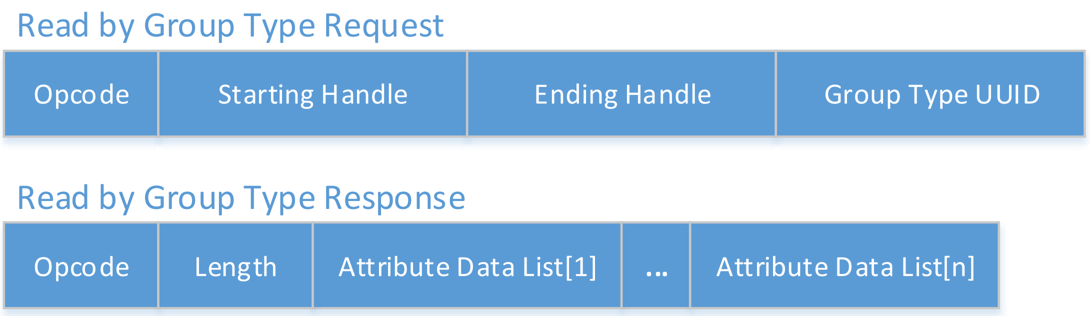


Figure 4.20: ATT Read By Group Type Command

4.2.1.3.2.2 Find By Type Value Command

This command is used to check if a specific service with known UUID is part of a GATT servers database. The corresponding procedure is called ”Primary Service Discovery by UUID”. Figure 4.21 visualizes the request from the client and the response from the server. The Opcode (0x06 for the request and 0x07 for the response). The attribute type can be set to the primary service UUID and the attribute value to the UUID of the service which should be found. The response contains a list of tuples where each tuple consists of a attribute handle and the group end handle. When doing a search for services the end handle defines the last attribute handle corresponding to the found service. This is important because after the client knows the attribute handle range of a service, it can discover all includes or characteristics of that service in this range.

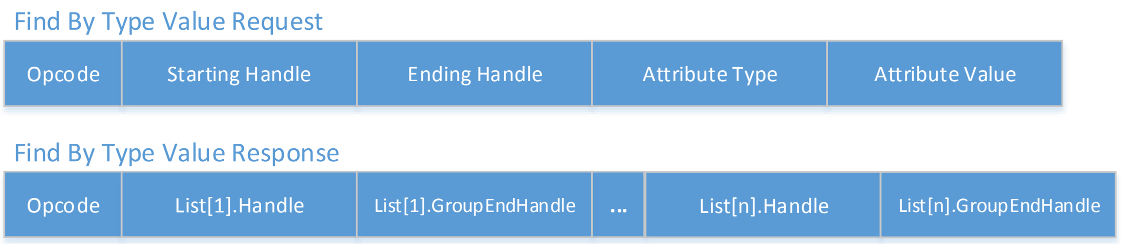


Figure 4.21: Find By Type Value Command

4.2.1.3.2.3 Read Command

Figure 4.22 shows the very simple structure of the read command. The request contains the attribute handle of the attribute to read and the response contains the value. This command can be used to read the value of a characteristic.

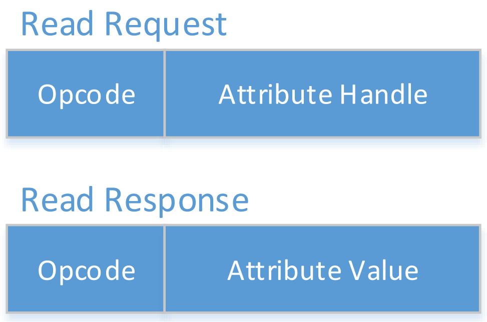


Figure 4.22: Read Command

4.2.1.3.2.4 Read By Type Command

Figure 4.23 shows the structure of the read by type command used to read all attributes in the specified range with a given type (UUID). The response consists of a list of values where the Length field specifies of how many octets each list element consists.

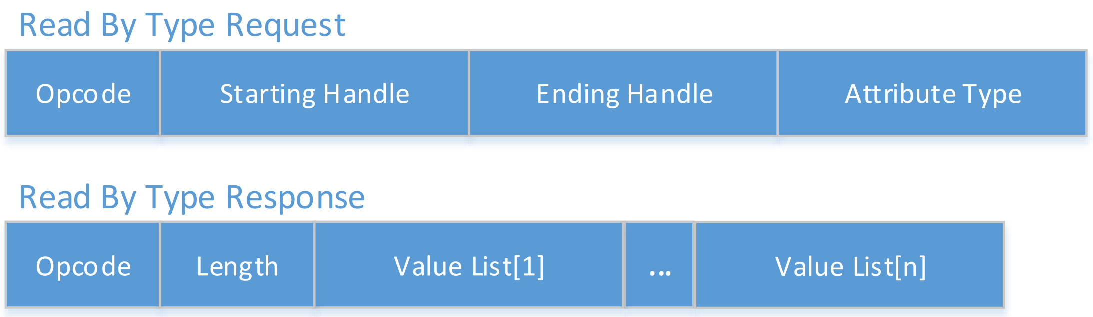


Figure 4.23: Read By Type Command

4.2.1.3.3 Interface to application

The interface exposed to the local user or application allows filling the GATT servers database with new elements like services and characteristics. This interface is vendor and implementation specific. It should at least allow the following procedures.

- Add a service to the profile

- Add an service include to a service
- Add a characteristic to a service
- Add descriptors to a characteristic

4.2.2 I2C Interface driver

The I2C interface is used to connect a temperature sensor to the SLE70 application controller. The SLE70 controller used contains no hardware module to act as I2C master so an implementation in software using the GPIO pins is necessary. Two pins are required for data and clock. See [23] for a general introduction to the standard.

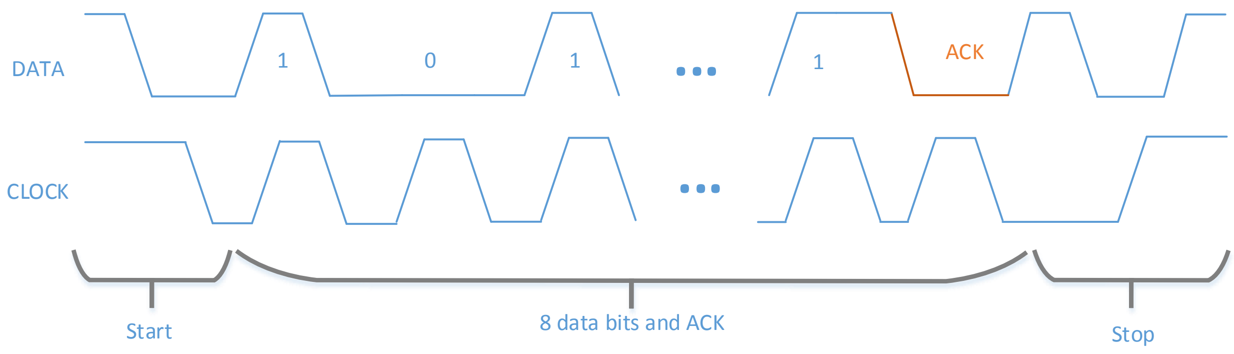


Figure 4.24: I2C frame

Figure 4.24 shows a frame. All used pins must be configured as pull-up since high is the default state of all lines. To begin data transmission a so called start transition must be generated. This condition is defined as pulling the data line down while the clock is high. Now a data byte can be transmitted. A byte consists of eight bits and each bit is transmitted by bringing the data line in the corresponding state and then pulling the clock high. The first data byte to be transmitted is special because it contains the address of the device on the I2C bus to which the data should be transmitted. The address consists of seven bits and the eight bit of the byte is used as indicator bit to tell if a read or write operation will be performed. After each transmitted byte the addressed slave device must send an acknowledge bit which means the data was received. This is done by the addressed device by pulling the data line low for a clock cycle. At last after all bytes have been transmitted a stop condition is transmitted. This is implemented by pulling the data line to high while the clock is already high.

During the transmission the data line pin must be reconfigured as input pin to read the acknowledge bits. Since the clock is generated also by this software is possible to meet all timing requirements with no problems.

4.2.3 UART Interface driver

Since the SLE70 controller supports only one-wire UART (which is half-duplex) extensions are needed. A two-wire UART connection plus hardware flow control with additional two wires RTS and CTS is needed. The existing driver is extended to support hardware flow control using two additional lines. These two lines are implemented by using two GPIO pins. By default the RTS line is asserted, allowing the other side to send frames. Whenever the a frame is to be send out, the RTS is cleared to prevent the other side from sending and once the line is clear data is send. After that RTS is asserted again to allow reception again.

4.2.4 NFC Stack

The NFC interface uses the Radio Frequency Interface (RFI) as transport channel. The NFC Tag Type 4 standard defines how data is logically structured inside an NFC tag and a set of commands to read and manipulate this data.

4.2.4.1 Tag Type 4

Data is stored in a simple two layered structure. The first layer is the application list. Each tag can have one or more applications. Every application contains a list of one or more files.

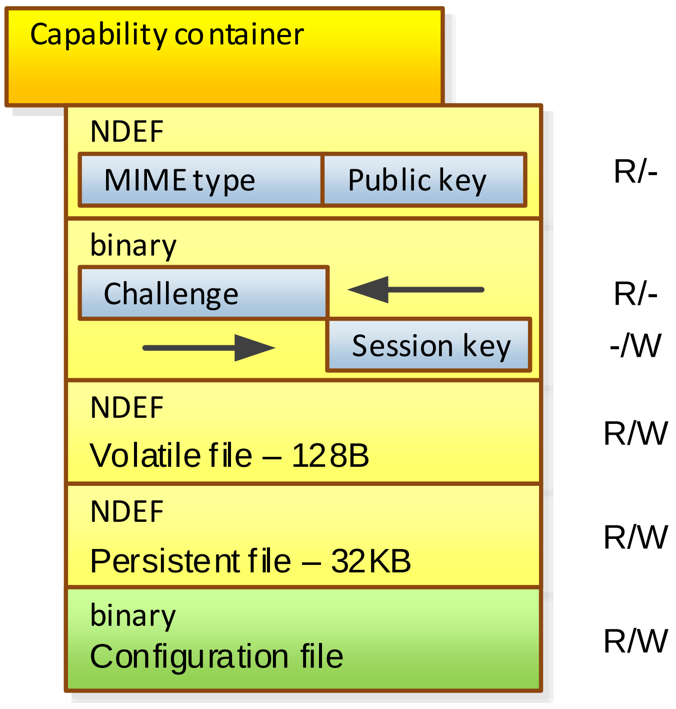


Figure 4.25: NFC Tag Type 4 structure

The first file in an application is the so called "Capability Container". This file exists only once per application and must have a certain file number which is defined in the standard. It contains a set of tag specific properties and a list of all other files in the same application.

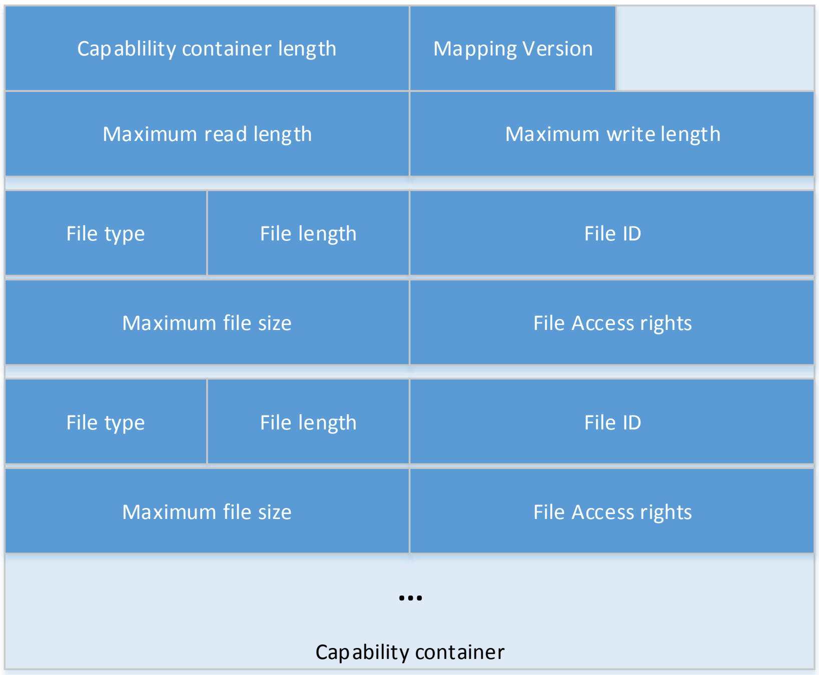


Figure 4.26: NFC Capability Container structure

Figure 4.26 visualizes the content of the capability container. Each row in the picture represents four data bytes. The first element in the file is the overall file length. This is required because a client who reads out the file has to know how much data has to be requested. The second element (one byte) is the mapping version which is a constant defined in the NFC standard. Finally the file header is concluded by the maximum length when reading or writing to a file. Embedded systems have limited buffer sizes when handling data from communication interfaces. After the header there follows a section for each file in the application. Such a section contains the file type and length, the file id which is used to select and access the corresponding file, the physical file size which the file can maximal consume in memory and access rights.

4.2.4.2 File management commands

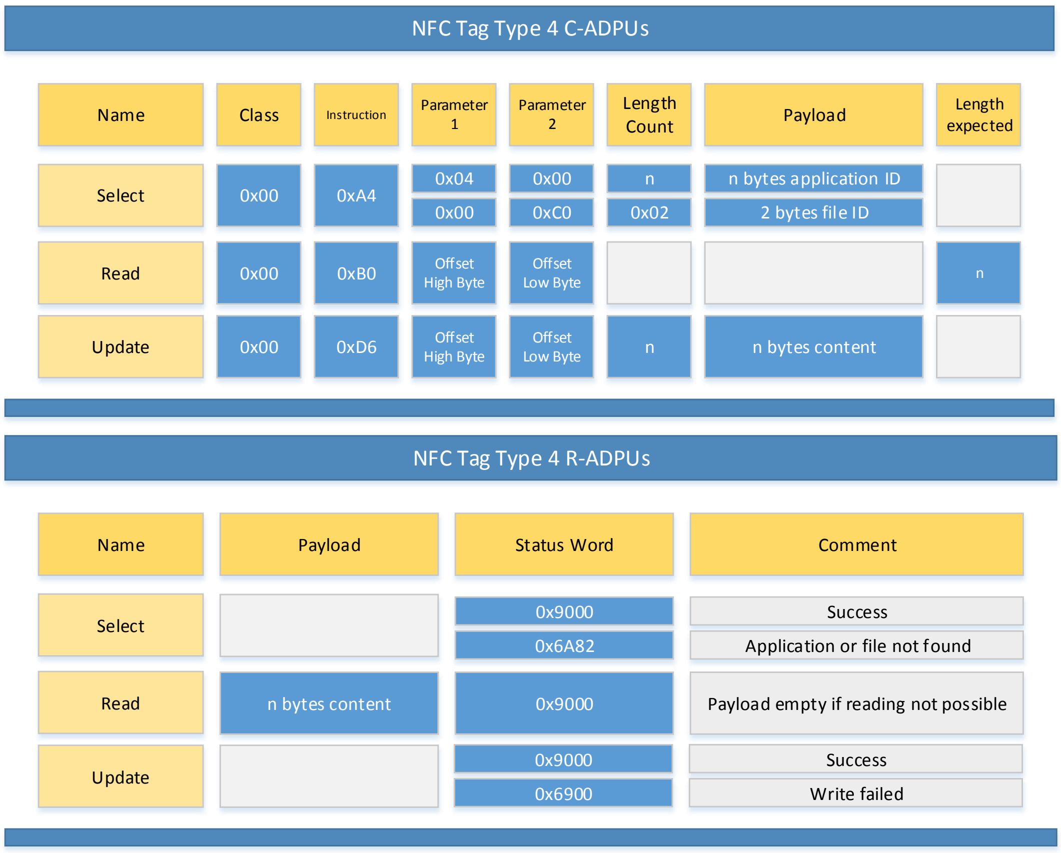
A module is required to parse and execute the commands which arrive from the NFC interface. It contains an array of command structs where each contains a command number, a bit mask of interfaces allowed to use this command, the minimum and maximum length of the received command frame and a function pointer to the corresponding routine to process the data.

There is a defined set of file management commands which have to be implemented on every NFC Tag Type 4 card.

The "Select" command has the purpose the select an application or file. When connecting to a tag the first action of the reader is to select the application of interest. Then the capability file is selected.

With the "Read" command the contents of the capability file are retrieved. The first word (2 bytes) of a file contain by definition the current file length. The reader usually reads out the word the get the file size and retrieves then the rest of the file by reading of a certain size (where the maximum frame size is defined in the header of the capability file).

At last the "Update" command can be used to update the contents of a file. The read and write commands both require the file offset position as parameter.



NFC Tag Type 4 C-ADPUs

Name	Class	Instruction	Parameter 1	Parameter 2	Length Count	Payload	Length expected
Select	0x00	0xA4	0x04	0x00	n	n bytes application ID	
			0x00	0xC0	0x02	2 bytes file ID	
Read	0x00	0xB0	Offset High Byte	Offset Low Byte			n
Update	0x00	0xD6	Offset High Byte	Offset Low Byte	n	n bytes content	

NFC Tag Type 4 R-ADPUs

Name	Payload	Status Word	Comment
Select		0x9000	Success
		0x6A82	Application or file not found
Read	n bytes content	0x9000	Payload empty if reading not possible
Update		0x9000	Success
		0x6900	Write failed

Figure 4.27: NFC Tag Type 4 command overview

Figure 4.27 gives an overview of the implemented commands. The first table lists the command ADPUs and the second table the response ADPUs. Each command frame starts with a class byte which is 0x00 for NFC. The second byte is the instruction number. Then there are parameters which meanings depends in the specific command. The length count is used to indicate how many bytes follow in the commands payload.

Whenever the reader sends a command frame to the tag it answers with a response frame. The response frame consists of an optional payload with the length specified in the length expected field of the command frame and a status word.

4.2.4.3 Authentication and secure session

Each time a user connects with a reader to the device a authentication is required to perform certain operations on the tag. After the authentication is complete an secured session is established which provides confidentiality and data integrity.

4.2.4.3.1 User management

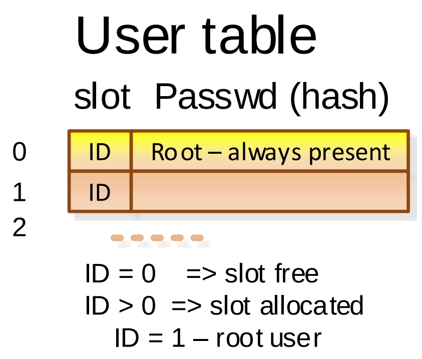


Figure 4.28: NFC user management

The application contains a table of users (see figure 4.28) in the non-volatile memory. Each user entry consist of a user id and a password hash. The user id is an two byte unsigned number and the password hash an 32 byte array which contains the SHA256 hash of the users password.

Every file in the NFC tag has its own access rights and depending on the application some or all files can be accessed only after authentication took place.

4.2.4.3.2 Authentication process

Figure 4.29 illustrates the authentication process. The key exchange mechanism to establish a new secure session is very similar to the TLS Pre-Shared-Key authentication. Both parties share a common key which is the combination of user id and password hash.

The password based key derivation function (PBKDF) is used to create new common pseudo random session key. The main inputs to this functions are the common shared key and a random number.

The random number is generated on the security controller when the peer reads out the NFC challenge file. Each read generates a new random number and invalidates the session. Whenever the system looses power or restarts the session is also invalidated.

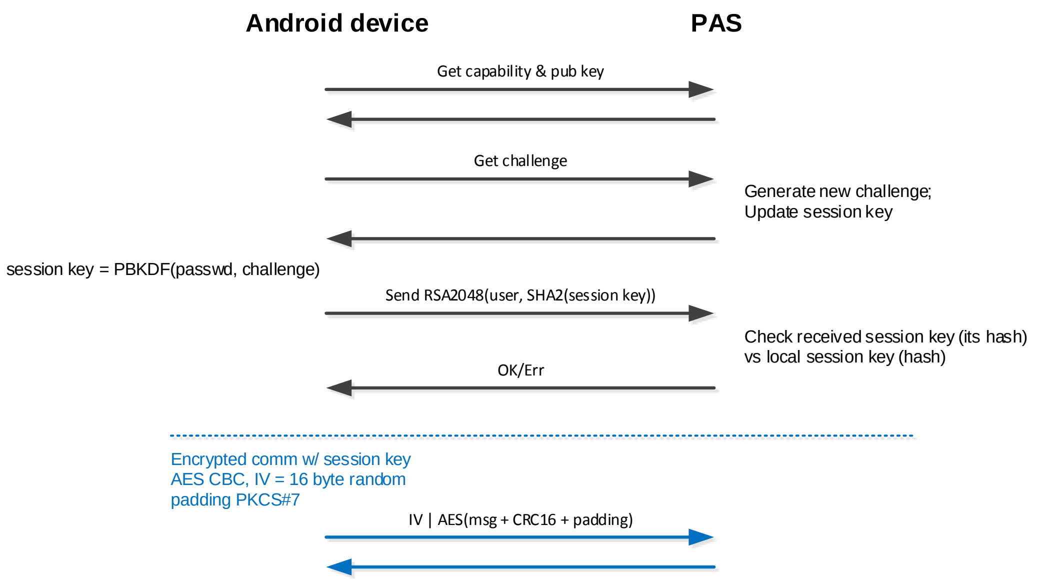
After the peer has read the random number both parties have all the necessary parameters to use the PBKDF in order to generate a new session key.

At last the peer writes the hash of the new session key back into the challenge file to finalize the new session. This last write is additionally encrypted using the RSA public key of the device which is stored in the NFC public key file.

4.2.4.3.3 Secure communication

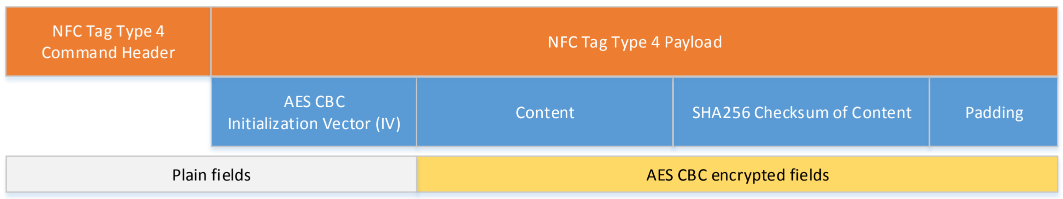
Upon completion of the authentication all files in the tag can be accessed. The payload of all read an write commands on files are encrypted from this point onwards.

The Advanced Encryption Standard (AES) encryption standard in Cipher Block Chaining (CBC) mode is used. Since this is a block cipher the length of the of encrypted fields must be a multiple of the ciphers block size which is 16 bytes. For this reason padding is needed. The CBC mode requires an Initialization Vector (IV).



Android device
PAS
Get capability & pub key
Get challenge
Generate new challenge;
Update session key
session key = PBKDF(passwd, challenge)
Send RSA2048(user, SHA2(session key))
Check received session key (its hash)
vs local session key (hash)
OK/Err
Encrypted comm w/ session key
AES CBC, IV = 16 byte random
padding PKCS#7
IV | AES(msg + CRC16 + padding)

Figure 4.29: NFC interface authentication



NFC Tag Type 4 Command Header
NFC Tag Type 4 Payload
AES CBC Initialization Vector (IV)
Content
SHA256 Checksum of Content
Padding
Plain fields
AES CBC encrypted fields

Figure 4.30: NFC Encrypted communication with Checksum

Figure 4.30 visualizes the structure of an encrypted frame. The frame begins with the plain NFC Tag Type 4 command header (e.g. a read or update command). The payload of the command contains the AES initialization vector, the real content, its SHA256 checksum and a padding.

4.2.5 Transport Layer Security (TLS) Protocol

The Transport Layer Security (TLS) Protocol is widely used in the Internet to secure communications around the world. Approved by the Internet Engineering Task Force (IETF) it is the most commonly used standard. The properties of the protocol are written down in so called Request for Comments (RFC) documents which are released by the IETF.

The application controller has the functionality of a TLS client which allows the system to create a secure connection and exchange encrypted and authenticated data.

4.2.5.1 RFC standards

Transport Layer Security (TLS) is described in RFC 5246 [10] which will be implemented in its current version 1.2. The Elliptic Curve cipher suites are described in RFC 4492 [9] and the X.509 public key infrastructure in RFC 5280.

4.2.5.2 General description

TLS is a stateful protocol which consists of two major phases. The first phase is the so called handshake phase where two peers (a server and a client) negotiate connection parameters in order to establish a secure link. In this phase the peers can authenticate each other by the use of asymmetric public key cryptography. Also a key exchange takes place to create a common shared secret which can be used in the second phase as key for symmetric cryptography. After the handshake is complete application data can be transfered over the secure channel.

In the second phase the protocol provides confidentiality by encrypting application data by using symmetric cryptography. Integrity and authenticity is provided by MAC.

The protocol supports many different cipher key exchange protocols, symmetric ciphers and MAC functions. When a TLS handshake takes place two peers try to agree on a common set of parameters (called cipher suites). A typical TLS implementation only supports a small subset of all possible defined cipher suites. In general it is a good idea to focus on a small number of cryptographically strong cipher suites which provide the level of security needed by the application.

4.2.5.3 Authentication

During the handshake two peers can authenticate themselves to each other. This can be done via multiple different methods. The most commonly used method in the Internet is authentication via RSA certificates. This means a peer possess a public RSA certificate and a corresponding private key. The certificate must be signed by a certificate authority which the other peer thrusts. The peer presenting the public certificate signs a random message with the private in order to prove that the private key is the devices possession and the other peer verifies this signature. In the future RSA certificates might be replaces with ECC certificates. An alternative way of authentication would for example be the use of a pre-shared key.

4.2.5.4 Key exchange

The protocol standard defines multiple ways to conduct a key exchange. A simple and computationally efficient method would be to generate a random key and include in the RSA signed (encrypted) message which proves to possession of the private key. This is called the RSA key exchange method. The disadvantage of this method is that it provides no Perfect Forward Security (PFS). It is obvious that the key is compromised should the RSA encryption be broken.

A better solution is the Diffie-Hellman key exchange method which can be conducted on a prime field or an elliptic curve. This protocol ensures that two parties end up the a shared key but listening parties can only compute the shared key if they solve the discrete logarithm problem which is believed to be very computationally expensive and practically impossible if the key length is chosen large enough.

The Diffie-Hellman protocol can be implemented on top of prime fields as well on elliptic curves. Using elliptic curves provides very good performance in terms of execution time an energy consumption as shown in [13].

4.2.5.5 Application data transmission

After the handshake is complete both peers can transmit application data over a secure channel. All sent packets (records) are appended with a Message Authentication Code (MAC) and encrypted using symmetric cryptography.

4.2.5.5.1 Message Authentication Code (MAC)

The the integrity and authenticity of a message is ensured by a MAC. There are many ways to construct a MAC function, in TLS the most commonly used method is to use a hash function.

A hash function is a function which takes a message as input and produces a pseudo-random message digest (hash). The standardized SHA256 hash function is an example for a capable function to be used to create a MAC.

It is important to choose a cryptographically strong hash function with collision resistance, otherwise an attacker might be able to forge messages. This means that there should not exist two inputs for the hash function that produce the same output (hash).

4.2.5.5.2 Symmetric Cryptography

There are two basic types of symmetric ciphers. Block ciphers and stream ciphers.

Block ciphers take a block of a fixed length and transform it into its encrypted representation and back. An example for a block cipher is the AES with a block size of 16 bytes.

Stream ciphers take a bunch of data of arbitrary length and transform it. This is done by generating a pseudo-random key-stream with a device or module which initiated with a secret key which is shared by the communicating parties. This type of cipher is generally considered to be easy to be implemented in hardware. The encryption used in the mobile phone Global System for Mobile Communications (GSM) standard is a typical example for the use of stream ciphers.

4.2.5.6 Handshake phase overview

The number and format of sent messages depends on the chosen cipher suite, more specifically on the authentication and key exchange method. Figure 4.31 (from RFC 5246) gives an overview of the TLS handshake phase. The following sections describe the involved messages. Messages marked with * are optional depending on the cipher suite.

4.2.5.6.0.1 ClientHello message

The ClientHello message is send by the client to the server as first message to indicate to the server which cipher suites and compression methods are supported. The cipher suite list is sorted by the clients preferences (stronger ciphers suites in the beginning). A random number is included in this message which has the main purpose to prevent replay attacks and can be used as challenge for the signature verification process for some cipher suites. Finally the message can contain a list of extensions depending on the proposed cipher suites and the clients capabilities. For example the so called "Maximum Fragment Length" extension limits the size of future sent TLS records to a chosen maximum. An other example would be the "Signature Algorithm" extension which contains all client supported signature algorithm pairs like RSA-SHA1 (default in TLS version 1.2) or ECDSA-SHA256.

4.2.5.6.0.2 ServerHello message

The ServerHello message is sent by the server to the client as direct answer to the ClientHello message. It contains the chosen cipher suite which is chosen by selecting the first cipher suite from the clients list which is common between server and client. The same selection method is used for the compression methods. In general compression methods should not be used in TLS because it offers a side channel to attackers and there exist practical documented attacks exploiting this weakness. Like the ClientHello message this one also contains a random number for the same purposes. A session id field is part of the message which contains a unique identification number for each client connection of the server. Sessions can be resumed in TLS in order so save resources (computation power on embedded systems). At last there can be a arbitrary number of extensions included in this message.

4.2.5.6.0.3 Certificate (Server) message

If the server wants to authenticate to the client via RSA or ECC certificates this message is sent directly after the ServerHello message. This message contains a list of certificates (also referred as certificate chain) which can be used to verify that the server is a trusted party. This process is called certificate path validation. The first certificate in the chain must be the servers certificate. After that there can follow an arbitrary number of intermediate certificates. Each of them must have been used to sign the previous one. The client verifies the chain until it reaches a certificate which is signed by a trusted root certificate which is contained in the clients certificate storage. If the end of the chain is reached and the last certificate can not be verified then the TLS handshake is aborted with an fatal error. There are many other reasons the path validation process can fail. For example the parsing of a certificate can fail or the validity time frame could be exceeded.

4.2.5.6.0.4 ServerKeyExchange message

This message is sent when using a cipher suite which has the PFS property. This means the key exchange method is for example Diffie-Hellman Ephemeral Key Exchange (DHE) or Elliptic Curve Diffie-Hellman Ephemeral Key Exchange (ECDHE). The PFS property ensures that in the asymmetric encryption is broken and its content is revealed to an attacker then all previous handshakes or sessions are not affected. However once the asymmetric encryption is broken, man in the middle attacks are possible or significantly easier.

If for instance the ECDHE key exchange protocol is used then this message will contain the specification of an elliptic curve and a ephemeral public point (public key) on it.

In addition this message contains a digital signature if server authentication is performed in this handshake. By validating this signature the client makes sure that the server is in possession of the private key corresponding to the sent public certificate.

4.2.5.6.0.5 CertificateRequest message

If this message is sent then the server expects the client to authenticate by sending a valid Certificate message later in the handshake. This message is optional.

4.2.5.6.0.6 ServerHelloDone message

This message concludes the message chain from the server and indicates that there are no more messages. This is needed because depending on the cipher suite the Certificate, ClientKeyExchange and CertificateRequest messages are optional.

4.2.5.6.0.7 Certificate (Client) message

If the server has sent a CertificateRequest message then the client sends this message directly after the ServerHelloDone message. This message has the same structure as the servers Certificate message and is processed the same way.

4.2.5.6.0.8 ClientKeyExchange message

If a cipher suite with PFS is used then this message is send to transport the public key of the client in the key exchange protocol to the server. When using the ECDHE protocol this message contains the clients public point (key) on the curve defined in the ServerKeyChange message.

4.2.5.6.0.9 CertificateVerify message

If the client sends a Certificate message then this message contains a digitally signed data fragment proving that the client is in possession of the private key corresponding to the send public certificate. If no client authentication is done then is message is skipped.

4.2.5.6.0.10 ChangeCipherSpec (Client) message

This message is formally not a handshake message but it is send as part of the handshake to indicate that from now on all TLS records are encrypted with the negotiated cipher suite.

4.2.5.6.0.11 Finished (Client) message

This is the last handshake message sent by the client. The client calculates the digital checksum (hash or message digest) over all previous handshake messages and uses the calulated value is input for the TLS Pseudo Random Function (PRF). The produced value is included in the Finished message. When the server receives this message the same hash and PRF values must be calculated if the handshake was correct and untampered. Trough this process all handshake messages are linked together and man-in-the-middle attacks are significantly harder but theoretically still possible especially if the used hash function is weak or not collision resistant.

4.2.5.6.0.12 ChangeCipherSpec (Server) message

This message has the same format and purpose as the ChangeCipherSpec message sent by the client.

4.2.5.6.0.13 Finished (Server) message

The message has the same format and purpose as the Finished message sent by the client. After this message is verified by client the handshake is complete.

4.2.5.7 Implementation Security Aspects

During implementation of the TLS protocol there are many pitfalls to be considered. Each simple bug can can lead to an attack which can compromise the system. Certain security measures have to be taken and this chapters describes some of them.

4.2.5.7.1 Lucky thirteen attack

When providing a cipher suite with MAC-then-Encrypt and CBC mode the implementation must be protected against the "lucky thirteen attack". This attack exploits a time side channel which is available in careless implementations.

After the decryption of the message, the padding is checked and removed. If the padding is wrong then an error occurred and the connection must be terminated. If the padding was correct the MAC is checked and if the checksum is wrong then the connection is terminated. In theory an attacker should not be able to distinguish these two error cases and the execution time of the software should always be the same no matter which error occurred. The proposed solution in [15] to this problem is that even if the padding is wrong, there must be a MAC check done with some dummy data. This should eliminate the timing side channel.

The following code demonstrates that the padding check consumes always the same execution time.

```
// Check padding
padding = TLS_RX_Buffer[TLS_RECORD_HEADER_LEN + record_len - 1];
if (padding >= AES_BLOCKSIZE)
       success = TLS_ALERT_UNEXPECTED_MESSAGE;
if (AES_BLOCKSIZE + padding + 1 + HASHSIZE > record_len)
       success = TLS_ALERT_UNEXPECTED_MESSAGE;
for (i = 1; i <= AES_BLOCKSIZE; i++)
{
       real_count &= (i <= (padding+1));
       pad_count += real_count *
         (TLS_RX_Buffer[TLS_RECORD_HEADER_LEN+record_len - i] == padding);
}
if (pad_count != (padding + 1))
       success = TLS_ALERT_UNEXPECTED_MESSAGE;
```

4.2.5.7.2 Connection downgrade attack

Many attacks on TLS rely on the possibility in the handshake phase to negotiate a version of the protocol and a cipher suite available on both endpoints. Man-in-the-middle attacks often have the goal to downgrade to a weaker cipher suite or a lower protocol version which is vulnerable to attacks as seen in [5]. A good solution is to allow only the newest protocol version and cipher suites which are deemed to be secure.

4.2.5.7.3 Predictable IV in CBC mode

When using the CBC mode of operation it is mandatory to use an unpredictable, random initialization vector for encryption. If not there is the possibility to perform chosen-plaintext-attacks to reveal unknown parts of a message. The attack is described in [2] and the solution is to use a true random number generator to create an initialization vector. The SLE70 controller contains a hardware true random number generator which is used in this implementation.

4.2.5.7.4 Compression as side channel

The TLS standard supports data compression in the protocol to reduce to used bandwidth. It turns out that this opens a new side channel which can be used to reveal information of an encrypted message. Examples for attacks exploiting this are CRIME and TIME as explained in [22]. The solution is to remove compression support from TLS.

4.2.5.8 TLS combined with BLE

There are multiple ways to use the TLS implementation in combination with BLE.

4.2.5.8.1 GATT server interface

The GATT server can contain a characteristics which is used as data source and sink for TLS data. TLS data can be transfered in this way over a BLE link. This method is not efficient because the Attribute Protocol (ATT) imposes an overhead which can be avoided.

4.2.5.8.2 L2CAP LE Credit Based Channel

A dedicated L2CAP LE Credit Based Channel can be opened using the L2CAP Signaling Protocol. This channel can then be used as tunnel for TLS data. This is the most resource efficient method.

4.2.5.8.3 IPv6 over BLE

The last alternative is a sub-case of the previous one. It proposes using TLS/DTLS on top of TCP/UDP on an IPv6 link. For this method the 6LoWPAN adaption layer is needed between L2CAP and and IPv6 as indicated in figure 2.4 in earlier chapters. This method introduces large overhead which is rewarded with compatibility with existing infrastructure.

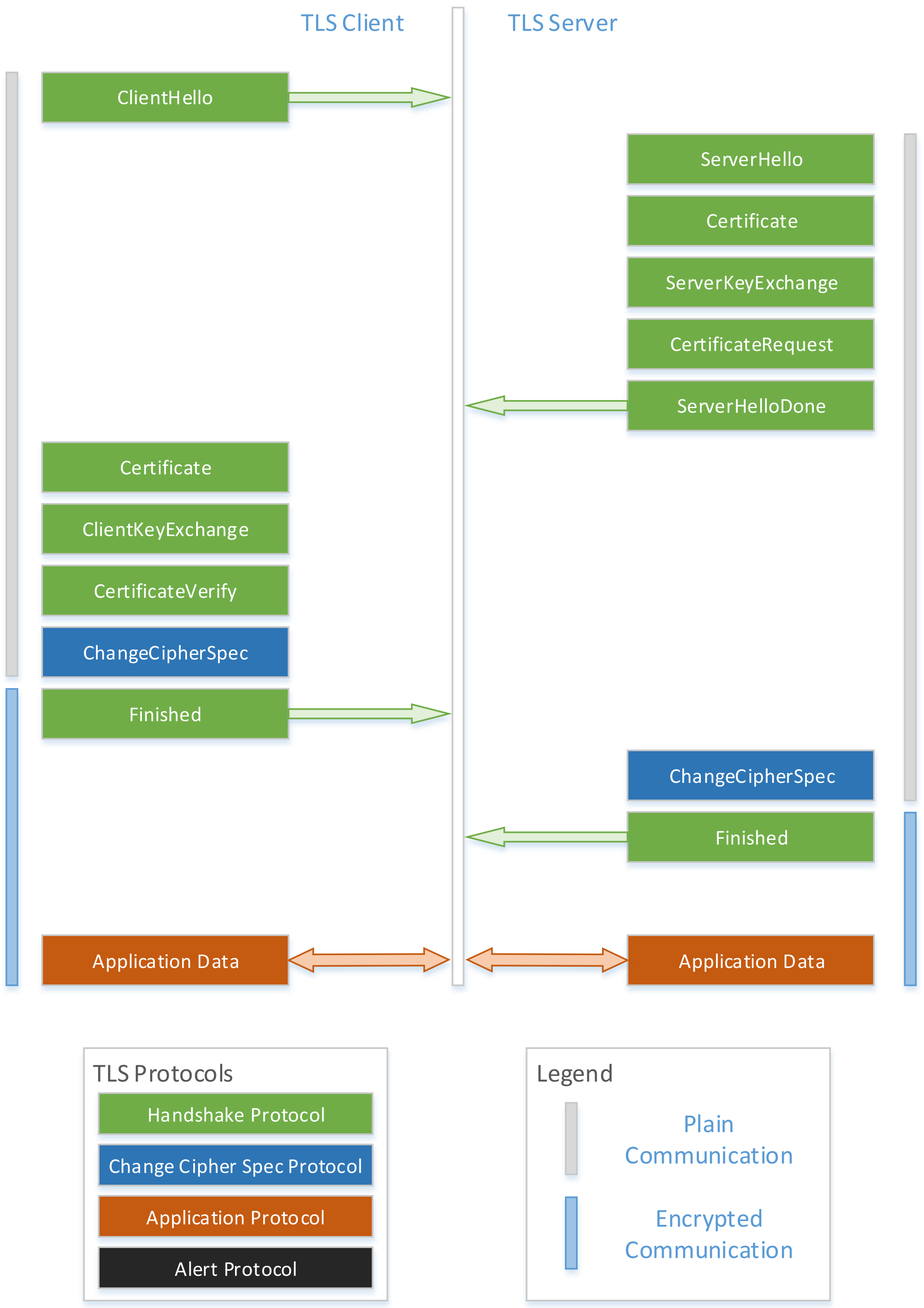


Figure 4.31: TLS handshake overview

4.3 Hardware Implementation

This chapter describes how the components of the personal assistant device are connection to each other. Figure 4.32 shows the schematic of the device.

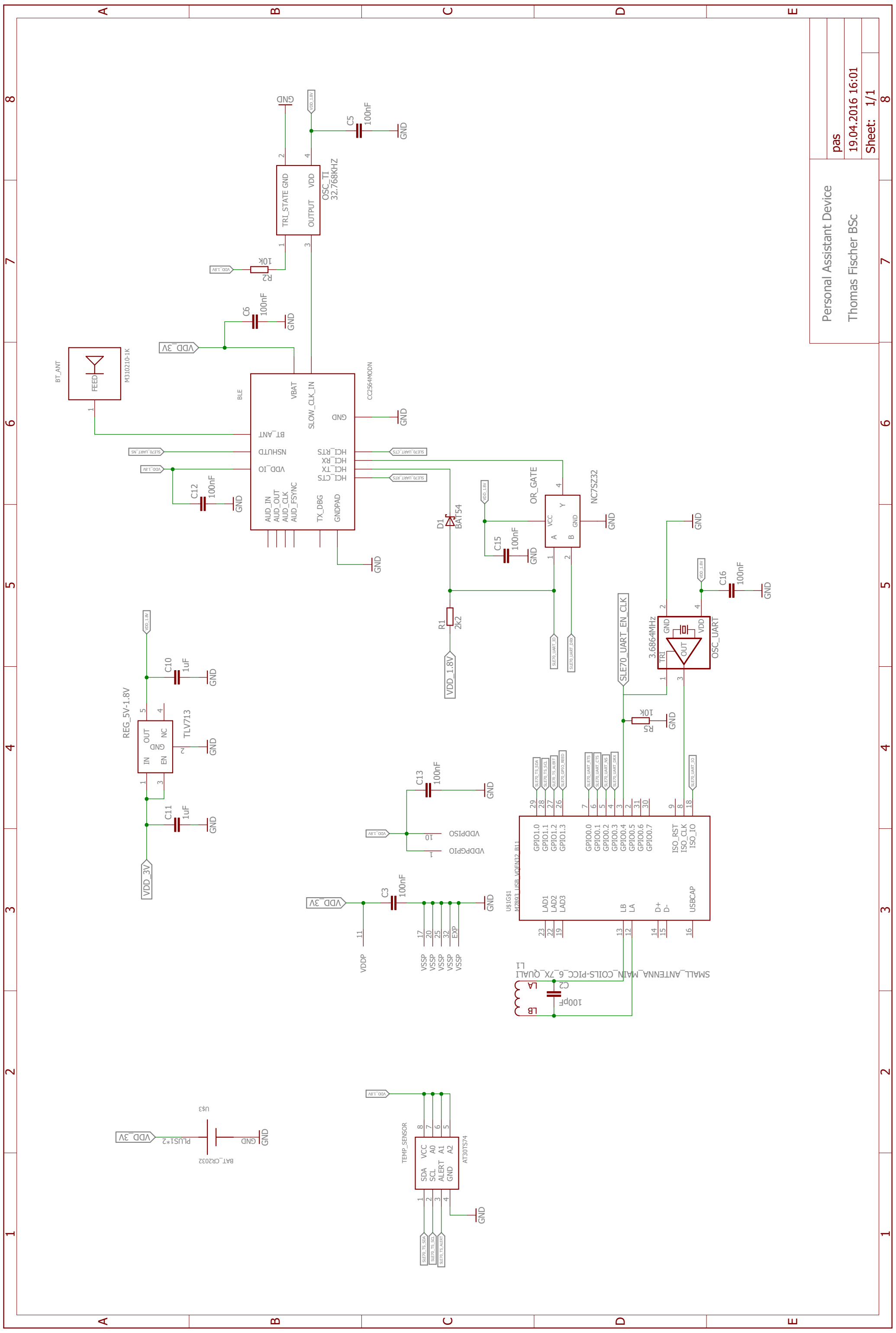


Figure 4.32: Hardware schematic

4.3.1 Bluetooth controller connection (UART)

The Bluetooth host (SLE70) and controller (TI CC2564) are connected via an Universal Asynchronous Receiver Transmitter (UART) interface. On top of this physical transport the Host Controller Interface (HCI) is available. The Infineon SLE70 chip provides a single wire UART module where transmission and reception is done over one wire. The Texas Instruments CC2564 Bluetooth controller on the other hand provides a 4-wire UART module where transmission and reception is done over two separate wires. In order to connect both modules properly some additional components are required as shown in figure 4.33.

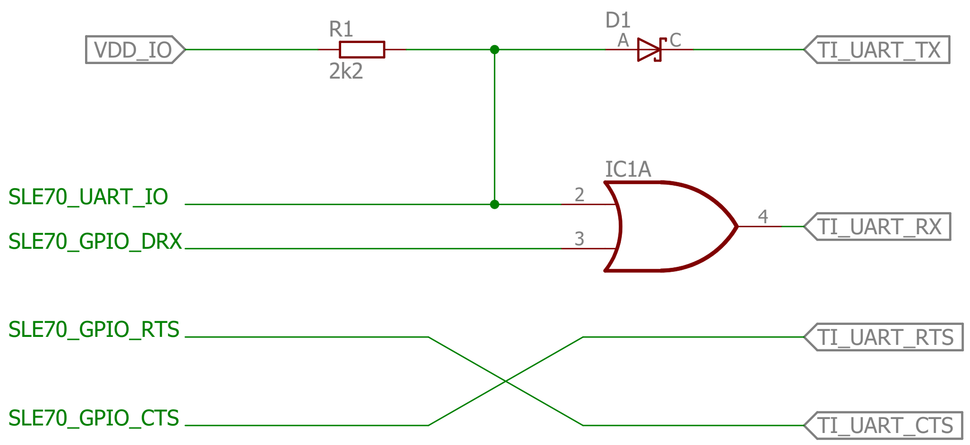


Figure 4.33: UART HCI Connection

Two GPIO pins are used to provide the RTS/CTS hardware flow control functionality. The RTS pin is configured as output pin and the CTS as input pin with pull-up resistor.

The SLE70 IO line is split in two lines, the RX and TX line of the TI chip. A Schottky diode is used to achieve proper behavior of the circuit. When the SLE70 listens on the IO line there are two cases two consider. Either a high or low bit is send. When the TI chip sends a low byte the circuit left of the diode is not affected and the hole voltage drops on the resistor. If a high is send then an infinitesimal current is flowing backwards through the diode and the voltage drop over the diode is very low. This results in a high an the SLE70 IO line. When the SLE70 chip wants two send something the IO line is directly feeded in the TI RX line (ignoring the OR gate for now). Because of the diode the TI TX line is never affected by sending someting.

The OR gate has the purpose to prevent the TI chip from receiving the same data it sends. Whenever the TI chip wants to send something, the ”Disable RX” (DRX) pin is set to high. This means the TI_UART_RX line will be high during this time period due the functionality of the OR gate.

4.3.2 Temperature sensor connection (I2C)

The temperature sensor is connected via I2C interface to the SLE70 controller. A clock and data line are required for this connection.

4.3.3 Reed sensor connection (GPIO)

The reed sensor is connected to the Infineon SLE70 controller via a GPIO pin. A resistor and capacity are used to stabilize the switching currents.

The formula $\tau = RC$ is used to find proper values for the resistor and capacitor. It is assumed that time interval of 20 ms is sufficient. This implies that the values 200 kΩ for the resistor and 100 nF for the capacity are suitable.

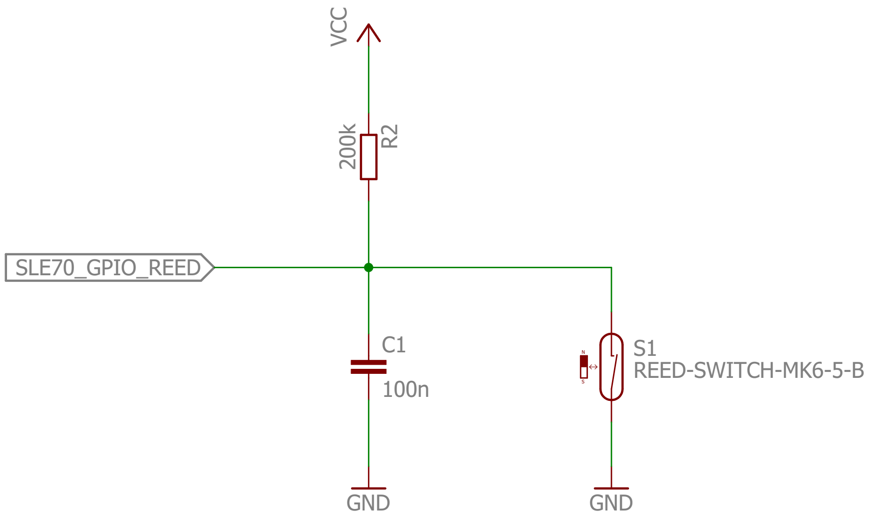


Figure 4.34: Reed sensor connection

4.3.4 Power management (Battery)

The SLE70 chip operates with a voltage of 5V, while the TI chip requires a voltage around 3.3V. The interface IO lines are operating with a voltage of 1.8V. The first prototype of the personal assistant device has an USB connector onboard which allows supplying the board with an external power source (like a USB battery). The USB connector provides a 5V supply with a maximum current of 100 mA (see [12]) to the board, so voltage regulators are required to bring the voltage down to 3.3V and 1.8V.

4.4 Android App Implementation

This application is based on the examples provided from Google in the Android SDK. Code parts are taken from the "Bluetooth LE Gatt" and "Card Reader" samples, as well as from the API guides and reference documentation at the Android SDK website. The created app is a composition of these mentioned parts with adaptations for the Personal Assistant Device and is merely for demonstration purposes.

4.4.1 Device Control Activity

The Device Control Activity contains an user interface with tabs. The SectionsPager-Adapter object is a container for the tabs. Two integers are used to define positions for the tabs needed in this Activity. Two members contain a reference to the BLE service and the temperature value characteristic which is shown to the user. The remaining methods are extracted to extra code listings which follow for simplicity.

```
public class MainActivity extends AppCompatActivity {

    SectionsPagerAdapter mSectionsPagerAdapter;
    final int PAGER_POSITION_NFC = 0;
    final int PAGER_POSITION_BLE = 1;

    private BluetoothLeService mBluetoothLeService;
    BluetoothGattCharacteristic mCharateristicTemperature;

    // class methods ...
}
```

The onCreate method is called when the system creates a new instance of the Activity. All member variables that should persist through the livetime of the Activity are initialized there. The setContentView method initializes the user interface to the layout specified in the function parameter. A basic toolbar which shows the app name is set up using the setSupportActionBar function. The SectionsPagerAdapter is a class which extends FragmentPagerAdapter and is the heart of the tabbed Activity. It is the container object for all the fragments which are used as tabs. Once the SectionsPagerAdapter is created it is put in the user interface with setAdapter method of the corresponding ViewPager container which hosts that object. Finally the TabLayout object is initialized with a reference to the ViewPager in order to know which tabs exist and which tab is currently selected so it can offer a tab selection interface to the user. Finally the registerReceiver function is used to register notifications from a service. This mechanism is explained later in more detail.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    mSectionsPagerAdapter = new
        SectionsPagerAdapter(getSupportFragmentManager());
    ViewPager ViewPager = (ViewPager) findViewById(R.id.container);
    ViewPager.setAdapter(mSectionsPagerAdapter);

    TabLayout tabLayout = (TabLayout) findViewById(R.id.tabs);
    tabLayout.setupWithViewPager(ViewPager);

    Intent gattServiceIntent = new Intent(this, BluetoothLeService.class);
    bindService(gattServiceIntent, mServiceConnection, BIND_AUTO_CREATE);

    registerReceiver(mGattUpdateReceiver, makeGattUpdateIntentFilter());
}
```

The onResume method is called whenever the Activity is shown to the user and has focus. Some dialog window could overlap the Activity or the user could switch to a different app. Whenever the Activity regains focus this method is called. In this implementation the registerReceiver function is used to register for service messages.

```
@Override
protected void onResume() {
    super.onResume();
    registerReceiver(mGattUpdateReceiver, makeGattUpdateIntentFilter());
}
```

The onPause method is called by the system when the Activity looses focus of the user. When this happens there is no need to update the values in the user interface. The unregisterReceiver function is used to stop receiving service notifications.

```
@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(mGattUpdateReceiver);
}
```

The Android system might kill Activities in order to regain free memory. When this happens the onDestroy method is called to allow the Activity to clean up. There is no garantuee that this method is really called. In emergency situations where the system runs out of memory the Activity might be killed instantly without calling this method. The Implementation of this method terminates the bind to the service and set the reference to the service to null to prevent further use.

```
@Override
protected void onDestroy() {
    super.onDestroy();
    unbindService(mServiceConnection);
    mBluetoothLeService = null;
}
```

To setup a proper connection to a service a ServiceConnection member object is created. After the service connection is established the onServiceConnected method is called which sets up a reference to the service which allows calling public methods of the service object.

```
private final ServiceConnection mServiceConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName componentName, IBinder service)
        {
        mBluetoothLeService = ((BluetoothLeService.LocalBinder)
            service).getService();
        if (!mBluetoothLeService.initialize()) {
            Toast.makeText(getBaseContext(), "Unable to initialize Bluetooth",
                Toast.LENGTH_LONG).show();
        }
    }

    @Override
    public void onServiceDisconnected(ComponentName componentName) {
        mBluetoothLeService = null;
    }
};
```

When registering a service broadcast receiver it is required to specify the events of interest. This specifiction is implemented in the makeGattUpdateIntentFilter method. The method creates an IntentFilter which is a container for defined service actions that can occur during the lifetime of this service. For example the the BLE service could broadcast the event ACTION_GATT_CONNECTED after a connection to a remote GATT server has been established.

```
private static IntentFilter makeGattUpdateIntentFilter() {
    final IntentFilter intentFilter = new IntentFilter();
    intentFilter.addAction(BluetoothLeService.ACTION_GATT_CONNECTED);
    intentFilter.addAction(BluetoothLeService.ACTION_GATT_DISCONNECTED);
    intentFilter.addAction(BluetoothLeService.ACTION_GATT_SERVICES_DISCOVERED);
    intentFilter.addAction(BluetoothLeService.ACTION_DATA_AVAILABLE);
    return intentFilter;
}
```

The BroadcastReceiver object is a member of the Activity and must implement the method onReceive which is called whenever an action event from the service is received. All relevant events from the service for the Activity must be handled here. In case of this implementation the user inteface is updated with new values from the service.

```
private final BroadcastReceiver mGattUpdateReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();
        BleFragment bleFragment =
            (BleFragment)mSectionsPagerAdapter.getItem(PAGER_POSITION_BLE);

        if (BluetoothLeService.ACTION_GATT_CONNECTED.equals(action)) {
            bleFragment.setConnectionStatus("Connected");
        } else if (BluetoothLeService.ACTION_GATT_DISCONNECTED.equals(action)) {
            bleFragment.setConnectionStatus("Disconnected");
        } else if
            (BluetoothLeService.ACTION_GATT_SERVICES_DISCOVERED.equals(action)) {
            displayGattServices(mBluetoothLeService.getSupportedGattServices());
        } else if (BluetoothLeService.ACTION_DATA_AVAILABLE.equals(action)) {
            String data = intent.getStringExtra(BluetoothLeService.EXTRA_DATA);
            bleFragment.setTemperature(data);
        }
    }
};
```

The displayGattServices is used to search in a remote GATT server for a characteristic of interest, the temperature characteristic of the personal assistant device. It iterates over all GATT services until it finds a service matching the UUID of the sensor station service of the personal assistant device. Then it iterates all available characteristics of this service until it finds the temperature value characteristic. A reference to this characteristic is then kept in the Activity.

```
private void displayGattServices(List<BluetoothGattService> gattServices) {
    if (gattServices == null) return;

    for (BluetoothGattService gattService : gattServices) {
        String uuid = gattService.getUuid().toString();

        if (uuid.equals(SampleGattAttributes.UUID_SERVICE_SENSOR_STATION))
        {
            List<BluetoothGattCharacteristic> gattCharacteristics =
                    gattService.getCharacteristics();

            for (BluetoothGattCharacteristic gattCharacteristic :
                gattCharacteristics) {
                String uuid_c = gattCharacteristic.getUuid().toString();

                if (uuid_c.equals(SampleGattAttributes.UUID_CHAR_TEMP_SENSOR))
                    mCharateristicTemperature = gattCharacteristic;
            }
        }
      }
}
```

The SectionsPagerAdapter extends the FragmentPagerAdapter by implementing methods to specify which tabs are shown.

```
public class SectionsPagerAdapter extends FragmentPagerAdapter {

        public SectionsPagerAdapter(FragmentManager fm) {
            super(fm);
        }

        @Override
        public Fragment getItem(int position) {
            if (position == PAGER_POSITION_NFC)
                return new BleFragment();
            if (position == PAGER_POSITION_BLE)
                return new BleFragment();
            return null;
        }

        @Override
        public int getCount() {
            return 2;
        }
        @Override
        public CharSequence getPageTitle(int position) {
            switch (position) {
                case PAGER_POSITION_NFC:
                    return "NFC";
                case PAGER_POSITION_BLE:
                    return "BLE";
            }
            return null;
        }
    }
```

4.4.2 BLE Fragment

The BLE fragment is used to display the connection status and current temperature value to the user. Since the Activity is bound to the BLE Service it receives all the service broadcast. This includes the information to be displayed in this fragment. The Activity uses the fragments interface to make values available to the user.

```
public static class BleFragment extends Fragment {
    public BleFragment() {
    }

    public void setTemperature(String temp) {
        TextView textViewTemp =
            (TextView)getView().findViewById(R.id.textViewTemp);
        textViewTemp.setText(temp, TextView.BufferType.EDITABLE);
    }

    public void setConnectionStatus(String status) {
        TextView textViewStatus =
            (TextView)getView().findViewById(R.id.textViewStatus);
        textViewStatus.setText(status, TextView.BufferType.EDITABLE);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View rootView = inflater.inflate(R.layout.fragment_ble_control,
            container, false);
        return rootView;
    }
}
```

4.4.3 BLE Service

Based on the BLE example from the Google SDK this service interacts with the Android system and the Bluetooth stack contained in it. The service allows components like an Activiy to bind itself to the servive. A bound service lives as long there are components bound to it and is then shutdown by the system.

The BluetoothLeService class extends the Android Service class and implementes the method onBind which returns an IBinder interface object. The broadcastUpdate method is used to send notifications to all components bound to the service with an registered

BroadcastReceiver. The initialize function is usally called by the bound component after the service bound has been created and makes sure that a Bluetooth adapter is available on this device. The connect method creates a connection to the Bluetooth GATT server on a remote devices and the disconnect method terminates that link. When a bound component is finished doing its work or terminated by system the onUnbind method is called to inform the service of the lost bound. When that happens the implementation in this class calls the close method to ensure that all system resources are properly freed before finishing this service.

It is important to note that GATT functions like getSupportedGattServices and readCharacteristic are non-blocking methods. These methods place a command in the queue of the Android Bluetooth system and return afterwards. Once the system has fulfilled the request a callback method is called. This service has a BluetoothGattCallback member which handles these callbacks properly by using the broadcastUpdate method to send notifications.

```java
public class BluetoothLeService extends Service {
    private BluetoothManager mBluetoothManager;
    private BluetoothAdapter mBluetoothAdapter;
    private String mBluetoothDeviceAddress;
    private BluetoothGatt mBluetoothGatt;
    private int mConnectionState = STATE_DISCONNECTED;

    private static final int STATE_DISCONNECTED = 0;
    private static final int STATE_CONNECTING = 1;
    private static final int STATE_CONNECTED = 2;

    public final static UUID UUID_PAS_CHARACTERISTIC_TEMPERATURE =
        UUID.fromString(SampleGattAttributes.UUID_PAS_CHARACTERISTIC_TEMP_SENSOR);

    private final BluetoothGattCallback mGattCallback = new
        BluetoothGattCallback() {
        @Override
        public void onConnectionStateChange(BluetoothGatt gatt, int status, int
            newState) {
            String intentAction;
            if (newState == BluetoothProfile.STATE_CONNECTED) {
                intentAction = ACTION_GATT_CONNECTED;
                mConnectionState = STATE_CONNECTED;
                broadcastUpdate(intentAction);
            } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
                intentAction = ACTION_GATT_DISCONNECTED;
                mConnectionState = STATE_DISCONNECTED;
                broadcastUpdate(intentAction);
            }
        }

        @Override
        public void onServicesDiscovered(BluetoothGatt gatt, int status) {
            if (status == BluetoothGatt.GATT_SUCCESS) {
                broadcastUpdate(ACTION_GATT_SERVICES_DISCOVERED);
            }
        }

        @Override
        public void onCharacteristicRead(BluetoothGatt gatt,
            BluetoothGattCharacteristic characteristic, int status) {
            if (status == BluetoothGatt.GATT_SUCCESS) {
                broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
            }
        }

        @Override
        public void onCharacteristicChanged(BluetoothGatt gatt,
            BluetoothGattCharacteristic characteristic) {
            broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
        }
    };

    private void broadcastUpdate(final String action) {
        final Intent intent = new Intent(action);
        sendBroadcast(intent);
    }

    private void broadcastUpdate(final String action, final
        BluetoothGattCharacteristic characteristic) {
        final Intent intent = new Intent(action);
        if (UUID_PAS_CHARACTERISTIC_TEMPERATURE.equals(characteristic.getUuid()))
            {
            final int heartRate =
                characteristic.getIntValue(BluetoothGattCharacteristic.FORMAT_UINT8,
                0);
            intent.putExtra(EXTRA_DATA, String.valueOf(heartRate));
            sendBroadcast(intent);
        }
    }

    public class LocalBinder extends Binder {
        BluetoothLeService getService() {
            return BluetoothLeService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    @Override
    public boolean onUnbind(Intent intent) {
        close();
        return super.onUnbind(intent);
    }

    private final IBinder mBinder = new LocalBinder();

    public boolean initialize() {
        if (mBluetoothManager == null) {
            mBluetoothManager = (BluetoothManager)
                getSystemService(Context.BLUETOOTH_SERVICE);
            if (mBluetoothManager == null) {
```

```
            return false;
        }
    }

    mBluetoothAdapter = mBluetoothManager.getAdapter();
    if (mBluetoothAdapter == null) {
        return false;
    }

    return true;
}

public boolean connect(final String address) {
    if (mBluetoothAdapter == null || address == null) {
        return false;
    }

    if (mBluetoothDeviceAddress != null &&
        address.equals(mBluetoothDeviceAddress) && mBluetoothGatt != null) {
        if (mBluetoothGatt.connect()) {
            mConnectionState = STATE_CONNECTING;
            return true;
        } else {
            return false;
        }
    }

    final BluetoothDevice device = mBluetoothAdapter.getRemoteDevice(address);
    if (device == null) {
        return false;
    }
    mBluetoothGatt = device.connectGatt(this, false, mGattCallback);
    refreshDeviceCache(mBluetoothGatt);
    mBluetoothDeviceAddress = address;
    mConnectionState = STATE_CONNECTING;
    return true;
}

public void disconnect() {
    if (mBluetoothAdapter == null || mBluetoothGatt == null) {
        return;
    }
    mBluetoothGatt.disconnect();
}

public void close() {
    if (mBluetoothGatt == null) {
        return;
    }
    mBluetoothGatt.close();
    mBluetoothGatt = null;
}

public void readCharacteristic(BluetoothGattCharacteristic characteristic) {
    if (mBluetoothAdapter == null || mBluetoothGatt == null) {
        return;
        }
        mBluetoothGatt.readCharacteristic(characteristic);
    }

    public List<BluetoothGattService> getSupportedGattServices() {
        if (mBluetoothGatt == null) return null;
        return mBluetoothGatt.getServices();
    }
}
```

4.4.4 NFC Fragment

The NFC fragement is a user interface to interact with the personal assistant device over NFC. Its main purpose is do support pairing the phone with the device in order to create a BLE connection.

When to phone is moved in close proximity the Device Control Activity should be automatically started. To achieve this the App is registering for NFC intends. The following code is added to the activities onCreate function. It checks if the app was automatically started by moving the phone to the device and keeps a reference to the NFC tag if this happened.

```
if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction())) {
  tag = (Tag)getIntent().getParcelableExtra(NfcAdapter.EXTRA_TAG);
  mNfcFragment.setTag(tag);
}
```

The following code is used to send a frame of data to the device over NFC and receive the answer which is in this case the Bluetooth address of the device.

```
if (mIsoDep == null)
    mIsoDep = IsoDep.get(mTag);

if (!mIsoDep.isConnected()) {
    mIsoDep.connect();
}

byte[] ans = mIsoDep.transceive(enable_cmd);

if (ans.length == SW_LEN + BDADDR_LEN && ans[0] == 0x90 && ans[1] == 0x00)
{
    byte[] bd_addr = Arrays.copyOfRange(ans, SW_LEN, SW_LEN + BDADDR_LEN);
    String bd_addr_str = byteArrayToString(bd_addr);
    EditText editText = (EditText)rootView.findViewById(R.id.editTextBleAddr);
    editText.setText(bd_addr_str, TextView.BufferType.EDITABLE);

    Toast.makeText(rootView.getContext(), "BLE activated.",
        Toast.LENGTH_LONG).show();
}
```

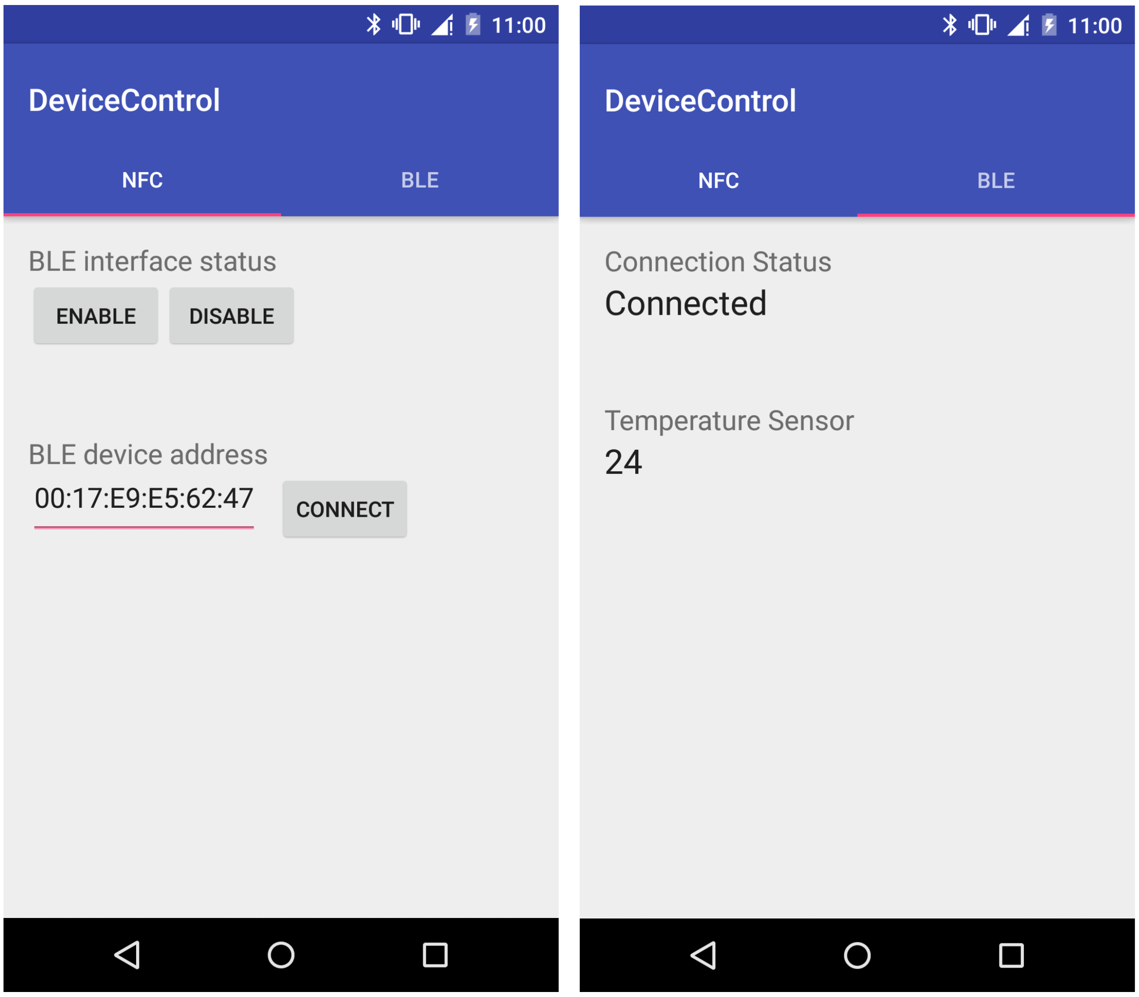


Figure 4.35: Android App Tabs

4.5 Testing

This section will briefly describe how important components of this project have been tested.

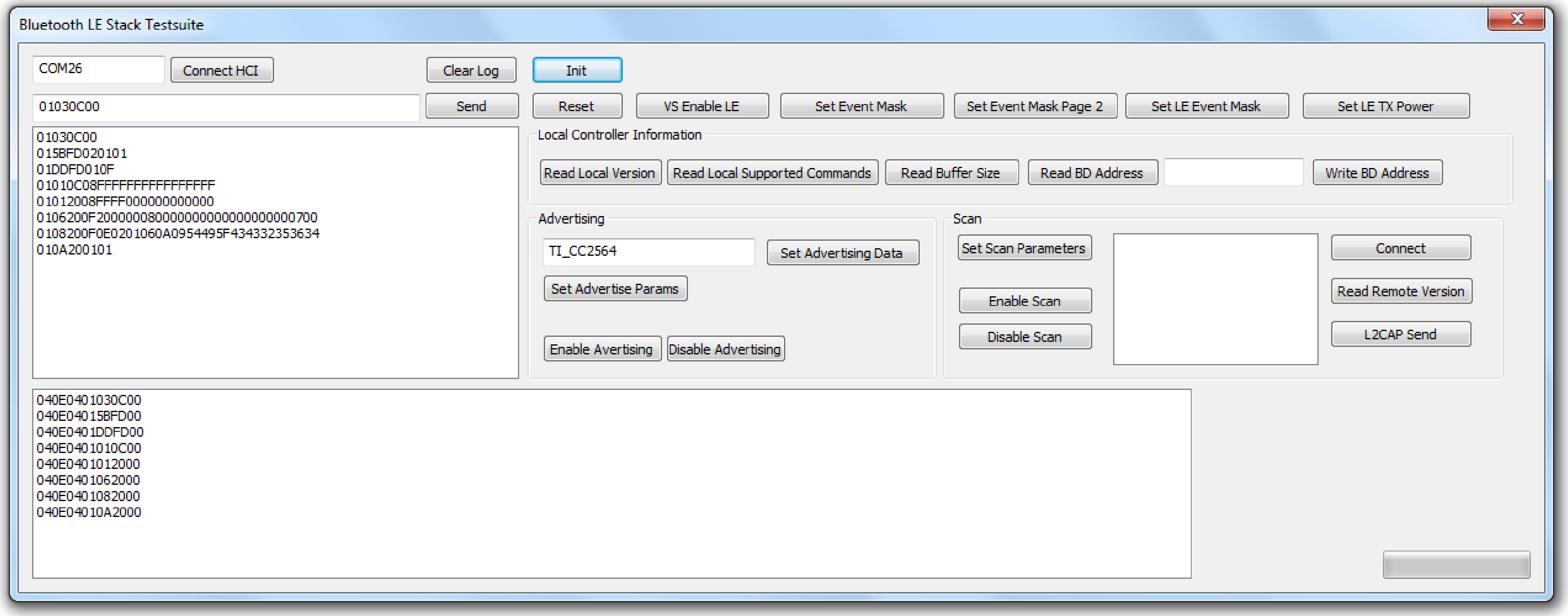
4.5.1 Bluetooth Low Energy Stack Testing

The BLE stack has been developed with multi-platform compatibility with allows running the code on Microsoft Windows. A simple graphical user interface has been created to manually test the functionality.

4.5.2 TLS Implementation Testing

Creating a standard compliant implementation of the TLS protocol also includes testing against other known TLS implementations. This implementation has been tested against OpenSSL and embedTLS (former PolarSSL).

Figure 4.37 shows the test setup which allows testing the implementation against mbedTLS. The data shown in the figure belongs to a TLS handshake.

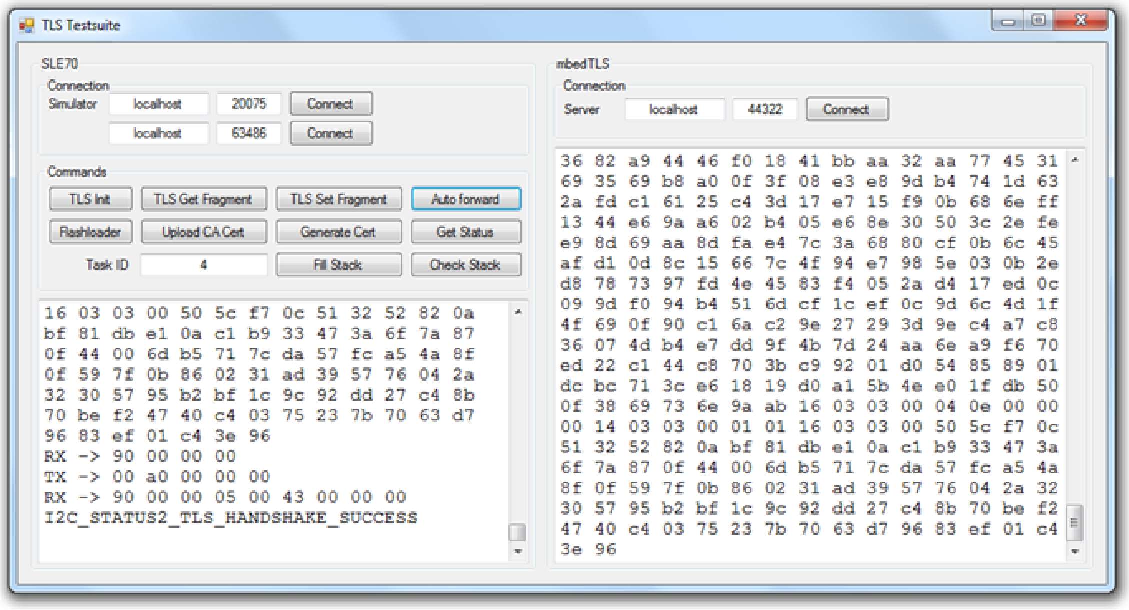


```
01030C00
```

```
01030C00
015BFD020101
01DDFD010F
01010C08FFFFFFFFFFFFFFFF
01012008FFFF000000000000
0106200F200000080000000000000000000700
0108200F0E0201060A0954495F434332353634
010A200101
```

```
040E0401030C00
040E04015BFD00
040E0401DDFD00
040E0401010C00
040E0401012000
040E0401062000
040E0401082000
040E04010A2000
```

Figure 4.36: BLE Test Suite



Simulator localhost 20075
localhost 63486
Task ID 4

```
16 03 03 00 50 5c f7 0c 51 32 52 82 0a
bf 81 db e1 0a c1 b9 33 47 3a 6f 7a 87
0f 44 00 6d b5 71 7c da 57 fc a5 4a 8f
0f 59 7f 0b 86 02 31 ad 39 57 76 04 2a
32 30 57 95 b2 bf 1c 9c 92 dd 27 c4 8b
70 be f2 47 40 c4 03 75 23 7b 70 63 d7
96 83 ef 01 c4 3e 96
RX -> 90 00 00 00
TX -> 00 a0 00 00 00
RX -> 90 00 00 05 00 43 00 00 00
I2C_STATUS2_TLS_HANDSHAKE_SUCCESS
```

Server localhost 44322

```
36 82 a9 44 46 f0 18 41 bb aa 32 aa 77 45 31
69 35 69 b8 a0 0f 3f 08 e3 e8 9d b4 74 1d 63
2a fd c1 61 25 c4 3d 17 e7 15 f9 0b 68 6e ff
13 44 e6 9a a6 02 b4 05 e6 8e 30 50 3c 2e fe
e9 8d 69 aa 8d fa e4 7c 3a 68 80 cf 0b 6c 45
af d1 0d 8c 15 66 7c 4f 94 e7 98 5e 03 0b 2e
d8 78 73 97 fd 4e 45 83 f4 05 2a d4 17 ed 0c
09 9d f0 94 b4 51 6d cf 1c ef 0c 9d 6c 4d 1f
4f 69 0f 90 c1 6a c2 9e 27 29 3d 9e c4 a7 c8
36 07 4d b4 e7 dd 9f 4b 7d 24 aa 6e a9 f6 70
ed 22 c1 44 c8 70 3b c9 92 01 d0 54 85 89 01
dc bc 71 3c e6 18 19 d0 a1 5b 4e e0 1f db 50
0f 38 69 73 6e 9a ab 16 03 03 00 04 0e 00 00
00 14 03 03 00 01 01 16 03 03 00 50 5c f7 0c
51 32 52 82 0a bf 81 db e1 0a c1 b9 33 47 3a
6f 7a 87 0f 44 00 6d b5 71 7c da 57 fc a5 4a
8f 0f 59 7f 0b 86 02 31 ad 39 57 76 04 2a 32
30 57 95 b2 bf 1c 9c 92 dd 27 c4 8b 70 be f2
47 40 c4 03 75 23 7b 70 63 d7 96 83 ef 01 c4
3e 96
```

Figure 4.37: TLS Test Suite

4.5.3 Prototype

A first prototype of the personal assistant device was constructed as seen in figure 4.38. The top board hosts the application controller. The middle board contains the sensors and the circuit necessary to connect the UART. Finally the bottom board hosts the Bluetooth controller.

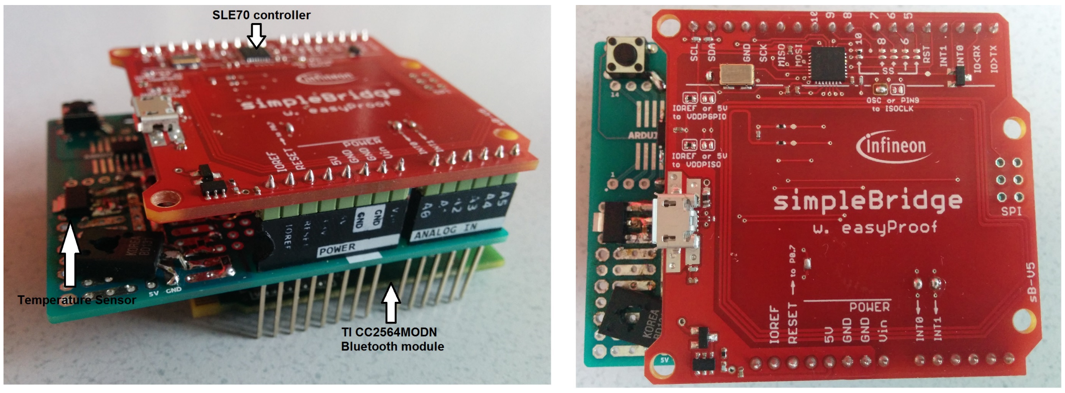


Figure 4.38: Personal Assistant Device Prototype

Chapter 5

Conclusion

Bluetooth Low Energy is supported by many smartphones today and will play an important role in the Internet of Things (IoT). Understanding the capabilities of this interface and its properties is an important precondition for many other projects.

The personal assistant device provides a Bluetooth stack with all mandatory features which allows supporting many possible use cases. This includes using the device as transceiver, beacon and observer. Two use cases where the device is used as mobile temperature sensor station and as door watchdog have been demonstrated by building a prototype of the system. The NFC interface can be used for user-friendly pairing with the device. An other aspect is the growing importance of security required in devices. The implemented TLS client uses the crypto hardware modules of the SLE70 chip and provides excellent performance for the designated use cases. This part of the software might also be reused on other IoT related projects.

5.1 Outlook

Future work could include performing tests to analyze the performance and energy consumption in more detail. By possessing the whole source code of the application controller it is possible to perform code optimizations to extend the lifetime of the device. The Bluetooth stack might be used in many other BLE related projects. There are many optional features described in the Bluetooth standard which are not yet implemented. Some of them might be considered as useful will be added in the future.

The TLS implementation can be extended with more cipher suites depending on the applications needs. One interesting encryption algorithm is ASCON [4], which was developed by the institute IAIK at TU Graz which can be used as alternative to the AES algorithm. In general the project was a success overall and more projects in this are will follow.

Appendix A

Acronyms

MCU	Micro Controller Unit
IDE	Integrated Development Environment
GPIO	General Purpose Input Output
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
BLE	Bluetooth Low Energy
HCI	Host Controller Interface
L2CAP	Link Control and Adaption Protocol
CID	Channel Identification Number
MTU	Maximum Transmission Unit
PDU	Protocol Data Unit
LE PSM	Low Energy Protocol Service Multiplexer
ATT	Attribute Protocol
SMP	Security Manager Protocol
LTK	Long Term Key
CSRK	Connection Signature Resolving Key
IRK	Identity Resolving Key
ECDH	Elliptic Curve Diffie-Hellman
GATT	Generic Attribute Profile
UUID	Universally Unique Identifier
GAP	Generic Access Profile
NFC	Near Field Communication
I2C	Inter-Integrated Circuit
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
AES	Advanced Encryption Standard
IoT	Internet of Things
RFI	Radio Frequency Interface
USB	Universal Serial Bus
GSM	Global System for Mobile Communications
IETF	Internet Engineering Task Force
RFC	Request for Comments
MAC	Message Authentication Code
TLS	Transport Layer Security
PFS	Perfect Forward Security

PRF	Pseudo Random Function
CBC	Cipher Block Chaining
IV	Initialization Vector
DHE	Diffie-Hellman Ephemeral Key Exchange
ECDHE	Elliptic Curve Diffie-Hellman Ephemeral Key Exchange

Bibliography

[1] *NFC Forum Type 4 Tag Operation Specification 2.0.* nfc-forum.org, 2010.

[2] Gregory V. Bard. Vulnerability of SSL to Chosen-Plaintext Attack. pages 1–10, 2004.

[3] Bluetooth SIG. *Bluetooth Specification Version 4.2.* 2014.

[4] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, Martin Schlaeffer. Ascon v1.1 - Submission to the CAESAR Competition. ascon.iaik.tugraz.at, 2015.

[5] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. pages 1–12.

[6] G. M. Shrestha and J. Imtiaz and J. Jasperneite. An optimized OPC UA transport profile to bringing Bluetooth Low Energy Device into IP networks. In *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–5, Sept 2013.

[7] Haolin Wang and Minjun Xi and Jia Liu and Canfeng Chen. Transmitting IPv6 packets over Bluetooth low energy based on BlueZ. In *Advanced Communication Technology (ICACT), 2013 15th International Conference on*, pages 72–77, Jan 2013.

[8] Hongwei Du. NFC Technology: Today and Tomorrow. International Journal of Future Computer and Communication, Vol. 2, No. 4, August 2013, 2013.

[9] IETF. *RFC4492 - Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS).* tools.ietf.org, 2006.

[10] IETF. *RFC5246 - The Transport Layer Security (TLS) Protocol Version 2.1.* tools.ietf.org, 2008.

[11] J. Yim and S. Kim and N. K. Kim and Y. B. Ko. IPv6 based real-time acoustic data streaming service over Bluetooth Low Energy. In *Communications, Computers and Signal Processing (PACRIM), 2015 IEEE Pacific Rim Conference on*, pages 269–273, Aug 2015.

[12] Jan, Axelson. *USB Complete - Everything You Need To Develop Custom USB Peripherals.* lakeview research llc, 3. aufl. edition, 2005.

[13] M. Koschuch and M. Hudler and M. Krüger. Performance evaluation of the TLS handshake in the context of embedded devices. In *Data Communication Networking (DCNET), Proceedings of the 2010 International Conference on*, pages 1–10, July 2010.

[14] Marcus Janke, Dr. Peter Laakmann. Attacks on Embedded Devices. Embedded World Conference Nurenberg, 2016.

[15] N. J. Al Fardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540, May 2013.

[16] P. Burzacca and M. Mircoli and S. Mitolo and A. Polzonetti. iBeacon technology that will make possible Internet of Things. In *Software Intelligence Technologies and Applications International Conference on Frontiers of Internet of Things 2014, International Conference on*, pages 159–165, Dec 2014.

[17] R. Couto and J. Leal and P. M. Costa and T. Galvão. Exploring Ticketing Approaches Using Mobile Technologies: QR Codes, NFC and BLE. In *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on*, pages 7–12, Sept 2015.

[18] Robert Davidson, Akiba, Carles Cufi, Kevin Townsend. *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking.* O'Reilly Media, Inc., 2014.

[19] Robin Heydon. *Bluetooth Low Energy: The Developer's Handbook.* Prentice Hall, 2012.

[20] S. Noguchi and M. Niibori and E. Zhou and M. Kamada. Student Attendance Management System with Bluetooth Low Energy Beacon and Android Devices. In *Network-Based Information Systems (NBiS), 2015 18th International Conference on*, pages 710–713, Sept 2015.

[21] T. Zhang and J. Lu and F. Hu and Q. Hao. Bluetooth low energy for wearable sensor-based healthcare systems. In *Healthcare Innovation Conference (HIC), 2014 IEEE*, pages 251–254, Oct 2014.

[22] Tal Be'ery and Amichai Shulman. (TLS attacks) A Perfect CRIME? Only TIME Will Tell. pages 1–33, 2013.

[23] Vincent Himpe. *Mastering the I2C Bus.* Publit Elektor, 2011.

[24] Z. M. Lin and C. H. Chang and N. K. Chou and Y. H. Lin. Bluetooth Low Energy (BLE) based blood pressure monitoring system. In *Intelligent Green Building and Smart Grid (IGBSG), 2014 International Conference on*, pages 1–4, April 2014.