HAW HAMBURG

BACHELORTHESIS
Martin Pyka

Entwicklung und Implementierung eines asynchronen Kommunikationsprotokolls für Bluetooth Low Energy

FAKULTÄT TECHNIK UND INFORMATIK
Department Informations- und Elektrotechnik

Faculty of Computer Science and Engineering
Department of Information and Electrical Engineering

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Martin Pyka

Entwicklung und Implementierung eines asynchronen Kommunikationsprotokolls für Bluetooth Low Energy

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Elektro- und Informationstechnik*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Karl-Ragmar Riemschneider
Zweitgutachter: Prof. Dr.-Ing. Pawel Buczek

Eingereicht am: 3. März 2021

Martin Pyka

Thema der Arbeit

Entwicklung und Implementierung eines asynchronen Kommunikationsprotokolls für Bluetooth Low Energy

Stichworte

Bluetooth Low Energy, asynchron, GATT, Kommunikationsprotokoll

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Untersuchung und der Entwicklung eines asynchronen Kommunikationsprotokolls für den Standard Bluetooth Low Energy, welches parallel nutzbare Lese- und Schreibprozeduren beinhalten soll.

Martin Pyka

Title of Thesis

Development and implementation of an asynchronous communication protocol for Bluetooth Low Energy

Keywords

Bluetooth Low Energy, asynchronous, GATT, communication protocol

Abstract

This thesis deals with the investigation and development of an asynchronous communication protocol, which should contain read and write operations that can be used in parallel.

Inhaltsverzeichnis

1 Einführung

1.1 Ziel und Motivation dieser Arbeit

Die Verbindung von Haushaltsgeräten mit mobilen Geräten, wie Smartphones, ist heutzutage nichts Außergewöhnliches mehr. Auf den mobilen Geräten werden Informationen, beispielsweise Kapazitäten des Akkus, Füllstände oder Nutzungsdauer der Haushaltsgeräte, angezeigt. Für die Verbindung von Haushaltsgerät und mobilem Gerät existieren, verschiedene Verbindungsarten.

Da Haushaltsgeräte immer häufiger von Akkumulatoren versorgt werden, liegt die Herausforderung darin, gute Leistungsdaten bei einem geringen Stromverbrauch zu liefern. Dafür wurde der Standard *Bluetooth Low Energy* entwickelt. Eine Besonderheit beim Standard Bluetooth Low Energy ist das Profil GATT, dieses Profil setzt auf eine synchrone und serielle Kommunikation.

Durch den modularen Aufbau der Geräte werden häufig mehrere Mikrocontroller in einem Haushaltsgerät verbaut. Für die Kommunikation zwischen den Mikrocontrollern kommen serielle Bussysteme zum Einsatz. Diese verursachen zum Teil hohe Latenzzeiten beim Datenzugriff. In dieser Arbeit wird untersucht, welche Einschränkungen durch die Nutzung des Standards Bluetooth Low Energy mit derartigen Systemen entstehen können.

Mit den Ergebnissen wird ein Kommunikationsprotokoll entwickelt und implementiert, welches asynchrone Kommunikation für den Standard Bluetooth Low Energy erlaubt.

Die Arbeit gliedert sich in die folgenden Kapitel. Die, für diese Arbeit wichtigen Grundlagen, befinden sich im Kapitel 2. Die Analyse der Ebene GATT des Standards Bluetooth Low Energy wird, hinsichtlich der Kommunikation der Unterprozeduren, im Kapitel 3 untersucht. Aus der Analyse entsteht ein asynchrones Verfahren, welches für das Kommunikationsprotokoll genutzt wird, dies wird im Kapitel 4 erklärt. Anforderungen an

die Implementierung des Kommunikationsprotokolls ist im Abschnitt 5.1 enthalten. Eine Implementierung des Kommunikationsprotokolls ist im Kapitel 5 beschrieben. Die Verifikation des implementierten Kommunikationsprotokolls ist im Kapitel 6 enthalten. Schlussendlich die Zusammenfassung der Ergebnisse, die offenen Punkte und Probleme, die Ausblicke, sowie das Fazit befinden sich im Kapitel 7.

2 Grundlagen

Dieses Kapitel behandelt die Grundlagen, die für diese Arbeit wichtig sind. Dafür wird zuerst der Standard Bluetooth Low Energy grob mit den einzelnen Ebenen erklärt und zum Schluss wird detaillierter auf die Ebene GATT eingegangen. Diese Ebene mit ihren Prozeduren ist die Grundlage für das Kommunikationsprotokoll.

2.1 Grundlagen zum Standard Bluetooth Low Energy

Der Standard Bluetooth Low Energy ist ein Funk-Standard, der im lizenzfreiem 2,4 GHz-Band arbeitet [5, Vol 1 | Part A | 1.2]. Spezifiziert von der Bluetooth Special Interest Group (SIG) ist der Standard Bluetooth Low Energy eine auf niedrigen Stromverbrauch optimierte drahtlose Kommunikationstechnologie [20, 3.3]. Die technischen Daten können der Bluetooth Core Spezifikation entnommen werden [5, Vol 1 | Part A | 1.2].

Der Standard Bluetooth Low Energy ist als Protokollstack aufgebaut, der aus mehreren Ebenen besteht. Die Ebenen übernehmen unterschiedliche Aufgaben. In der Abbildung 2.1 sind die Ebenen skizziert und nachfolgend werden diese kurz erläutert.

2.1.1 Ebene Physical Layer

Die Ebene Physical Layer (PHY) ist für das Senden und Empfangen von elektromagnetischen Wellen zuständig [5]. Im Allgemeinen können Informationen mit Hilfe elektromagnetischer Wellen durch Variieren der Amplitude, Frequenz oder Phase der Welle innerhalb eines bestimmten Frequenzbandes übertragen werden.

Beim Standard Bluetooth Low Energy wird die Gaußsche Frequenzumtastung (GFSK) als Modulationsschema verwendet [12]. Nach den Autoren J.Liu und M. Cai, ist das Modulatuionsschema GFSK weit verbreitet, als Gründe nennen Sie das es eine hervorragende Leistungs- und Spektrumsausnutzung bietet. In Ihrem Artikel [13] untersuchen

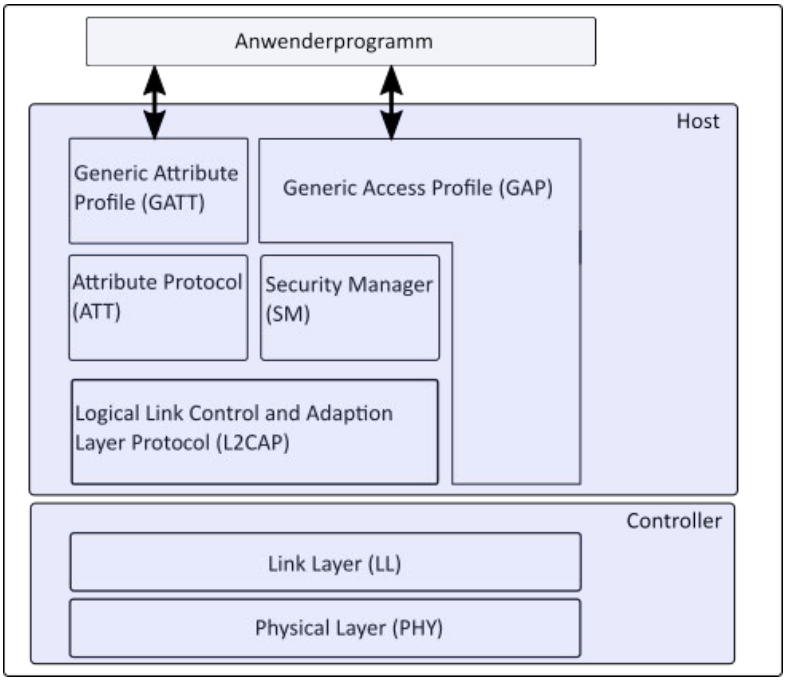

Abbildung 2.1: Protokollstack des Standards Bluetooth Low Energy, die Abbildung ist modifiziert nach Nordic Semiconductor, 2019, [19]

Sie das Modulationsschema, und kommen zum Ergebnis, dass die Signalqualität mit dem Modulationsschema GFSK eine gute Qualität erreicht, dies wird durch das weit offene Augendiagramm [13, Fig. 6] im Aufsatz deutlich.

2.1.2 Ebene Link Layer

Die Ebene Link Layer (LL) ist für die Suche nach Kommunikationspartnern (Scanning), Annoncieren des eigenen Dienstes (Advertising) sowie das Erstellen und Aufrechterhalten von Verbindungen (Connection Handling) verantwortlich. Des Weiteren ist die Ebene Link Layer für die Prüfsummenbildung (CRC) und die Verschlüsselung (Encryption) zuständig. Um dies zu ermöglichen, definiert die Bluetooth SIG die drei folgenden Grundkonzepte: Kanäle, Pakete und Verfahren.

Als Kanäle existieren Advertising-Kanäle und Daten-Kanäle. Advertising-Kanäle werden verwendet, um eine Verbindung aufzubauen und Daten-Kanäle, um Daten auszutauschen.

Die Daten der Ebene Link Layer bestehen aus Paketen, welche die folgenden Bestandteile beinhalten.

- 8 Bit Präambel
- 32 Bit Zugangsadresse
- 8 Bit Header
- 8 Bit Längenfeld
- 0-27 Byte Nutzdaten
- 32 Bit MIC-Wert (Message Integrity Check)
- 24 Bit CRC-Wert (Cyclic Redundancy Check)

Des Weiteren existieren über 20 Verfahren. Dazu gehören zum Beispiel:

- Connection Update Procedure (Verfahren für das Aktualisieren der Verbindung)
- Encryption Procedure (Verfahren für das Entschlüsseln der Verbindung)
- Data Length Update Procedure (Verfahren für das Aktualisieren der Daten Länge)

Die komplette Auflistung der Verfahren ist in der Spezifikation enthalten [5, Vol 6 | Part B | 5.1].

2.1.3 Ebene Logical Link Control and Adaptation Layer Protocol

Die Ebene Logical Link Control and Adaptation Layer Protocol (L2CAP) ist die Multiplexschicht für den Standard Bluetooth Low Energy [5, Vol 3 | Part A | 1]. Diese Ebene definiert zwei grundlegende Konzepte: den L2CAP-Kanal und die L2CAP-Befehle. Der Standard Bluetooth Low Energy verwendet drei feste Kanäle, diese werden in der folgenden Aufzählung genannt.

- Ein Kanal als Signalisierungskanal
- Ein Kanal für die Ebene Security Manager (SM)
- Ein Kanal für die Ebene Attribute Protocol (ATT)

2.1.4 Ebene Security Manager

Die Ebene Security Manager (SM) definiert ein Protokoll für den Austausch (Pairing) und die Verteilung von Sicherheitsschlüsseln. Pairing ist der Vorgang, einem anderen Gerät durch den Austausch von Schlüsseln zu vertrauen. Nach dem Pairing-Vorgang können die Schlüssel verteilt und für die Verschlüsselung der weiteren Kommunikation genutzt werden.

2.1.5 Ebene Generic Access Profile

Die Ebene Generic Access Profile (GAP) definiert die Grundvoraussetzungen eines Gerätes, das den Standard Bluetooth Low Energy anwendet. Es definiert, dass ein Gerät die Ebenen Physical Layer (PHY), Link Layer (LL), L2CAP, Security Manager (SM), Attribute Protocol (ATT) und Generic Attribute Profile (GATT) bereitstellen muss. Außerdem werden die Verhaltensweisen und Methoden für die Geräteerkennung (Device discovery), den Verbindungsaufbau (Connection), die Sicherheit (Security), die Authentifizierung (Authentication), die Zuordnungsmodelle (Association model) und die Diensterkennung (Service Discovery) beschrieben [5, Vol 1 | Part A | 6.2].

2.1.6 Ebene Attribute Protocol

Die Ebene Attribute Protocol (ATT) definiert eine Reihe an Regeln für den Zugriff auf Daten. Die Daten werden als Attribute bezeichnet. Alle Attribute sind in einer Tabelle definiert und bestehen aus einer Nummer (Handle), einem Typ (Type), einer Beschreibung der Eigenschaften (Permission) und Daten der Attribute (Values). Die Tabelle, in der die Attribute definiert sind, wird als Attribut-Server bezeichnet. Ein Gerät, dass auf den Attribut-Server zugreift, wird als Klient (Client) bezeichnet. Ein Gerät, welches wiederum Attribute des Attribut-Server verfügbar macht, wird als Server bezeichnet.

Die Attribute werden eindeutig durch die Kennung Universally Unique Identifier (UUID) identifiziert. Die Bluetooth SIG hat feste, 16 Bit lange UUIDs für viele Anwendungsfälle bereits spezifiziert. Diese festen Kennungen sind in dem Dokument „Assigned Number Document“ [6] definiert.

Durch die Festlegung der Kennung UUID kann ein Client auf das Format und die Eigenschaften der Attribute schließen. Unterschieden wird zwischen den Eigenschaften lesbar

(Read), schreibbar (Write), Benachrichtigung ohne Bestätigung (Notify) und Benachrichtigung mit Bestätigung (Indicate). Mit den Eigenschaften können verschiedene Anwendungsfälle realisiert werden.

Beispiel 2.1 *Das „Heart Rate Profile“ [4] wird von der Bluetooth SIG als Anwendungsfall spezifiziert. Dieser Anwendungsfall besteht aus den Informationen Herzfrequenz-Messung mit der UUID 0x2A37, Sensor Standort mit der UUID 0x2A38 und Kontrollpunkt mit der UUID 0x2A39.*
Die Information Herzfrequenz-Messung besteht aus einem Byte Daten des Attributs und hat die Eigenschaft Notify. Die Information Sensor Standort besteht aus einem Byte Daten des Attributs und hat die Eigenschaft Read. Die Information Kontrollpunkt besteht aus einem Byte Daten des Attributs und hat die Eigenschaft Write.

Des Weiteren ist es möglich, dass der Entwickler selbst UUIDs spezifiziert. Diese nennen sich herstellerspezifische UUID (vendor specific UUID) und bestehen aus 128 Bit.

Beispiel 2.2 *Eine Kennung UUID, die von einem Entwickler genutzt werden könnte: 0xF3BAB216-49B9-4D6B-9104-6993260CFAE5*

Zur Übertragung von Attributen stehen von der Ebene Link Layer pro LL-Paket 0-27 Byte Nutzdaten zur Verfügung. Die 0-27 Byte Nutzdaten des Link-Layer-Pakets setzen sich aus 4 Byte Verwaltungsdaten, der Ebene L2CAP und aus 0-23 Byte für das Attribute Protokoll zusammen. Die Länge eines ATT-Pakets werden als Maximum Transmission Unit (MTU) bezeichnet. Als Standardgröße der MTU wird 23 Byte genommen, es sind aber zwischen 0 und 512 Byte möglich. Die MTU wird von dem Client und dem Server nach dem Verbindungsaufbau ausgehandelt. Der Vorteil für die Einhaltung der Standardgröße ist, dass die ATT-Daten mit nur einem Link-Layer-Paket versendet werden können.
Ein ATT-Paket sieht wie folgt aus:

- 1 Byte ATT Opcode
- 2 Byte Opcode Overhead (Variable Length)
- 20 Byte bis 509 Byte ATT Payload (ATT-Daten)

2.1.7 Ebene Generic Attribute Profile

Die Ebene Generic Attribute Profile (GATT) setzt auf die Ebene Attribute Protocol (ATT) auf und legt die gemeinsamen Operationen und einen Rahmen für die Daten fest, die auf der Ebene Attribute Protocol (ATT) transportiert und gespeichert werden [5, Vol 1 | Part A | 6.4.2].

Für den Rahmen der Daten auf dem Server werden die Attribute zu sogenannten „Services“[1] gruppiert. In den Services sind sogenannte „Characteristics“[2] enthalten [5, Vol 3 | Part G | 2.6.1]. Services und Characteristcs sind als Attribute auf dem Attribut-Server gespeichert. In der folgenden Abbildung wird der Aufbau eines Services mit einer Characteristic skizziert und nachfolgend kurz erläutert.

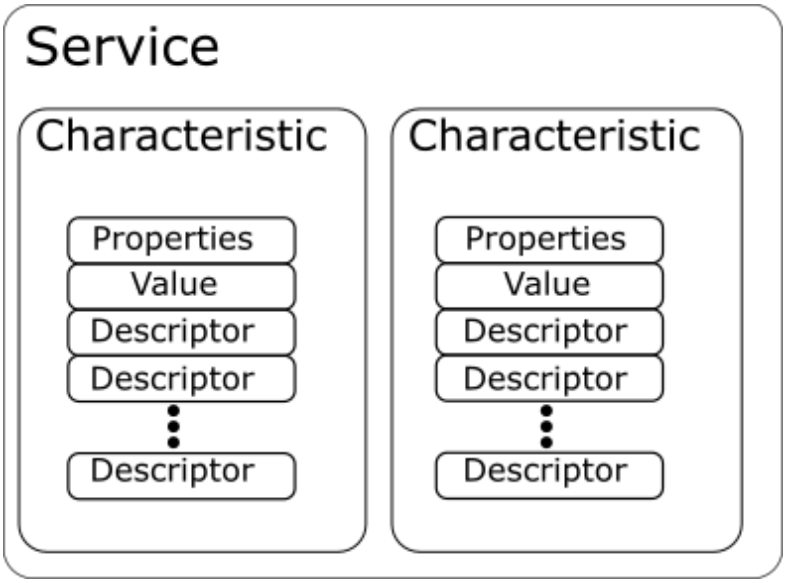

Abbildung 2.2: Darstellung der Services und Characteristics der Ebene, modifiziert nach Bluetooth SIG, 2019, [5, VOL 3| Part G | 2.6.1]

Ein Service kann eine oder mehrere Characteristics enthalten. Eine Characteristic enthält mehrere Beschreibungen über die Characteristic (Descriptor). Die Beschreibungen der Characteristic enthalten wiederum zusätzliche Eigenschaften der Characteristics. Optional können auch die Eigenschaften des Attributs in einer Characteristic enthalten sein

[1] Die Nutzung des englischen Begriffs wurde entgegen der deutschen Grammatik bewusst gewählt.
[2] Siehe vorherige Fußnote.

[5, Vol 3 | Part G | 4.12]. Ein Descriptor ist ebenfalls ein Attribut und wird auf dem Attribut-Server gespeichert.

Jedes Attribut wird auf dem Attribut-Server gespeichert und erhält zur Identifizierung eine Nummer. Für eine Characteristic nennt sich diese Nummer Characteristic Value Handles. Diese wird von dem Client genutzt, um auf eine Characteristic zugreifen zu können. Die Nummer Characteristic Value Handles wird entsprechend der Reihenfolge der Attribute im Attribut-Speicher vergeben.

Weitere Informationen zu den Services und Characteristics sind in der Bluetooth Core Spezifikation an den Stellen [5, Vol 3 | Part G | 2.6.2] und [5, Vol 3 | Part G | 2.6.4] zu finden.

Standardisierte Verfahren der Ebene GATT

Die Ebene GATT wurde entworfen, damit ein Client mit einem Server kommunizieren kann. Dafür hat die Bluetooth SIG Verfahren für die Ebene GATT standardisiert. Die standardisierten Verfahren werden, gemäß der Spezifikation, als Prozeduren bezeichnet.

Die Umsetzung einer Prozedur wird durch Unterprozeduren realisiert, welche jeweils beschreiben, wie die Ebene Attribute Protocol (ATT) verwendet wird, um die entsprechende Prozedur zu ermöglichen.

Die folgenden Prozeduren der Ebene GATT sind im Standard Bluetooth Low Energy definiert (Entnommen aus: [5, Vol 3 | Part G | 4.1]).

1. Server Configuration (Server-Konfiguration)
2. Primary Service Discovery (Auffinden von Primär-Services)
3. Relationship Discovery (Auffinden von Beziehungen)
4. Characteristic Discovery (Auffinden von Characteristics)
5. Characteristic Descriptor Discovery (Auffinden des Descriptor einer Characteristic)
6. Reading a Characteristic Value (Lesen von Daten einer Characteristic)
7. Writing a Characteristic Value (Schreiben von Daten einer Characteristic)

8. Notification of a Characteristic Value (Benachrichtigen über Daten einer Characteristic)
9. Indication of a Characteristic Value (Anzeigen von Daten einer Characteristic)
10. Reading a Characteristic Descriptor (Lesen des Descriptor einer Characteristic)
11. Writing a Characteristic Descriptor (Beschreiben des Descriptor einer Characteristic)

Für diese Arbeit sind die Prozeduren Reading a Characteristic Value, Writing a Characteristic Value, Notification of a Characteristic Value und Indication of a Characteristic Value von Interesse, da diese für den Austausch von Daten zuständig sind.

Für die Kommunikation eines Clients zum Server, werden die Prozeduren *6. Reading a Characteristic Value* und *7. Writing a Characteristic Value* verwendet. Während die Prozedur *6. Reading a Characteristic Value* das Lesen ermöglicht, erlaubt die Prozedur *7. Writing a Characteristic Value* das Schreiben der Daten des Servers.

Für die Kommunikation vom Server zum Client werden die Prozeduren *8. Notification of a Characteristic Value* und *9. Indication of a Characteristic Value* verwendet. Damit die beiden letztgenannten Prozeduren von dem Server genutzt werden können, muss der Client diese erlauben. Dafür beschreibt der Client den Deskriptor Client Characteristic Configuration Descriptor (CCCD).

Die letztgenannten Prozeduren werden beide verwendet, um einen Client über den Wert einer Characteristic des Servers zu informieren. Dabei wird beim Senden einer Notifikation keine Bestätigung des Clients gefordert, wohingegen das Senden einer Indikation von dem Client bestätigt werden muss. Ohne Bestätigung kann keine weitere Indikation gesendet werden.

2.2 Prozeduren

Unterprozeduren werden im Zusammenhang mit dem Standard Bluetooth Low Energy genutzt, um die Prozeduren der Ebene GATT umzusetzen. Für diese Arbeit werden die Unterprozeduren analysiert und in die Kategorien zuverlässige und unzuverlässige Prozeduren eingeordnet.

2.2.1 Zuverlässige Prozeduren

Eine Prozedur wird als zuverlässig bezeichnet, wenn auf eine Anfrage in irgendeiner Form reagiert wird. Dies kann eine Bestätigung, eine Antwort oder eine Fehlermeldung sein. Die Reaktion des Empfängers muss der Anfrage des Senders zuzuordnen sein.

Durch eine Reaktion der Anfrage kann ein Empfänger den Erhalt einer Anfrage quittieren. Dies kann mit oder ohne eine Prüfung der Anfrage erfolgen. Je nachdem, ob eine Prüfung erfolgreich oder fehlerhaft war, kann eine Bestätigung oder Fehlermeldung gesendet werden. Im Standard Bluetooth Low Energy werden Anfragen, die eine Reaktion benötigen, immer vor dem Senden der Antwort geprüft. Die Prüfung wird meistens von den Ebenen in dem Host-System durchgeführt. Durch ein Anfrage-Reaktionspaar kann ein Sender sicherstellen, dass die Anfrage bearbeitet wird.

2.2.2 Unzuverlässige Prozeduren

Eine Prozedur wird als unzuverlässig bezeichnet, wenn auf eine Anfrage nicht reagiert wird. Ohne Reaktion des Empfängers hat der Sender keine Gewissheit darüber, ob der Empfänger die Anfragen erhalten hat und ggf. bearbeitet. Bei Anfragen ohne Reaktionen lässt es sich nicht ausschließen, dass kein Empfänger die Anfrage erhält oder bearbeitet.

Im Standard Bluetooth Low Energy können Anfragen, die keine Reaktion benötigen, jederzeit von einem Sender gesendet werden, unabhängig davon, in welchem Zustand sich der Empfänger befindet. Dies kann dazu führen, dass der Speicher des Empfängers überläuft. Um dies zu vermeiden, kann ein Empfänger Anfragen, die keine Reaktion benötigen, verwerfen. Häufig wird diese Entscheidung auf den Ebenen des Host-Systems entschieden, sodass ein Anwendungs-Programm davon unbelastet bleibt.

3 Analyse der Ebene GATT

Das Ziel dieses Kapitels ist, die Ebene GATT hinsichtlich der Kommunikation zwischen einem Client und einem Server zu analysieren. Dafür wird zuerst die Kommunikation vom Client zum Server und darauffolgend die Kommunikation vom Server zum Client betrachtet. Diesbezüglich werden die Eigenschaften analysiert. Auf dieser Basis werden Einschränkungen für die Nutzung von Systemen, die Latenzen für den Zugriff auf Daten erzeugen, erläutert. Zum Schluss werden Einschränkungen der Ebene GATT analysiert, die beim Zugriff auf eine Vielzahl von Informationen auftreten können.

3.1 Kommunikation vom Client zum Server

Für eine Kommunikation vom Client zum Server werden beim Standard Bluetooth Low Energy die Prozeduren der Ebene GATT genutzt. Das Ziel der Kommunikation ist es, die Daten (Values) der angebotenen Characteristics des Servers zu lesen oder zu beschreiben.

3.1.1 Prozedur Reading a Characteristic Value

Für das Lesen von Characteristics gibt es die Prozedur *Reading a Characteristic Value*. Hierfür sind die folgenden Unterprozeduren definiert (Entnommen aus: [5, Vol 3 | Part G | 4.8]).

- Read Multiple Variable Length Characteristic Values (Lesen von mehreren Characteristics mit verschiedenen Längen der Daten)
- Read Multiple Characteristic Values (Lesen von mehreren Characteristics)
- Read Long Characteristic Values (Lesen von Characteristics, die länger als die Standardgröße MTU sind)

- Read Using Characteristic UUID (Lesen der Characteristics durch die Angabe von der Nummer UUID)
- Read Characteristic Value (Lesen der Characteristic mit der Angabe von der Nummer Characteristic Value Handles)

Die aufgezählten Unterprozeduren erfüllen die Aufgabe, eine oder mehrere Characteristics zu lesen. Der Unterschied zwischen den Unterprozeduren besteht in den Anwendungsbereichen, für die sie spezifiziert wurden. So werden mit den ersten zwei Unterprozeduren mehrere Characteristics gleichzeitig ausgelesen. Die darauf folgende Unterprozedur dient dem Auslesen von Characteristics, die Daten enthalten, die größer als die Standardgröße MTU sind. Die letzten zwei Unterprozeduren sind zum Lesen einer Characteristic, die nicht größer als die Standardgröße MTU ist. Abschließend besteht die Unterscheidung der letzten zwei Unterprozeduren in den Übergabeparametern. *Read Using Characteristic UUID* erwartet als Übergabeparameter die UUID und *Read Characteristic Value* den Characteristic Value Handles.

Für diese Arbeit ist die Betrachtung eines der vier Unterprozeduren der Prozedur *Reading a Characteristic Value* ausreichend, da die Eigenheiten der Unterprozeduren nur für einzelne Anwendungsfälle relevant sind. Aus diesem Grund wird in dieser Arbeit stellvertretend für die Prozedur *Reading a Characteristic Value* die Unterprozedur *Read Characteristic Value* analysiert.

Unterprozedur Read Characteristic Value

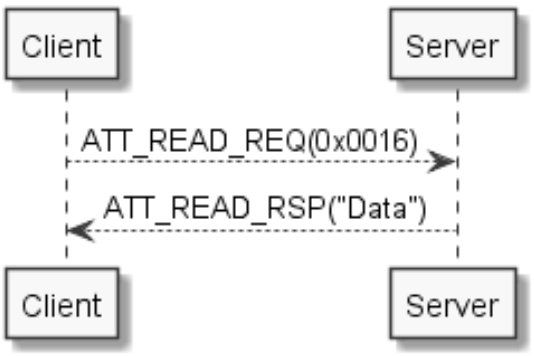

Abbildung 3.1: Eine beispielhafte Anfrage mit der Unterprozedur Read Characteristic Value, modifiziert nach Bluetooth SIG, 2019, [5, Vol 3| Part G| 4.8.1]

In der Abbildung 3.1 werden die ATT-Prozeduren der Unterprozedur *Read Characteristic Value* dargestellt. Eine Transaktion beginnt mit dem Client, dieser stellt eine Anfrage durch das Senden der ATT-Prozedur ATT Read Request an den Server. Daraufhin antwortet der Server mit der ATT-Prozedur ATT Read Response, diese ATT-Prozedur enthält die Daten (Values) der angefragten Characteristic. Alternativ kann der Server mit der ATT-Prozedur ATT ERROR RSP auf eine fehlerhafte Anfrage antworten. Die Transaktion der Unterprozedur wird durch die Antwort des Servers beendet. Besonders hervorzuheben ist, dass aus der Antwort des Servers zwei Informationen abgeleitet werden können: zum einen eine Bestätigung, dass die Anfrage erhalten wurde und zum anderen die angefragten Daten des Servers.

Ein weiterer Aspekt, der in Betracht gezogen werden muss, ist, dass die Bluetooth SIG für die Ebene *Attribute Protocol* (ATT) die Nutzung des sequenziellen Anfrage-Antwort Protokolls (Sequential Request-Response Protocol) definiert hat. Im Sinne des Protokolls soll während einer Transaktion des Client keine weitere Transaktion an denselben Server gesendet werden, bis die Transaktion beendet wurde [5, Vol 3 | Part F | 3.3.2]. Daraus resultiert, dass eine Anfrage des Client zum Blockieren weiterer Kommunikation führt. Erst mit der Antwort des Servers kann weiter kommuniziert werden. Dieses Verhalten entspricht einer synchronen Kommunikation. Aus diesem Grund kann das Lesen von Attributen als synchrone Kommunikation kategorisiert werden.

Zusätzlich ist die Unterprozedur *Read Characteristic Value* eine zuverlässige Prozedur, da der Server mit der ATT-Prozedur ATT ERROR RSP auf fehlerhafte Anfragen hinweist und valide Anfragen durch das Senden der Antwort bestätigen kann.

3.1.2 Zusammenfassung der Prozedur Reading a Characteristic Value

Tabelle 3.1: Zusammenfassung: Kommunikation mit der Prozedur *Reading a Characteristic Value*

Unterprozeduren	**Synchron**	**Asynchron**	**Verlässlich**
Read Multiple Variable Length Characteristic Values	X		X
Read Multiple Characteristic Values	X		X
Read Long Characteristic Values	X		X
Read Using Characteristic UUID	X		X
Read Characteristic Value	X		X

Die Tabelle 3.1 fasst die Analyse zusammen. Es ist zu erkennen, dass die Prozedur *Reading a Characteristic Value* ausschließlich aus synchronen und verlässlichen Unterprozeduren besteht.

3.1.3 Prozedur Writing a Characteristic Value

Eine weitere Kommunikation vom Client zum Server besteht im Beschreiben der Daten einer Characteristic. Dafür gibt es die Prozedur *Writing a Characteristic Value*. Für diese Prozedur hat die Bluetooth SIG die folgenden Unterprozeduren definiert (Entnommen aus: [5, Vol 3 | Part G | 4.9]).

- Signed Write Without Response (Gesichertes Schreiben ohne Antwort)
- Write Long Characteristic Values (Schreiben von langen Characteristics)
- Reliable Writes (Zuverlässiges Schreiben)
- Write Without Response (Schreiben ohne Antwort)
- Write Characteristic Value (Schreiben einer Characteristic)

Die Betrachtung der fünf Unterprozeduren der Prozedur *Writing a Characteristic Value* unterscheiden sich nicht nur in den Anwendungsbereichen, sondern führen auch zu unterschiedlichen Kommunikationsarten. Deshalb werden im Folgenden die fünf Unterprozeduren der Prozedur *Writing a Characteristic Value* analysiert.

Unterprozedur Signed Write Without Response

Die Unterprozedur *Signed Write Without Response* wird genutzt, wenn eine Bluetooth Low Energy Verbindung nicht verschlüsselt ist, jedoch einzelne Nachrichten verschlüsselt werden sollen. Eine unverschlüsselte Verbindung wird eingesetzt, wenn das Verschlüsseln der gesamten Verbindung zu hohen Latenzen führt.

Die Unterprozedur bietet keine Möglichkeit, auf fehlerhafte Anfragen zu antworten, denn eine Antwort wird für diese Unterprozedur nicht erwartet. Die Unterprozedur *Signed Write Without Response* ist dementsprechend eine unzuverlässige Prozedur. Des Weiteren wird in dieser Unterprozedur asynchron kommuniziert, da eine Anfrage nicht zum Blockieren weiterer Kommunikation führt.

Für diese Arbeit ist die Unterprozedur *Signed Write Without Response* nicht relevant, da der Anwendungsfall zu speziell ist. Aus diesem Grund wird diese Unterprozedur nicht weiter betrachtet.

Unterprozedur Write Long Characteristic Values

Die Unterprozedur *Write Long Characteristic Values* wird genutzt, um Daten, die länger als 20 Byte sind, auf eine Characteristic zu schreiben. Hierzu werden zwei ATT-Prozeduren genutzt. Mit der ersten ATT-Prozedur ATT Prepare Write Req kann der Client Daten senden. Für dieses Ziel werden die Daten in 20 Byte Datenpakete aufgeteilt. Diese Datenpakete werden sequenziell versendet. Der Server schreibt die Werte nicht direkt in die Characteristic, sondern wartet auf die zweite ATT-Prozedur. Sobald der Client die zweite ATT-Prozedur ATT Execute Write Req sendet, werden die Daten in die Characteristic geschrieben.

Eine Besonderheit dieser Unterprozedur ist, dass jedes Datenpaket bestätigt wird. Der Inhalt der Bestätigung sind die gesendeten Daten. Der Absender kann somit prüfen, ob die Daten richtig empfangen wurden.

Mit der ATT-Prozedur ATT Prepare Write Req kommt es zum Blockieren weiterer Kommunikation. Zwar kann der Absender dem Empfänger durch das Senden der ATT-Prozedur ATT Prepare Write Req weitere Daten senden, jedoch kann keine anderweitige Kommunikation durch andere Prozeduren erfolgen. Der Sender kann die ATT-Prozedur ATT Prepare Write Req erst senden, wenn der Empfänger mit der ATT-Prozedur ATT

Prepare Write Resp geantwortet hat. Dieses Verhalten zeigt, dass die Unterprozedur *Write Long Characteristic Values* synchron kommuniziert.

Auf eine fehlerhafte Anfrage kann der Empfänger mit der ATT-Prozedur ATT ERROR RSP reagieren. Die Bestätigung der korrekten Anfrage wird durch die Antwort gesendet. Dadurch ist die Unterprozedur *Write Long Characteristic Values* eine zuverlässige Prozedur.

Unterprozedur Reliable Writes

Die Unterprozedur *Reliable Writes* nutzt dieselben ATT-Prozeduren wie *Write Long Characteristic Values*. Der Unterschied liegt in der Anwendung der ATT-Prozeduren. Die Unterprozedur *Reliable Writes* wird genutzt, um sicherzustellen, dass die versendeten Daten korrekt sind. Zu diesem Zweck vergleicht der Client die beantworteten Daten mit den gesendeten Daten und kann bei einem Fehler die ganze Übertragung abbrechen.

Des Weiteren wird diese Unterprozedur genutzt, um unterschiedliche Characteristics zeitgleich zu beschreiben. Für dieses Ziel sendet der Client die Daten sequenziell mit der ATT-Prozedur *ATT Prepare Write Req* an unterschiedliche Characteristics. Im Anschluss werden die Daten mit der ATT-Prozedur *ATT Execute Write Req* in die unterschiedlichen Characteristics geschrieben.

Wie die vorherige Unterprozedur ist auch diese Unterprozedur eine zuverlässige Prozedur, die synchron kommuniziert. Da jedoch die beiden zuletzt genannten Unterprozeduren nur für bestimmte Anwendungsfälle genutzt werden, werden diese Unterprozeduren in dieser Arbeit nicht weiter betrachtet.

Unterprozedur Write Characteristic Value

Abschließend werden noch die beiden letzten Unterprozeduren analysiert. Die Unterprozedur *Write Characteristic Value* besteht aus den ATT-Prozeduren ATT Write Request und ATT Write Response.

Wie in der Abbildung 3.2 dargestellt wird führt eine Anfrage mit der ATT-Prozedur ATT Write Request zum Start einer Transaktion. Erst die Antwort des Servers mit der ATT-Prozedur ATT Write Response beendet diese Transaktion. Alternativ kann der Server auf fehlerhafte Anfragen mit der ATT-Prozedur ATT ERROR RSP antworten. Eine

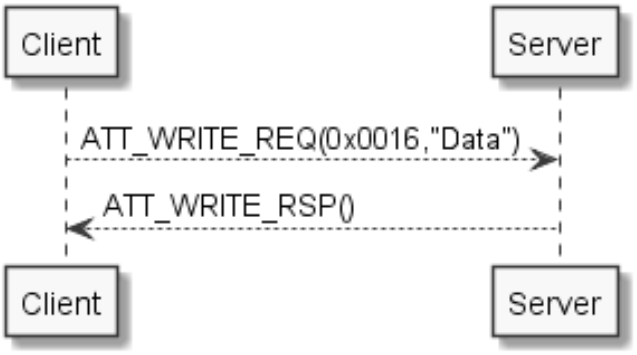

Abbildung 3.2: Eine beispielhafte Anfrage mit der Unterprozedur Write Characteristic Value, modifiziert nach Bluetooth SIG, 2019, [5, Vol 3| Part G| 4.9.3]

Kommunikation mit der Unterprozedur *Write Characteristic Value* ist dementsprechend synchron. Zusätzlich können fehlerhafte Anfragen vom Empfänger mitgeteilt oder valide Anfragen durch das Senden der Antwort bestätigt werden. Demnach ist die Unterprozedur eine zuverlässige Prozedur.

Hervorzuheben ist, dass aus der Antwort des Servers nur die Information hervorgeht, ob die Anfrage vom Client als valide eingestuft wurde. Es werden durch die ATT-Prozedur keine Daten des Servers versendet. Letztlich bleibt noch die Analyse der Unterprozedur *Write Without Response*.

Unterprozedur Write Without Response

Die Abbildung 3.3 zeigt, dass die Unterprozedur *Write Without Response* ausschließlich aus der ATT-Prozedur ATT WRITE CMD besteht. Ein Transaktionsbeginn oder -ende

Abbildung 3.3: Eine beispielhafte Anfrage mit der Unterprozedur Write Without Response, modifiziert nach Bluetooth SIG, 2019, [5, Vol 3| Part G| 4.9.1]

gibt es nicht. Mit dieser Unterprozedur entsteht kein Blockieren des Kommunikationsprozesses und es muss keine Antwort gesendet werden. Entsprechend ist die Kommunikation asynchron. Zusätzlich ist die Unterprozedur eine unzuverlässige Prozedur, denn ohne Antwort weiß der Client nicht, ob die Nachricht korrekt erhalten wurde.

3.1.4 Zusammenfassung der Prozedur Writing a Characteristic Value

Tabelle 3.2: Zusammenfassung: Kommunikation mit der Prozedur *Writing a Characteristic Value*

Unterprozeduren	**Synchron**	**Asynchron**	**Verlässlich**
Signed Write Without Response		X	
Write Long Characteristic Values	X		X
Reliable Writes	X		X
Write Without Response		X	
Write Characteristic Value	X		X

Die Tabelle 3.2 fasst die Analyse zusammen. Es ist zu erkennen, dass die Prozedur *Writing a Characteristic Value* zwei Unterprozeduren enthält, die asynchron und nicht verlässlich sind. Stellvertretend für diese Prozedur wird die Unterprozedur *Write Without Response* gewählt. Weiter sind drei Unterprozeduren enthalten, die synchron und verlässlich sind. Stellvertretend für diese Unterprozeduren wird die Unterprozedur *Write Characteristic Value* ausgewählt. Durch die speziellen Anwendungsbereiche der anderen Characteristics können diese nicht für das allgemeine Kommunikationsprotokoll genutzt werden. Im weiteren Verlauf werden die Stellvertreter der Unterprozeduren weiter analysiert.

3.2 Kommunikation vom Server zum Client

In diesem Abschnitt wird die Kommunikation vom Server zum Client analysiert. Die Ebene GATT enthält für diese Kommunikationsart zwei Prozeduren. Zum einen die Prozedur *Notification of a Characteristic Value* und zum anderen die Prozedur *Indication of a Characteristic Value*.

3.2.1 Prozedur Notification of a Characteristic Value

Die Prozedur *Notification of a Characteristic Value* besteht aus der Unterprozedur *Notifications*. Diese Unterprozedur wird genutzt, wenn ein Server einen Client benachrichtigen möchte, ohne dass der Client eine Bestätigung sendet. Wie die folgende Abbildung zeigt, besteht die Unterprozedur lediglich aus der ATT-Prozedur ATT_HANDLE_VALUE_NTF (siehe Abbildung 3.4).

Abbildung 3.4: Eine beispielhafte Anfrage mit der Unterprozedur Notifications, modifiziert nach Bluetooth SIG, 2019, [5, Vol 3| Part G| 4.10.1]

Da die Unterprozedur *Notifications* keine Antwort oder Bestätigung beinhaltet, kann der Empfänger keine Fehler mitteilen oder Anfragen bestätigen. Demzufolge ist die Unterprozedur eine unzuverlässige Prozedur. Des Weiteren führt die Unterprozedur *Notifications* nicht zum Blockieren der Kommunikation, denn die Transaktion besteht lediglich aus einer ATT-Prozedur. Nach dem Senden kann die Kommunikation fortgeführt werden und somit ist die Kommunikation mit der Unterprozedur asynchron.

3.2.2 Prozedur Indication of a Characteristic Value

Die Prozedur *Indication of a Characteristic Value* besteht aus der Unterprozedur *Indications*. Eine Transaktion beginnt mit der ATT-Prozedur ATT HANDLE VALUE IND und endet mit der Bestätigung des Empfängers durch die ATT-Prozedur ATT HANDLE VALUE CFM (siehe Abbildung 3.5).

Da während einer Transaktion keine weiteren Transaktionen parallel laufen können, führt die Unterprozedur zum Blockieren weiterer Kommunikation. Demzufolge ist die Kommunikation mit der Unterprozedur synchron. Zusätzlich ist die Unterprozedur durch die

Bestätigung mit der ATT-Prozedur ATT_HANDLE_VALUE_CFM eine zuverlässige Prozedur.

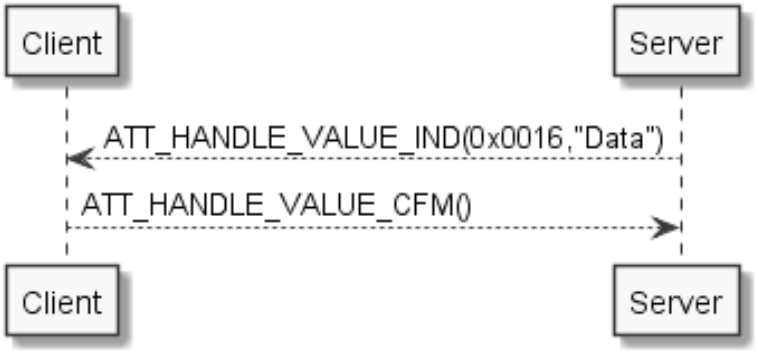

Abbildung 3.5: Eine beispielhafte Anfrage mit der Unterprozedur Indications, modifiziert nach Bluetooth SIG, 2019, [5, Vol 3| Part G| 4.11.1]

3.2.3 Zusammenfassung der Ergebnisse der Kommunikation vom Server zum Client

Tabelle 3.3: Zusammenfassung: Kommunikation vom Server zum Client

Unterprozeduren	**Synchron**	**Asynchron**	**Verlässlich**
Notifications		X	
Indications	X		X

Die Tabelle 3.3 fasst die Analyse zusammen. Es ist zu erkennen, dass während die Prozedur *Indicate* synchron und verlässlich ist, die Prozedur *Notification* asynchron und nicht verlässlich ist.

3.3 Antwortverhalten der zuverlässigen Prozeduren

Bei den zuverlässigen Prozeduren wird eine Transaktion durch eine Antwort des Servers oder eine Bestätigung des Client beendet. Die Antwort sowie die Bestätigung implizieren mehrere Informationen. Zum einen wird bestätigt, dass die Anfrage empfangen wurde und zum anderen, dass die Anfrage semantisch und syntaktisch korrekt ist.

Die Antwort des Servers oder die Bestätigung des Client wird meistens auf der Ebene ATT gesendet. Bei einer fehlerhaften Anfrage erfährt ein Anwenderprogramm nicht, dass eine fehlerhafte Anfrage gestellt wurde. Die Ebene ATT sendet dementsprechend die ATT-Prozedur ATT_ERROR_RSP() mit einem Fehlercode (Error Code), der auf den Fehler hinweist [5, Vol 3 | Part F | 3.4.1.1]. Nachdem die Ebene ATT die Anfrage als valide eingestuft hat, kann ein Anwenderprogramm eine eigene Prüfung vornehmen.

Bei der eigenen Prüfung durch das Anwenderprogramm kann die Anfrage des Client nach selbst definierten Anforderungen geprüft werden. Demzufolge kann vom Anwenderprogramm die ATT-Prozedur ATT_ERROR_RSP() mit einem Fehlercode (Error Code) versehen werden. Exemplarisch hierfür existiert der Fehlercode Application Error (0x80 - 0x9F), damit werden eigene Fehlerfälle definiert. Wenn das Anwenderprogramm keinen Fehler entdeckt, wird die Antwort bzw. Bestätigung, entsprechend der Vorgabe der Prozedur, auf der Ebene ATT gesendet.

Analyse der Antworten der Prozeduren

Die Antworten der Prozeduren *Reading* und *Writing* bestehen aus unterschiedlichen ATT-Prozeduren. Aus diesem Grund werden im Folgenden die ATT-Prozeduren der stellvertretenden Unterprozeduren der Prozeduren *Reading* und *Writing* miteinander verglichen.

Die Unterprozedur *Write Characteristic Value* wird durch die ATT-Prozedur ATT_WRITE_RSP() beendet. Im Vergleich zur ATT-Prozedur ATT_READ_RSP(Data) besteht diese ATT-Prozedur nicht aus der Kopplung von Daten und Bestätigung der Anfrage, sondern nur aus einer Bestätigung der Anfrage. Der Grund hierfür ist, dass die Prozedur *Writing* keine Daten anfragt, sondern Daten versendet. Daraus lässt sich die Schlussfolgerung ziehen, dass die Unterprozedur *Write Characteristic Value* genutzt werden kann, um eine Entkoppelung von Daten und Bestätigung zu erhalten.

3.4 Einschränkung der Kommunikation

Die synchrone Kommunikation zwischen einem Client und einem Server führt zu Einschränkungen beim Zugriff auf Characterics. In den folgenden Abschnitten werden die

Einschränkungen erarbeitet, die für ein System entstehen, welches während eines Zugriffs auf Daten Latenzen erzeugt und eine hohe Anzahl an Daten zur Verfügung stellt.

3.4.1 Ressourcenverbrauch

Eine Einschränkung entsteht bei der Verwaltung von Attributen. Für Anwendungen von Bluetooth Low Energy kommen häufig Mikrocontroller zum Einsatz, welche begrenzte Ressourcen haben. Demzufolge muss der Ressourcenverbrauch gering gehalten werden.

Limitierung der Anzahl der Characteristics

Eine Einschränkung des Standard Bluetooth Low Energy entsteht durch eine hohe Anzahl an Characteristics.

Zu jeder Characteristic gehören Daten, die sich aus dem Characteristic Value Handles, der UUID der Characteristic, der Beschreibung der Eigenschaften und den Daten (Values) der Characteristic zusammensetzen. Zusätzlich enthalten Characteristics Descriptoren, die ebenfalls Ressourcen verbrauchen, da sie auch Attribute sind. Deswegen besteht jeder Descriptor wieder aus einem Handle, einer UUID, einer Beschreibung der Eigenschaften und aus den Daten des Descriptors. Jedes Attribut der Characteristic führt zu einem Ressourcenverbrauch auf dem Mikrocontroller. Besonders die 128 Bit langen vendor specific UUIDs der Attribute, führen zu einem hohen Ressourcenverbrauch. Dies führt dazu, dass die Anzahl der Characteristics begrenzt ist.

Beispiel 3.1 *Bei einem Mikrocontroller können für die Attribute 100 kByte genutzt werden. Als Beispiel wird für ein Attribut die Größe von 200 Byte verwendet. Dadurch können maximal 500 Attribute gespeichert werden.*
Für dieses Beispiel wird das Heart-Rate Profil genutzt. Das Heart-Rate Profil besteht aus 6 Attributen.
Pro Characteristic ergeben sich somit:
$6 \cdot 200\ Byte = 1.2\ kByte$
Demzufolge ergibt sich für die Anzahl der maximalen Characteristics:
100 kByte/1,2 kByte = 83
Dadurch können in diesem Beispiel etwa 83 Characteristics gespeichert werden.

Ungenutzte Adressierung

Aus der Limitierung der Anzahl der Characteristics ergeben sich ungenutzte Adressräume. Häufig werden für die Characteristics die 128 Bit langen vendor specific UUIDs genutzt, sodass 2^{128} unterschiedliche Adressen möglich sind. Durch eine Anpassung der Adressen an die Maximalanzahl der Informationen, können Ressourcen eingespart werden. Eine neue Adressierung führt somit zu einem geringeren Ressourcenverbrauch.

Beispiel 3.2 *Da mit einem Byte 256 Adressen verteilt werden können, reicht dies um 83 Sensoren mit Adressen zu versehen. Somit können pro Characteristic 120 Bit also 15 Byte eingespart werden.*

3.4.2 Addieren von Latenzzeiten

Die Haupteinschränkung besteht beim Zugriff auf Informationen durch die synchrone Kommunikation. Für den Fall, dass das Lesen von Informationen zu Latenzen führt, entsteht die Einschränkung, dass die Kommunikation für die Dauer der Latenz blockiert ist.

Die Blockierung verlängert sich um die Latenzen der einzelnen Anfragen. Dementsprechend addieren sich die Latenzen auf. Bei vielen Anfragen nacheinander führt das zu langen Zugriffszeiten auf die Daten.

Beispiel 3.3 *Ein Gerät besteht aus 10 Sensoren und ein Sensor braucht 200 ms bis ein Wert erzeugt wurde, dann kommt es bei einer synchronen Abfrage zu einer Zugriffszeit von 2 Sekunden*[1]*.*

Die im Anhang A enthaltene Abbildung A.2 zeigt drei synchrone Anfragen. Die Anfragen erzeugen Latenzen während des Zugriffs auf Daten. Durch die Latenzen der einzelnen Abfragen entsteht eine Zugriffszeit, die der Summe der einzelnen Latenzen entspricht. Durch die synchrone Kommunikation ergibt sich eine gesamte Latenzzeit, die der Summe aus der Latenzzeiten 1, 2 und 3 der Module Modul_1 und Modul_2 entspricht[2].

[1] Die Übertragungszeiten der Anfragen werden nicht berücksichtigt, da diese in jeder Anfrage enthalten sind.
[2] Siehe vorherige Fußnote.

4 Konzept

In diesem Teil der Arbeit wird ein Konzept erstellt, das beschreibt, wie eine asynchrone Kommunikation auf Basis der zuverlässigen Prozeduren entwickelt werden kann. Dafür muss das Kommunikationsprotokoll die standardisierten Verfahren der Ebene GATT nutzen. Die Umsetzung des Kommunikationsprotokolls soll keine Veränderungen des Bluetooth Low Energy Host- oder Controllersystems benötigen. Für die Praxis hat dies den Vorteil, dass die Umsetzung des Kommunikationsprotokolls für Geräte, die den Standard Bluetooth Low Energy unterstützen (z. B. Smartphones), durch Anwenderprogramme erreicht werden kann.

4.1 Asynchrones Verfahren

Die Prozeduren *Reading* und *Writing* bestehen aus jeweils zwei ATT-Prozeduren. Diese führen zur Kopplung von Anfrage und Antwort. Um die Kopplung der Anfrage des Client mit der Antwort des Servers aufzuheben, wird in dieser Arbeit ein Verfahren entwickelt. Mit dem Verfahren als Grundlage entsteht ein Kommunikationsprotokoll für das asynchrone Schreiben und Lesen von Informationen des Servers. Das Verfahren wird im Folgenden als asynchrones Verfahren bezeichnet. In der Abbildung 4.1 wird das asynchrone Verfahren skizziert und nachfolgend kurz erläutert.

Das asynchrone Verfahren wird durch die *Anmeldung der Anfrage* vom Client gestartet. Daraufhin wird die Anfrage vom Server bestätigt. Die Zeit, die vergeht bis die Anfrage beantwortet wird, kann durch unterschiedliche Faktoren variieren. Wenn dem Server die angefragten Daten vorliegen, sendet dieser die *Antwort auf die Anfrage*. Nach dem Erhalt der Antwort muss der Client diese bestätigen. Mit der *Bestätigung der Antwort* wird das asynchrone Verfahren beendet.

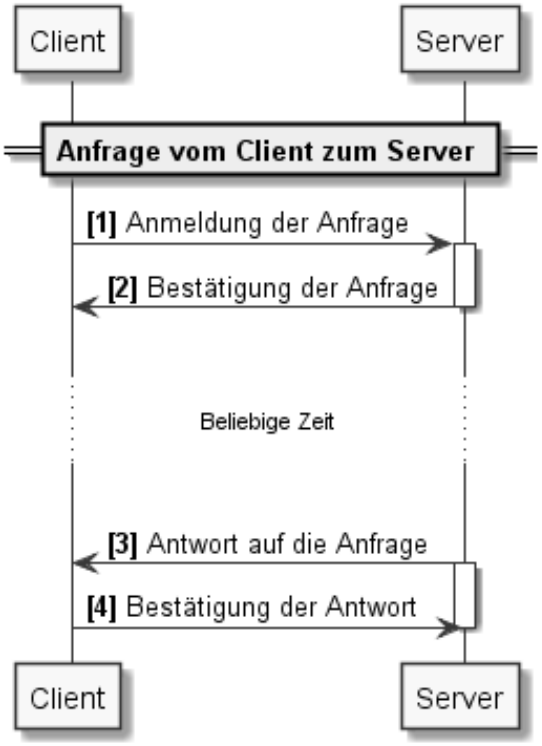

Abbildung 4.1: Darstellung der allgemeinen Schritte des asynchronen Verfahren für Bluetooth Low Energy

4.2 Kommunikationsprotokoll

Für das Kommunikationsprotokoll werden die vier Schritte des asynchronen Verfahrens mit den Prozeduren der Ebene GATT umgesetzt.

4.2.1 Anmeldung der Anfrage

Damit eine Anfrage zu einem asynchronen Datenaustausch führen kann, müssen dem Server Informationen gesendet werden. Die Informationen bestehen aus der Art der Operation, der Adressierung der Anfrage und der Anfrage. Dieser Vorgang wird als *Anmeldung der Anfrage* bezeichnet.

Damit der Client seine Anfrage anmelden kann, muss eine Anfrage mithilfe der Prozedur *Writing* gestellt werden. Mit dieser Prozedur kann ein Client Daten an einen Server

senden. Für die *Anmeldung der Anfrage* werden die benötigten Informationen der Anfrage in Bereiche unterteilt und wie folgt als *Writing*-Anfrage aufgebaut:

ATT_WRITE_REQ(0x01, O KEY [VALUE])

Die Bereiche werden definiert als:

- **O** für die genutzte Operation.
- **KEY** für die Adressierung der Antwort.
- **VALUE** für die Übertragung von einem Wert.

Die Umsetzung der *Anmeldung der Anfrage* mit der Prozedur *Writing* hat den Vorteil, dass die Anfrage nicht durch das Senden von Daten, sondern durch das Senden einer Bestätigung beendet wird (siehe Unterunterabschnitt 3.3). Für das Kommunikationsprotokoll wird dieser Vorteil genutzt, um eine Entkoppelung von Daten und Bestätigung zu erreichen. Da das lange Blockieren der Kommunikation durch die Latenzzeiten der Daten entsteht (siehe Unterabschnitt 3.4.2), kann durch die Entkopplung die Transaktion unabhängig von den Latenzen der Daten beendet werden.

4.2.2 Bestätigung der Anfrage

Durch die Entkopplung der Daten und der Bestätigung wird auf eine *Anmeldung der Anfrage* nur eine Bestätigung der Anfrage gesendet. Die *Bestätigung der Anfrage* wird von der Ebene ATT beantwortet. Oder ein Anwenderprogramm kann, wie bereits in Abschnitt 3.3 erwähnt, eine eigene Prüfung einer Anfrage vornehmen. Empfehlenswert ist, die Prüfung der Anfrage sehr kurz zu halten. Denn eine lange Prüfung verzögert das Beenden der Transaktion, wodurch weitere Kommunikation blockiert wird.

4.2.3 Antwort auf die Anfrage

Die *Antwort auf die Anfrage* ist eine Kommunikation vom Server zum Client. Dadurch können nur die zwei Prozeduren *Notification* oder *Indicate* genutzt werden. Wie in der Analyse erarbeitet, ist die Prozedur *Notification* eine unzuverlässige Prozedur. Demzufolge wird für die *Antwort auf die Anfrage* die Prozedur *Indicate* genommen, welche eine zuverlässige Prozedur ist.

Die *Antwort auf die Anfrage* muss die angemeldete Anfrage beantworten, sobald dem Server die benötigten Daten vorliegen. Die *Antwort auf die Anfrage* wird wie folgt aufgebaut:

ATT_HANDLE_VALUE_IND(0x01, O KEY [VALUE|STATUS])

Die Bereiche werden definiert als:

- **O** für die genutzte Operation.
- **KEY** für die Adressierung der Antwort.
- **VALUE** für die Übertragung von einem Wert
  oder
- **STATUS** für die Rückmeldung von Fehlern oder Erfolgen.

Mit dem Datenfeld der Indikation kann der Server auf die *Anmeldung der Anfrage* antworten. Der Server hat dafür zwei Möglichkeiten. Zum einen kann die Anfrage Fehler bei der Verarbeitung verursacht haben, dann kann der Server mit dem Bereich [STATUS] auf den Fehler hinweisen. Zum anderen kann die Anfrage fehlerfrei sein, dann kann der Client mit dem Bereich [VALUE] die angefragten Daten senden oder mit dem Bereich [STATUS] einen Erfolg mitteilen.

4.2.4 Bestätigung der Antwort

Zuletzt muss der Erhalt der Antwort durch die ATT-Prozedur ATT_HANDLE_VALUE_CFM() durch den Client bestätigt werden.

4.2.5 Zusammenfassung des Ablaufs einer Anfrage

Damit können durch die Nutzung von den Prozeduren *Writing* und *Indicate* die vier Stufen des asynchronen Verfahrens umgesetzt werden. Es ergibt sich der folgende Ablauf für eine Anfrage:

Die Abbildung 4.2 zeigt den Ablauf einer allgemeinen Anfrage mit den Prozeduren der Ebene GATT. Durch die Entkopplung von Antwort und Bestätigung kann zwischen der *Bestätigung der Anfrage* (2) und der *Antwort auf die Anfrage* (3) weitere Kommunikation

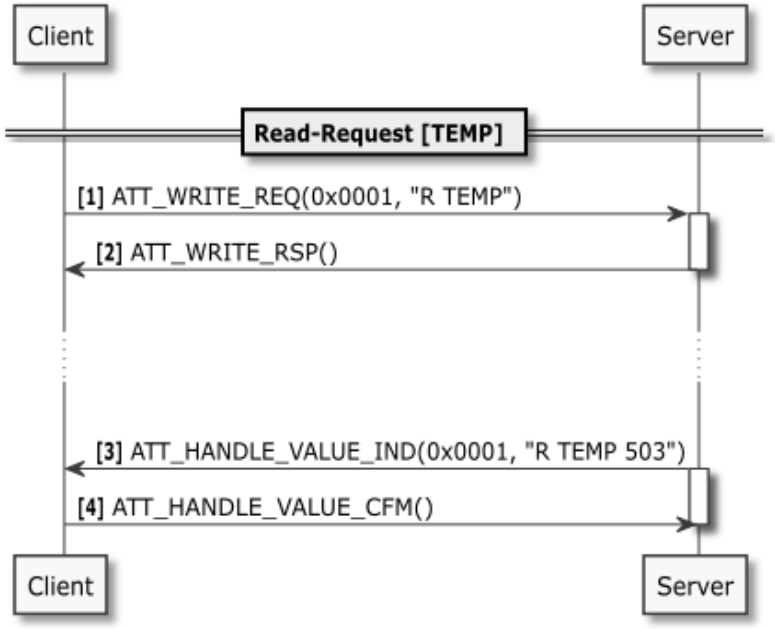

Abbildung 4.2: Das Kommunikationsprotokoll auf Basis des asynchronen Verfahrens

ermöglicht werden. Dies wird dadurch erreicht, dass eine Transaktion von dem Client mit der ATT-Prozedur ATT_WRITE_REQ (1) gestartet wird und durch die *Bestätigung der Anfrage* (2) die Transaktion der Prozdeur *Writing* beendet wird.

Nach der Bearbeitung der Anfrage sendet der Server die Antwort und eröffnet somit wieder eine Transaktion. Mit der ATT-Prozedur ATT_HANDLE_VALUE_IND (3) sendet der Server dem Client die Daten [VALUE] oder die Rückmeldung [STATUS] zu der Anfrage. Der Client beendet die Transaktion durch das Senden der *Bestätigung der Antwort* (4).

In der Abbildung 4.2 werden die Transaktionen durch Blöcke beim Server dargestellt. Während dieser Zeit ist keine weitere Kommunikation möglich.

4.3 Lese-Anfrage

In den folgenden Abschnitten wird erklärt, wie eine Anfrage gestellt werden muss, damit sie als Lese-Anfrage identifiziert werden kann.

4.3.1 Anmelden einer Lese-Anfrage

Eine Lese-Anfrage muss von einer Schreib-Anfrage unterschieden werden. Damit eine Anfrage als Lese-Anfrage identifiziert werden kann, muss der Bereich [O] auf die Operation Lese-Anfrage [R] gesetzt werden. Eine Eigenschaft der Lese-Anfrage ist, dass keine zusätzlichen Daten versendet werden, demnach wird der Bereich [VALUE] leer gelassen. Daraus folgt, dass die Lese-Anfrage wie folgt aufgebaut wird:

ATT_WRITE_REQ(0x01, R KEY)

Beispiel 4.1 *Eine Lese-Anfrage für die Temperatur einer Akkumulator-Zelle: ATT_WRITE_REQ(0x01, R TEMP1)*

4.3.2 Antwort auf die Lese-Anfrage

Dieser Schritt entspricht der *Antwort auf die Anfrage*. Vor dem Senden der Antwort wird die Anfrage geprüft. Im Falle eines Fehlerzustands muss eine Antwort gesendet werden, die auf den Fehler hinweist.

Datenstruktur der Antwort auf die Lese-Anfrage

Wenn kein Fehler besteht, werden die Daten wie folgt gesendet:

ATT_HANDLE_VALUE_IND(0x01, R KEY VALUE)

Beispiel 4.2 *Eine Read-Antwort auf die Anfrage der Temperatur sieht dementsprechend wie folgt aus: ATT_HANDLE_VALUE_IND(0x01, R TEMP 500)*

Wenn ein Fehler besteht, wird die folgende Antwort gesendet:

ATT_HANDLE_VALUE_IND(0x01, R KEY STATUS)

Beispiel 4.3 *Im Falle eines Fehlerzustands könnte eine Antwort wie folgt aussehen: ATT_HANDLE_VALUE_IND(0x01, R TEMP ERROR2)*

Damit die zwei Antwortmöglichkeiten auf die Lese-Anfrage voneinander unterschieden werden können, müssen für die Bereiche [VALUE] und [STATUS] unterschiedliche Wertebereiche definiert werden.

4.4 Schreib-Anfrage

In den folgenden Abschnitten wird erklärt, wie eine Anfrage gestellt werden muss, damit sie als Schreib-Anfrage identifiziert werden kann.

4.4.1 Anmeldung einer Schreib-Anfrage

Damit die Anfrage mit den Prozeduren Writing als Schreib-Anfrage identifiziert werden kann, muss der Bereich [O] auf die Operation Schreib-Anfrage [W] gesetzt werden. Des Weiteren werden für eine Schreib-Anfrage, die Daten in den Bereich [VALUE] gesetzt. Daraus folgt, dass die Schreib-Anfrage wie folgt aufgebaut sein muss:

ATT_WRITE_REQ(0x01, W KEY VALUE)

Beispiel 4.4 *Eine Schreib-Anfrage für die Soll-Temperatur einer Heizung: ATT_ WRITE_ REQ(0x01, R S_ TEMP2 2500)*

4.4.2 Antwort auf eine Schreib-Anfrage

Dieser Schritt entspricht der *Antwort auf eine Anfrage*. Vor dem Senden der Antwort wird die Semantik der Anfrage geprüft. Daraufhin müssen die Daten verarbeitet werden. Im Falle eines Fehlerzustands muss eine Antwort gesendet werden, die auf den Fehler hinweist.

Datenstruktur der Antwort auf die Schreib-Anfrage

Wenn kein Fehler besteht, werden die Daten wie folgt gesendet:

ATT_HANDLE_VALUE_IND(0x01, W KEY STATUS)

Beispiel 4.5 *Eine Antwort auf die Schreib-Anfrage der Temperatur sieht dementsprechend wie folgt aus: ATT_ HANDLE_ VALUE_ IND(0x01, W S_ TEMP2 OK)*

Wenn ein Fehler besteht, wird die folgende Antwort gesendet:

ATT_HANDLE_VALUE_IND(0x01, W KEY STATUS)

Beispiel 4.6 *Im Falle eines Fehlerzustands könnte eine Antwort wie folgt aussehen: ATT_ HANDLE_ VALUE_ IND(0x01, W S_ TEMP2 ERROR5)*

4.5 Effiziente Nutzung der Attribute Characteristic

Der Standard Bluetooth Low Energy sieht für jeden Datenwert vor, ein eigenes Attribut zu erstellen. Daraus entsteht der Nachteil, dass sich der Ressourcenverbrauch, Implementierungs- und Wartungsaufwand mit steigenden Datenmengen erhöht. Hat man zusätzlich eine große Anzahl gleichartiger Daten, entstehen redundante Attribute.

Um den Ressourcenverbrauch, Implementierungs- und Wartungsaufwand für eine Vielzahl von Attributen zu verringern, wird in dieser Arbeit nur eine Characteristic für die Lese- und Schreib-Anfrage genutzt. Der Vorteil, nur eine Characteristic zu nutzen, besteht darin, dass die Anzahl der Informationen, die der Server enthält, nicht die Anzahl der Characteristics und dementsprechend den Ressourcenverbrauch beeinflusst. Des Weiteren wird die Implementierung und Wartung einfacher, da die Implementierung und Wartung der Characteristics nicht von der Anzahl der Attribute beeinflusst wird.

Weitere Möglichkeiten, wie das Erstellen einer einzelnen Characteristic für das Lesen und einer weiteren für das Schreiben, führen dazu, dass der Bereich [O] der Prozeduren weggelassen werden kann. Die Auswirkungen einer solchen Änderung müssten untersucht werden. Für diese Arbeit wird darauf verzichtet, da beide Implementierungsarten zu einem geringeren Ressourcenverbrauch führen.

5 Implementation

Für die Umsetzung des Kommunikationsprotokolls sind mindestens zwei Systeme notwendig. Ein Server- und ein Client-System. Die Systeme werden in den folgenden Abschnitten aufgrund der funktionalen Anforderungen entwickelt und beschrieben. Die zwei Systeme bestehen aus unterschiedlichen Gerätekonzepten. Für den Server wird die Funktionalität Bluetooth Low Energy als Ein-Chip-System (System-on-a-Chip) realisiert und für den Client wird die Funktionalität Bluetooth Low Energy in Form eines Moduls bereitstellt.

5.1 Systemanforderungen

In diesem Kapitel werden Anforderungen an die Implementierung des Kommunikationsprotokolls gestellt. Da das System aus zwei Teilsystemen besteht, müssen an die jeweiligen Bestandteile Anforderungen gestellt werden. Die Teilsysteme bestehen zum einen aus einem Bluetooth Low Energie Gerät, welches als Client agiert, und zum anderen aus einem Bluetooth Low Energie Gerät, welches als Server agiert. Im Folgenden werden die Anforderungen an die Teilsysteme aufgelistet.

5.1.1 Anforderungen an den Server für Bluetooth Low Energy

Für die Entwicklung eines Bluetooth Low Energy Servers werden folgende Anforderungen gestellt:

- *A_S0*: Der Bluetooth Low Energy Server muss mindestens eine Bluetooth Low Energy-Verbindung mit einem Client herstellen können.
- *A_S1*: Der Bluetooth Low Energy Server muss für die Kommunikation zum Client die standardisierte Operation Indication des Profils GATT nutzen können.

- *A_S2*: Der Bluetooth Low Energy Server muss mindestens 10 *aktive Anfragen*[1] annehmen können.
- *A_S3*: Der Bluetooth Low Energy Server muss mindestens einen Service zur Verfügung stellen.
- *A_S4*: Der Bluetooth Low Energy Server muss mindestens 10 *aktive Anfragen*[1] nachverfolgen können.
- *A_S5*: Der Bluetooth Low Energy Server muss Lese- und Schreibanfragen, wie vom Kommunikationsprotokoll definiert, empfangen und verarbeiten können.
- *A_S6*: Der Bluetooth Low Energy Server muss *aktive Anfragen*[1], die länger als 20 Sekunden brauchen, abbrechen und dem Client dies mitteilen können.
- *A_S7*: Der Bluetooth Low Energy Server sollte zum Kompilierzeitpunkt einstellbare Werte enthalten.
- *A_S8*: Der Bluetooth Low Energy Server sollte einstellbare Werte für die maximale Anzahl der *aktiven Anfragen* enthalten.
- *A_S9*: Der Bluetooth Low Energy Server muss *aktive Anfragen*[1] mit einem Wert oder einem Status beantworten.
- *A_S10*: Der Bluetooth Low Energy Server muss den Erhalt einer Anmeldung der Anfrage bestätigen und verarbeiten.
- *A_S11*: Der Bluetooth Low Energy Server muss den Erhalt einer Anmeldung der Anfrage mit einem Fehlercode bestätigen, wenn nicht genug Ressourcen vorhanden sind.
- *A_S12*: Der Bluetooth Low Energy Server muss feste Größen für eine Schreib- und Lese-Anfrage definieren.
- *A_S13*: Der Bluetooth Low Energy Server muss den Erhalt einer Anmeldung der Anfrage mit einem Fehlercode bestätigen, wenn die feste Größe der Anfrage nicht der definierten Größe für eine Schreib- und Lese-Anfrage entspricht.
- *A_S14*: Der Bluetooth Low Energy Server sollte die Bereiche der Anmeldung der Anfrage auf Existenz prüfen.

[1] Aktive Anfragen = Anfragen, die vom Server empfangen wurden und verarbeitet werden.

- *A_S15*: Der Bluetooth Low Energy Server sollte, wenn Bereiche der Anmeldung der Anfrage nicht existieren, dem Client dies mitteilen können.
- *A_S16*: Der Bluetooth Low Energy Server muss das *zeitlich versetztes Senden und Empfangen*[2] von Daten unterstützen.

5.1.2 Anforderungen an den Client für Bluetooth Low Energy

Für die Entwicklung eines Bluetooth Low Energy Client werden folgende Anforderungen aufgestellt:

- *A_C0*: Der Bluetooth Low Energy Client muss eine Bluetooth Low Energy Verbindung zu einem Server herstellen können.
- *A_C1*: Der Bluetooth Low Energy Client muss für die Kommunikation zu einem Server die standardisierte Operation *Write* des Profils GATT nutzen können.
- *A_C2*: Der Bluetooth Low Energy Client muss Schreib- und Lese-Anfragen, wie sie von dem Kommunikationsprotokoll definiert sind, an den Server stellen können.
- *A_C3*: Der Bluetooth Low Energy Client muss dem Server das Senden von Indikationen erlauben können.
- *A_C4*: Der Bluetooth Low Energy Client muss Indikationen empfangen können.
- *A_C5*: Der Bluetooth Low Energy Client sollte vom Anwender durch eine Menüführung steuerbar sein.
- *A_C6*: Der Bluetooth Low Energy Client sollte die eigenen Anfragen sowie die Antworten des Servers in der Python-Konsole ausgeben können.
- *A_C7*: Der Bluetooth Low Energy Client wird die eigenen Anfragen mit den Antworten des Servers automatisiert zuordnen.

5.2 Umgebung des Server-Systems

In diesem Abschnitt wird die Umgebung für das Server-System vorgegeben. Dafür werden die Hardware und die dazu benötigte Software vorgestellt.

[2] Zeitlich versetzte Senden und Empfangen = Asynchrone Kommunikation.

5.2.1 Mikrocontroller

Das Kommunikationsprotokoll für das Server-System soll auf dem Mikrocontroller nRF52840 [16] von Nordic Semiconductor umgesetzt werden. Dieser Mikrocontroller verfügt über einen Cortex-M4F-Prozessor, 256 KB RAM und einen integrierten Flash-Speicher von 1 MB.

5.2.2 Software Entwicklungskit

Als Software Entwicklungskit soll das nRF5 SDK von Nordic Semiconductor in der Version v17.0.2 genutzt werden.

Die Dokumentation für das Software-Entwicklungskit von Nordic Semiconductor ist auf der Unternehmensseite [15] zu finden. Dieses Software-Entwicklungskit liefert eine Vielzahl an Programmstapel (Softwarestack) für Funkprotokolle. Diese werden von dem Unternehmen Nordic Semiconductor als SoftDevices [17] bezeichnet. Ein SoftDevice beinhaltet z. B. ein Softwarepaket für den Standard Bluetooth Low Energy.

Ein wesentlicher Vorteil des SoftDevices ist, dass die Themen Verbindungsaufbau, Datensicherheit, Datenintegrität, Fehlerbehebung und Fehlersicherung, gemäß dem Standard Bluetooth Low Energy in dem SoftDevice von Nordic Semiconductor implementiert und zertifiziert sind. Als solch ein SoftDevice wurde das SoftDevice S140 in der Version 7.0.1 ausgewählt. Dieses ist kompatibel mit dem Mikrocontroller nRF52840. Es bietet den Softwarestack für Bluetooth Low Energy in der Rolle eines sogenannten Central- oder Peripheral-Gerätes. Die komplette Liste der Eigenschaften des SoftDevice S140 ist auf der Unternehmensseite [17] zu finden. Einige Eigenschaften sind im Folgenden aufgeführt.

- Qualifiziert für Bluetooth 5.1
- Durchsatz von bis zu 2 Mbit/s
- Long Range
- CSA #2
- LE Secure Connections
- Custom UUID
  und weitere

5.2.3 Entwicklungsboard

Für die Entwicklung soll das Entwicklungsboard *PAN1780 ETU* [9] von Panasonic Industry Europe GmbH zum Einsatz kommen. Das Entwicklungsboard *PAN1780 ETU* enthält das Modul *PAN1780-Modul*. Dieses enthält den Mikrocontroller nRF52840 von Nordic Semiconductor und ermöglicht das schnelle Bauen und Testen von Prototypen. Um das Entwicklungsboard *PAN1780 ETU* mit dem Software Entwicklungskit nRF5 von Nordic Semiconductor zu nutzen, existiert eine Anleitung im Dokument PAN1780 Modul Integration Guide [14].

5.2.4 Integrierte Entwicklungsumgebung

Die Software soll in der integrierten Entwicklungsumgebung (IDE) Embedded Studio[11] von SEGGER Microcontroller GmbH entwickelt werden. Emmbedded Studio ermöglicht das Verwalten, Erstellen, Testen und Bereitstellen von Anwendungen für eingebettete Systeme (Embedded Systems). Das Unternehmen Nordic Semiconductor empfiehlt die Nutzung der IDE Embedded Studio und stellt eine kommerzielle Lizenz kostenfrei zur Verfügung.

5.2.5 Informationen des Server-Systems

Nach der Meinung der Autoren Pahl und Beitz sind „Um der steigenden Komplexität heutiger Produkte Rechnung zu tragen, [...] Module als weiteres Element der Produktgestaltung zu berücksichtigen.“[2, 738]. Des Weiteren kommen die Autoren Eitelwein, Malz und Weber zu dem Ergebnis „Modularisierung führt zu einer deutlichen Erhöhung des Unternehmenserfolges.“[7]. Mit diesen Aussagen ist es naheliegend, dass auch in der Praxis das Konzept der Modularisierung etabliert ist. Übertragen auf diese Arbeit, bedeutet dies, dass ein System aus mehreren Modulen aufgebaut ist. Daraus ergibt sich, dass Daten des Systems auch auf unterschiedlichen Modulen verteilt liegen[3]. Für die Kommunikation der Module werden häufig serielle Datenbusse genutzt, als Beweis dient der Bus STD[4] [1]. Weitere Datenbusse, wie I2C, UART oder SPI werden ebenfalls für die Kommunikation genutzt.

[3] In dieser Arbeit wird als Modul die Kombination aus Sensoren und Mikrocontroller definiert.
[4] Der Bus STD ist ein modulares Aufbau- und Verbindungsschema für 8 Bit Mikroprozessor Kartensysteme.

In dieser Arbeit sollen die Informationen des Server-Systems auf unterschiedlichen Modulen vorliegen. Die Module sollen über einen seriellen Datenbus miteinander verbunden sein. Für die Umsetzung werden die Module durch ein eigens entwickeltes Testsystem realisiert. Das Testsystem kann über die Schnittstelle UART angefragt werden. Die Entwicklung des Testsystems ist im Abschnitt 6.2 beschrieben. In der Abbildung 5.1 wird der Aufbau des Servers mit dem Testsystem skizziert.

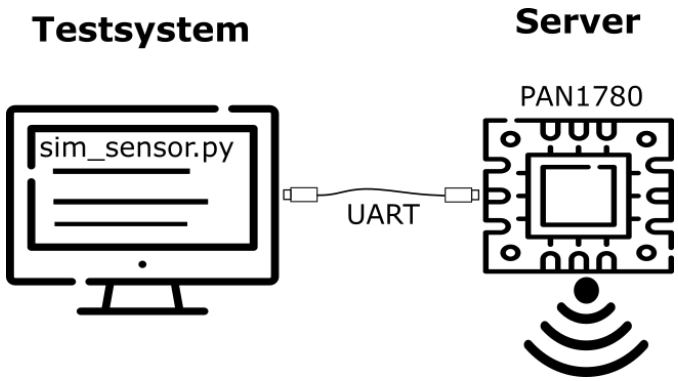

Abbildung 5.1: Der Bluetooth Low Energy Server verbunden mit dem Testsystem [8]

5.3 Implementierung des Server-Systems

Als Basis für die Implementierung des Server-Systems dient das im Abschnitt 4.2 definierte Kommunikationsprotokoll. Im Folgenden wird das Kommunikationsprotokoll für den Mikrocontroller nRF52840 des Unternehmens Nordic Semiconductor implementiert. Im Abschnitt D.3 ist der dazugehörige Quellcode enthalten.

5.3.1 Komponenten

Für die Implementierung werden die in Abbildung 5.2 gezeigten Komponenten benötigt. In den folgenden Abschnitten werden ihre Funktionen näher erläutert.

Die Implementierung des Standard Bluetooth Low Energy des Unternehmens Nordic Semiconductor arbeitet mit Ereignissen (Event), die von dem SoftDevice ausgelöst werden. Zu einem Ereignis kommt es, wenn z. B. ein Client eine Anfrage sendet oder Daten über eine Schnittstelle empfangen wurden. Ereignisse werden über Rückruffunktionen (Callbacks) vom SoftDevice an das Anwenderprogramm weitergeleitet. Hierbei kann es

Abbildung 5.2: Komponenten des Servers

sich auch um Interrupt-Handler handeln. Für die Implementierung werden die in Abbildung 5.3 gezeigten Ereignisse genutzt.

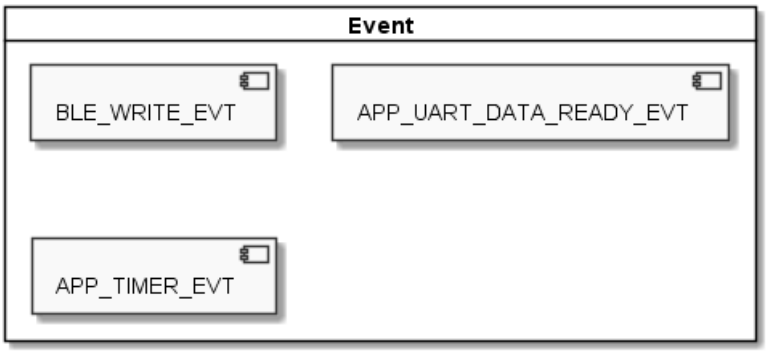

Abbildung 5.3: Ereignisse des Servers

BLE_WRITE_EVT

Dieses Ereignis wird genutzt, um die Anfragen des Client anzunehmen und abzuspeichern.

APP_UART_DATA_READY_EVT

Dieses Ereignis wird genutzt, um die Daten, die über die Schnittstelle UART empfangen wurden, anzunehmen und abzuspeichern.

APP_TIMER_EVT

Dieses Ereignis wird genutzt, um die Anfragen, die eine bestimmte Zeit überschritten haben (Timeout), vorzumerken.

Die Ereignisse des SoftDevice, die auch Interrupt-Handler enthalten können, sollen schnellstmöglich beendet werden. Dafür werden zeitintensive Handlungen als Aufgaben (Task) in der Hauptroutine ausgeführt. Für die Implementierung werden die in Abbildung 5.4 gezeigten Aufgaben genutzt.

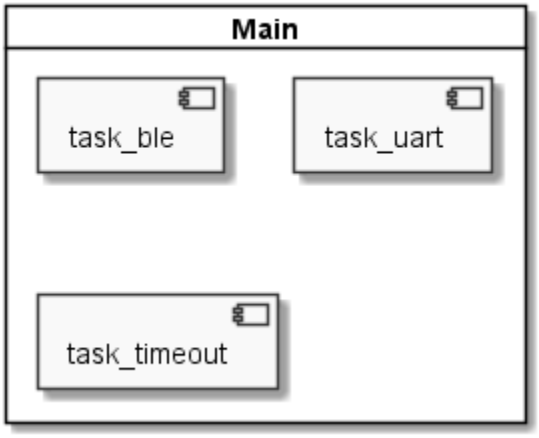

Abbildung 5.4: Aufgaben der Servers

task_ble

Diese Aufgabe wird genutzt, um die Anfragen des Client zu bearbeiten.

task_uart

Diese Aufgabe wird genutzt, um die Daten, die über die Schnittstelle UART empfangen wurden, zu bearbeiten.

task_timeout

Diese Aufgabe wird genutzt, um die Anfragen, die eine bestimmte Zeit überschritten haben, abzubrechen.

Für die Abarbeitung der Anfragen und Daten werden die in Abbildung 5.5 gezeigten Warteschlangen (Queue) genutzt.

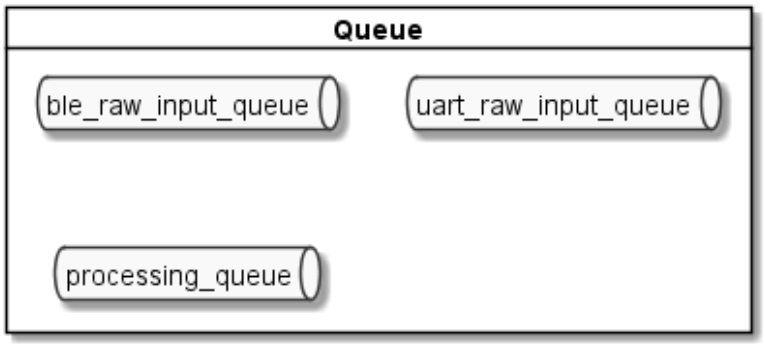

Abbildung 5.5: Warteschlangen des Servers

ble_raw_input_queue
: Diese Warteschlange wird genutzt, um die Anfragen des Client zu speichern.

uart_raw_input_queue
: Diese Warteschlange wird genutzt, um die eingehenden Daten, die über die Schnittstelle UART empfangen werden, zu speichern.

processing_queue
: Diese Warteschlange wird genutzt, um die Anfragen, die aktuell in Bearbeitung sind, zu speichern.

Die Warteschlangen werden nach dem Prinzip *First in-First out* (FiFo) realisiert.

5.3.2 Bereiche der Anfragen und Antworten

In diesem Abschnitt werden die Bereiche der Lese-, Schreib-Anfrage, sowie der Antworten definiert.

Während eine Anfrage aus den Bereichen [O], [KEY] und [VALUE] (siehe Unterabschnitt 4.2.1) besteht, enthält eine Antwort die Bereiche [O], [KEY] und [VALUE] oder [STATUS] (siehe Unterabschnitt 4.2.3). Um die Verarbeitung der Daten zu vereinfachen, werden alle Daten als vorzeichenlose ganzzahlige Datentypen mit einer Größe von 8 Bits (uint8_t) definiert. In diesem Format können die Daten aus dem Bluetooth Low Energy Puffer gelesen werden und über die Schnittstelle UART versendet werden. Die Anzahl der Bereiche bestimmt die Summe der enthaltenen Daten.

In der folgenden Tabelle sind die Größen der Bereiche für die Lese-, Schreib-Anfragen sowie für die Antwort dieser Implementierung definiert.

Tabelle 5.1: Die Definitionen der Bereiche für die Implementierung des asynchronen Kommunikationsverfahrens.

Bereiche		**O**	**KEY**	**VALUE**	**STATUS**
Anfragen	Größe	1	3	2	1
	Beispiel	0x52	0x875FC3	0x4200	0x70
Lesen		X	X		
Schreiben		X	X	X	
Antwort mit Status		X	X		X
Antwort mit Wert		X	X	X	

Für eine *Lese-Anfrage* wird der Bereich [O] als 0x52 definiert, dies entspricht dem ASCII-Zeichen „R“. Dementsprechend wird eine *Schreib-Anfrage* als 0x57 definiert, dies entspricht dem ASCII-Zeichen „W“.

Es ergibt sich für eine *Lese-Anfrage* eine Größe von vier Byte und für eine *Schreib-Anfrage* eine Größe von sechs Byte. Für die *Antwort mit Status* ergibt sich eine Größe von vier Byte und für eine *Antwort mit Wert* eine Größe von sechs Byte.

5.3.3 Ereignis BLE_WRITE_EVT

Das Ereignis BLE_WRITE_EVT hat die Aufgabe, die Anfrage des Client anzunehmen und abzuspeichern. Dafür wird zuerst eine Eingangsprüfung gemacht, darauf folgt die Bestätigung der Anfrage. Wenn aus der Eingangsprüfung kein Fehler hervorgeht, wird die Anfrage in die Queue ble_raw_input_queue gespeichert.

Die Bestätigung der Anfrage soll dem Client das Ergebnis der Eingangsprüfung mitteilen (siehe Unterabschnitt 4.2.2).

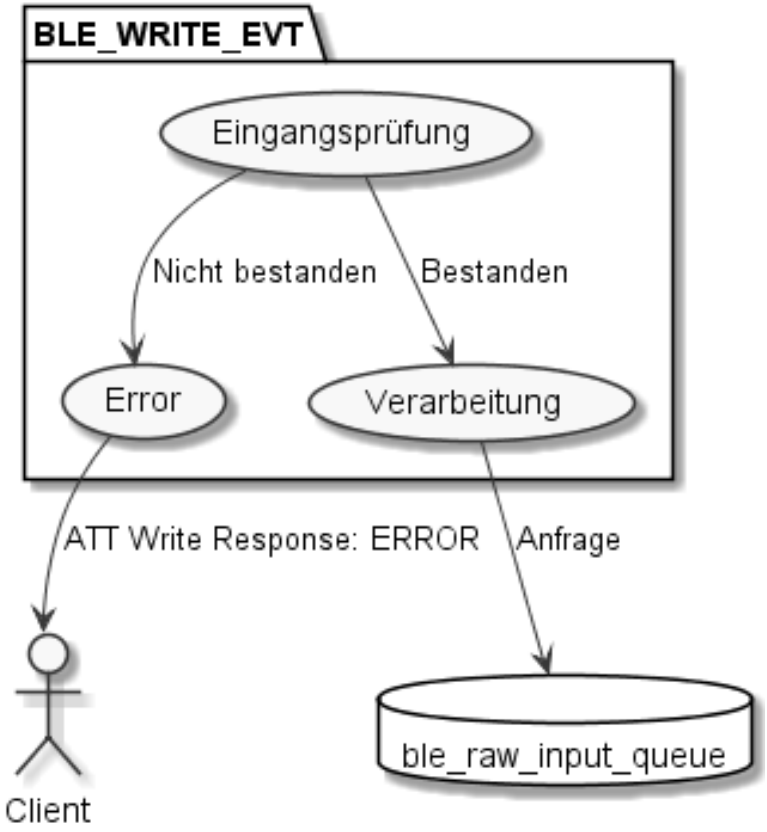

Abbildung 5.6: Ablauf des Ereignisses BLE_WRITE_EVT

Eingangsprüfung

Die Eingangsprüfung konzentriert sich auf das erste Prüfen der Anfrage des Client. Dafür wird die Anfrage des Client zuerst auf die Länge geprüft, die dafür möglichen Längen sind in dem Unterabschnitt 5.3.2 definiert.

Darüber hinaus werden die Ressourcen geprüft. In dieser Implementierung sind die Ressourcen die verfügbaren Plätze der Queues. Anfragen sollen nicht angenommen werden, wenn die Queue ble_raw_input_queue oder processing_queue ihre maximale Anzahl der Einträge erreicht hat. Die maximale Anzahl der Einträge der Queues sind zur Kompilierzeit einstellbar.

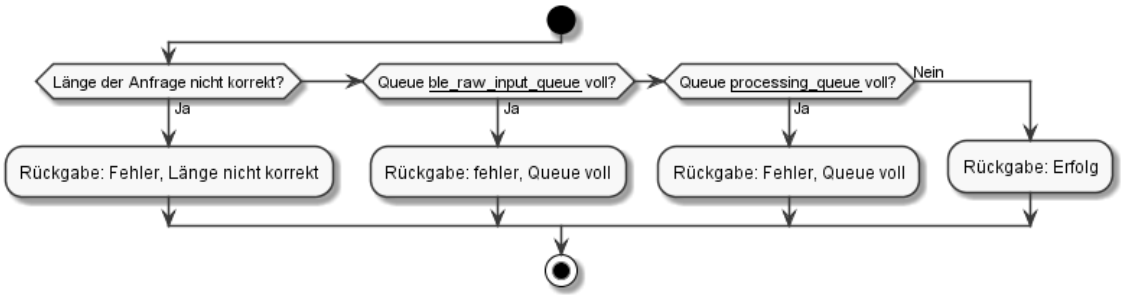

Abbildung 5.7: Ablauf der Eingangsprüfung

Bestätigung der Anfrage

Um dem Client mitzuteilen, dass kein Fehler vorliegt, wird die ATT-Prozedur „ATT_Write_Response()“ gesendet. Um dem Client wiederum mitzuteilen, dass ein Fehler vorliegt, wird die ATT-Prozedur „ATT_ERROR_RSP()“ mit den folgenden Fehlercodes gesendet.

Fehlerfall	Fehlercode
Falsche Länge der Anfrage	BLE_GATT_STATUS_ATTERR_INVALID_ATT-_VAL_LENGTH (0x010D)
Queue voll	BLE_GATT_STATUS_ATTERR_CPS_PROC_ALR-_IN_PROG (0x01FE)

Tabelle 5.2: Fehlercodes für die Bestätigung der Anfrage

5.3.4 Ereignis APP_UART_DATA_READY_EVT

Das Ereignis APP_UART_DATA_READY_EVT wird ausgeführt, wenn Daten über die Schnittstelle UART empfangen wurden. Im Ereignis werden zum einen die Daten gesammelt und zum anderen die gesammelten Daten der Queue uart_raw_input_queue hinzugefügt, sodass diese von der Aufgabe task_uart bearbeitet wird.

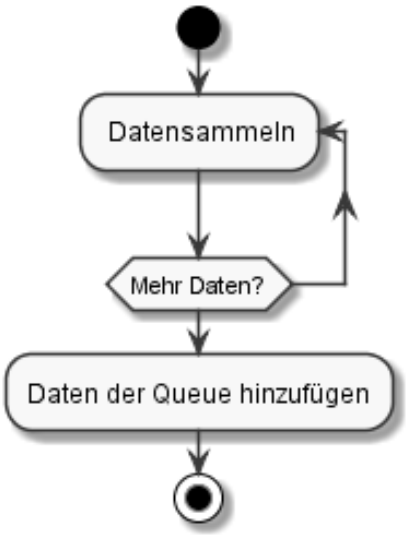

Abbildung 5.8: Ablauf des Ereignisses APP_UART_DATA_READY_EVT

5.3.5 Ereignis APP_TIMER_EVT

Für die aktiven Anfragen werden Anwendungs-Timer erstellt. Diese laufen ab, wenn die Anfragen nicht beantwortet wurden. Das Ereignis APP_TIMER_EVT wird ausgeführt, wenn ein Anwendungs-Timer abgelaufen ist. Es hat die Aufgabe, die abgelaufenen Anfragen an die Aufgabe task_timeout zuzuleiten, damit diese bearbeitet werden.

5.3.6 Aufgabe task_ble

Die Aufgabe task_ble wird ausgeführt, wenn in der Queue ble_raw_input_queue Anfragen enthalten sind. Die Aufgabe entnimmt eine Anfrage aus der Queue ble_raw_-input_queue und führt eine Bereichsprüfung aus. Die Idee hinter der Bereichsprüfung ist, die Anfrage vor dem Weiterbearbeiten zu prüfen. Dafür werden in dieser Arbeit die Bereiche [O] und [KEY] der Anfragen auf Existenz geprüft.

Für den Bereich [O] ist das Lesen und Schreiben zulässig. Die Existenzprüfung des Bereichs [KEY] ist abhängig von dem gewählten System, welches als Datenträger gewählt wird. Im Unterabschnitt 6.2.2 wird für das Testsystem ein Datenträger definiert. Dementsprechend wird auch die Existenzprüfung des Bereichs [KEY] im Unterabschnitt 6.2.2 definiert. Kommt es bei der Bereichsprüfung zum Fehler, wird dem Client eine Antwort

mit Status gesendet. Im Status ist ein Fehlercode enthalten. Zulässige Anfragen werden weiter verarbeitet.

Die Weiterverarbeitung leitet die zulässigen Anfragen an die endgültige Verarbeitung weiter. In diesem System werden die zulässigen Anfragen über die Schnittstelle UART ausgesendet. Zum weiteren Verfolgen werden die zulässigen Anfragen der Queue processing_queue hinzugefügt und dienen dem Nachverfolgen der aktiven Anfragen. Für jede aktive Anfrage wird ein Timer erstellt. Wenn Anfragen nach einer bestimmten Zeit nicht beantwortet sind, werden die Anfragen aus der Queue processing_queue entfernt.

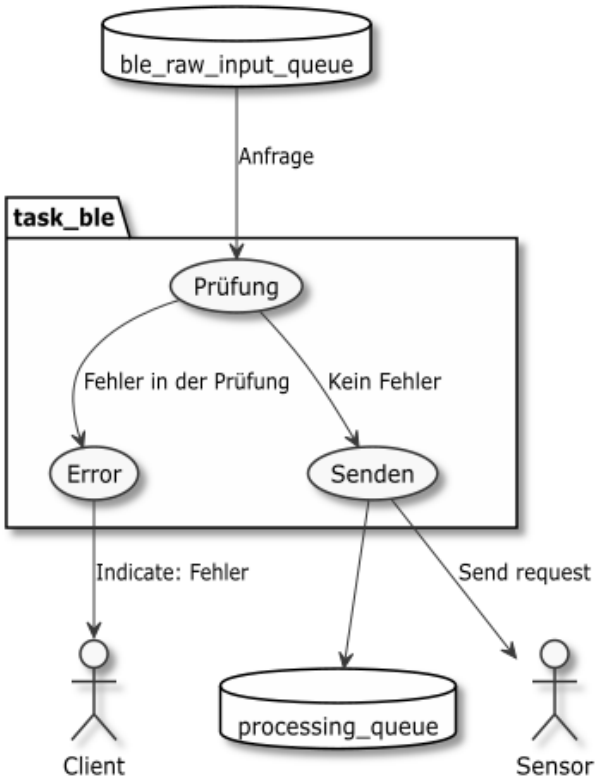

Abbildung 5.9: Ablauf der Aufgabe task_ble

5.3.7 Aufgabe task_uart

Die Aufgabe task_uart wird ausgeführt, wenn in der Queue uart_raw_input_queue Daten enthalten sind. Die Aufgabe entnimmt die Daten aus der Queue und prüft, ob eine passende Anfrage in der Queue processing_queue enthalten ist (matching). Sobald

die passende Anfrage in der Queue processing_queue gefunden ist, stoppt die Aufgabe den dazugehörigen Anwendungs-Timer, entfernt die Anfrage aus der Queue processing_-queue und sendet die Antwort der Anfrage an den Client.

Ist keine passende Anfrage zu den Daten in der Queue processing_queue enthalten, werden die Daten verworfen. Wenn das Matching fehlschlägt, kann dies zwei Gründe haben. Entweder hat die Anfrage zu lange gedauert und wurde durch die Aufgabe task_timeout bearbeitet oder fehlerhafte Daten werden von der Schnittstelle UART empfangen. In den beiden Fällen wird keine Antwort auf die Anfrage des Client erwartet, sodass das Verwerfen der Daten keine negativen Auswirkungen hat.

5.3.8 Aufgabe task_timeout

Die Aufgabe task_uart wird ausgeführt, wenn Anfragen des Client abgelaufen sind. Die Aufgabe task_uart entfernt die Anfrage aus der Queue processing_queue und sendet dem Client die Antwort auf die Anfrage mit dem Fehlercode IND_TIMEOUT.

5.4 Umgebung des Client-Systems

In der Praxis werden üblicherweise mobile Endgeräte mit passender Anwendungssoftware als Client-Systeme genutzt. Da eine Anwendungssoftware spezifisch für ein Betriebssystems und dessen Version entwickelt wird, wurde in dieser Arbeit ein alternativer Weg gewählt.

Das Unternehmen Panasonic Industry Europe GmbH bietet, neben dem Modul *PAN1780*, noch ein weiteres Modul an, das Modul *PAN1780AT* [10]. Dieses Modul enthält den Mikrocontroller nRF52840 [16] von Nordic Semiconductor, auf dem eine Firmware mit dem AT-Befehlssatz vom Unternehmen BlueRadios [3] bereits enthalten ist. Damit können die Funktionen von Bluetooth Low Energy über ein standardisiertes serielles Protokoll von z. B. einem externen Mikrocontroller oder einem Computer sehr einfach genutzt werden. In dieser Arbeit wird das Modul *PAN1780AT* mit einem Computer über die USB-Schnittstelle verbunden. Der Computer verbindet sich über UART mit dem Modul *PAN1780AT*. Sobald die UART-Verbindung hergestellt ist, kann der Computer mit dem AT-Befehlssatz von BlueRadios, das Modul *PAN1780AT* steuern. Für die Steuerung wird

ein Python-Skript entwickelt. Dafür wird der Interpreter CPython in der Version 3.8.6 genutzt.

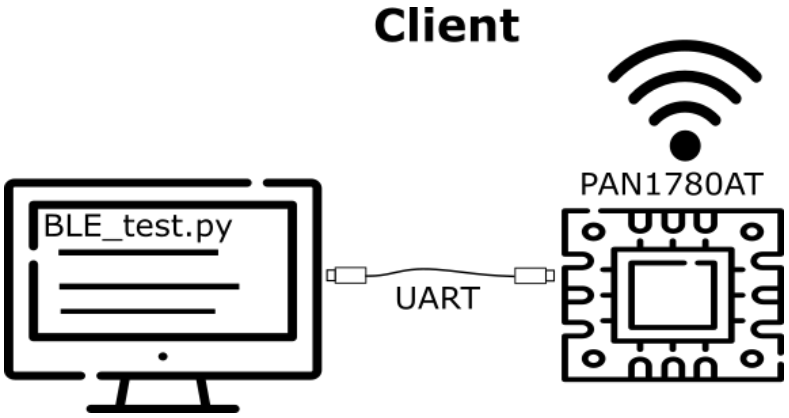

Abbildung 5.10: Darstellung des Bluetooth Low Energy Client, dieser besteht aus dem Python-Skript und dem Bluetooth Low Energy Modul [8].

5.5 Implementierung des Client-System

Für die Implementierung des Client-Systems mit dem Modul *PAN1780AT* werden die folgenden Funktionen mithilfe des AT-Befehlssatzes von BlueRadios umgesetzt.

Das Modul *PAN1780AT* wird durch das Senden von Kommandos gesteuert. Die Kommandos haben die Syntax: „COMMAND“ + cr. Alle Kommandos mit korrekter Syntax werden mit einem OK oder einem ERROR beantwortet. Des Weiteren werden von dem Modul *PAN1780AT* Ereignisse (Event) gesendet, wenn z. B. ein Verbindungsaufbau stattgefunden hat oder eine Antwort auf eine GATT Read-Anfrage erhalten wurde. Für diese Arbeit sind die folgenden Event relevant:

CONNECT/DISCONNECT
: Das Event wird gesendet, wenn eine Verbindung zu einem Bluetooth Low Energy Gerät aufgebaut oder geschlossen wurde.

GATT_DONE
: Das Event wird gesendet, wenn eine GATT Anfrage erfolgreich oder mit einem Fehler abgeschlossen wurde.

GATT_VAL

Das Event wird gesendet, wenn GATT Daten erhalten wurden. Dies geschieht durch eine vorangegangene GATT Read-Anfrage oder durch das Erhalten von Indikationen oder Notifikationen.

In den nachfolgenden Abschnitten wird der Ablauf des Python-Skripts *BLE_test.py* erläutert. Im Listing D.1 ist der dazugehörige Quellcode enthalten.

5.5.1 Serieller Verbindungsaufbau

Nach dem Öffnen der seriellen Verbindung von einem Computer zum *PAN1780AT* wird die serielle Verbindung zum Modul *PAN1780AT* durch das Senden des Kommandos „AT“ geprüft. Als Antwort wird ein „OK“ erwartet.

5.5.2 Aufbau einer Verbindung mit Bluetooth Low Energy

Für den Aufbau einer Verbindung mit Bluetooth Low Energy wird das Kommando „ATDMLE“ genutzt. Als Parameter wird die Adresse des Bluetooth Low Energy Servers übergeben. Eine erfolgreiche Verbindung zum Bluetooth Low Energy Server wird durch das Event „CONNECT“ mitgeteilt.

5.5.3 Auslesen von Events

Um die Events zuverlässig auszulesen, wird ein leichtgewichtiger Prozess (Thread) erstellt, dessen Aufgabe darin besteht, den Puffer der seriellen Verbindung auszulesen. Die ausgelesenen Daten werden nach relevanten Events gefiltert. Die relevanten Events sind im Abschnitt 5.5 aufgelistet. Diese relevanten führen zur Ausgabe von Informationen im Terminal (siehe Anhang C).

5.5.4 Funktionalitäten des Client-Systems

Damit eine Benutzbarkeit des Client-Systems gegeben ist, können von einem Anwender Aktionen über eine Menüführung ausgewählt werden. In der Abbildung 5.11 sind die Aktionen, die zur Verfügung stehen, dargestellt und werden im Folgenden kurz erklärt.

0) *Ausgabe des Menüs*
   Das Menü wird ausgegeben, sodass die Aktionen wieder sichtbar sind.

1) *Indikationen erlauben*
   Diese Aktion beschreibt den Client Characteristic-Configuration Descriptor (CCCD), sodass der Client dem Server die Erlaubnis gibt, Indikationen zu senden.

4) *Stellen einzelner Anfragen*
   Diese Aktion erlaubt es, aus vorgegebenen validen Anfragen eine zu wählen und zu senden. Es kann zwischen Lesen und Schreiben gewählt werden.

5) *Zurücksetzten des Moduls*
   Mit dieser Aktion kann das Modul BlueRadios zurückgesetzt werden.

6) *Erstellen einer Verbindung mit dem Server*
   Diese Aktion kann eine Verbindung zwischen dem Modul PAN1780AT und dem Bluetooth Low Energy Server herstellen.

8) *Trennen und beenden*
   Diese Aktion führt eine Trennung von dem Modul BlueRadios und dem Bluetooth Low Energy Server durch, beendet die serielle Verbindung zum Modul BlueRadios und das Programm.

Weitere Aktionen werden im Unterabschnitt 6.3.3 erklärt. Die übrigen Aktionen aus der Abbildung 5.11 werden für die Verifikation genutzt und sind deshalb kein Bestandteil des Client-Systems.

```
--------------------------------------------------------------------------------
                        0) Print Menu
                        1) Enable indications
                        2) Burst Test
                        3) Test modus
                        4) Send single request
                        5) Reset BlueRadios
                        6) Reconnect to BLE device
                        7)
                        8) Disconnect and exit
--------------------------------------------------------------------------------
```

Abbildung 5.11: Ausgabe der Funktionsauswahl des Client-Systems

6 Verifikation

In diesem Kapitel soll die Implementierung des Server- und Client-Systems getestet werden. Das Ziel ist es zu zeigen, dass mit dem Kommunikationsprotokoll asynchron kommuniziert werden kann. Dafür wird eine Testmethode genutzt, die in dieser Arbeit als *Burstmodus* bezeichnet wird.

Ein weiteres Ziel dieses Kapitels ist es, ein Testsystem zum Simulieren von Latenzen zu entwickeln, sowie das Client-System zu erweitern, damit automatische Testfälle durchlaufen werden können.

6.1 Testumgebung

In der Abbildung 6.1 werden die Systeme gezeigt, die zum Testen des asynchronen Kommunikationsprotokolls benötigt werden. Die Testumgebung besteht aus dem Testsystem und dem Server, sowie aus dem Client.

6.2 Testsystem zum Simulieren von Latenzzeiten

Durch unterschiedliche Auslastung von Peripherien entstehen in der Praxis beim Zugriff auf Informationen variierende Latenzzeiten. Diese werden in dieser Arbeit von einem Testsystem simuliert. Das Simulieren von Latenzzeiten hat den Vorteil, dass wiederholbare Tests durchgeführt werden können, sodass Testfälle unter gleichbleibenden Bedingungen ausgeführt werden. Zum Simulieren von Latenzzeiten wird das Server-System mit einem Computer über die serielle Schnittstelle UART verbunden. Auf dem Computer werden Latenzzeiten mit dem Python-Skript *sim_ sensor.py* simuliert, dieses ist im Anhang D enthalten. Als Interpreter wird CPython in der Version 3.8.6 genutzt.

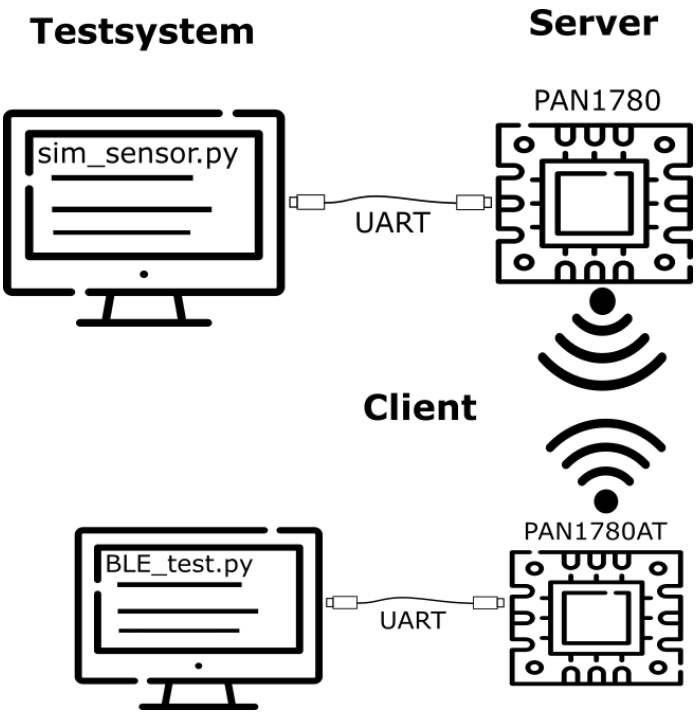

Abbildung 6.1: Darstellung der Systeme für die asynchrone Kommunikation [8].

6.2.1 Funktionalitäten des Testsystems

Die grundlegende Funktion des Python-Testsystems ist es, Informationsanfragen des Server-Systems anzunehmen und zu beantworten. Die Anfragen sollen über die Schnittstelle UART angenommen werden. Eine Anfrage besteht aus der gewünschten Operation (lesen oder schreiben), aus einer Adresse und aus einem beliebigen Wert. An jede Adresse wird eine feste Wartezeit gekoppelt, bis die Anfrage beantwortet wird. Nach der Annahme der Anfrage wird ihr ein fortlaufender Wert zugeteilt. Nach Ablaufen der Wartezeit wird der Wert als Rückgabewert genutzt. Die Rückgabewerte werden für die Auswertung genutzt.

Da das Python-Testsystem mehrere Anfragen parallel verarbeiten kann, beeinträchtigen sich die Latenzzeiten der Anfragen nicht. Durch verschiedene Latenzzeiten der Anfragen können schnelle Anfragen, langsamere Anfragen überholen. Dieses Überholen von Anfragen beweist dann, dass das implementierte Kommunikationsprotokoll asynchron kommuniziert.

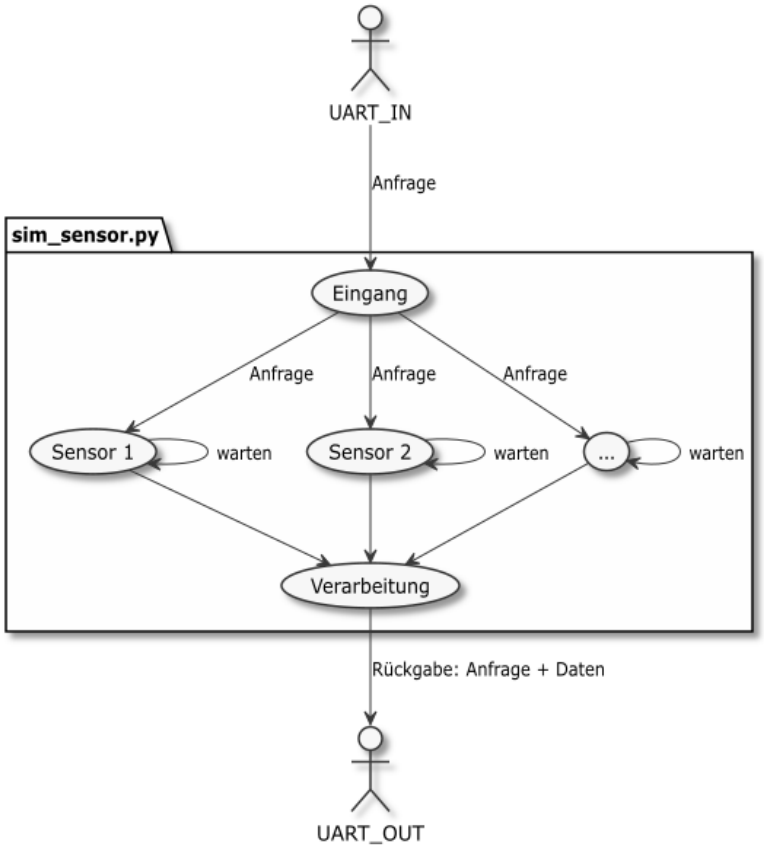

Abbildung 6.2: Ablauf des Python-Skripts *sim_ sensor.py* (siehe Listing D.2)

6.2.2 Aufbau des Testsystems für die simulierten Latenzen

Für den Aufbau des Testsystems wird die Datenstruktur aus der Tabelle A.1 genutzt. Die Datenstruktur besteht aus den folgenden drei Komponenten:

- **Modul**
- **Untermodul**
- **Sensor**

Die Komponenten dienen der einfacheren Adressierung der Anfragen. Jede Komponente wird mit einem Byte adressiert. Die Adressierung der Komponenten wird für den Bereich [KEY] der Bluetooth Low Energy Anfragen sowie für die UART-Anfragen genutzt. Die Adresse setzt sich aus der Reihenfolge Modul, Untermodul und Sensor zusammen.

Beispiel 6.1 *Um den Sensor 1(0x3F) des Untermoduls 1. Zelle(0x8C) des Moduls Akkumodul(0x7A) anzufragen, ergibt sich die Adresse 0x7A8C3F.*

Für die Entwicklung des Testsystems werden die folgenden Anforderungen definiert:

- *A_ T0*: Das Testsystem muss Daten senden können.
- *A_ T1*: Das Testsystem muss Daten empfangen können.
- *A_ T2*: Das Testsystem muss nur Anfragen an Sensoren beantworten, die in der Tabelle A.1 definiert sind.
- *A_ T3*: Das Testsystem muss gültige Anfragen mit einem Wert beantworten.
- *A_ T4*: Das Testsystem muss mehr als eine Anfrage zeitgleich bearbeiten.
- *A_ T5*: Das Testsystem muss an die Anfragen feste Wartezeiten von minimal 5 ms und maximal 900 ms koppeln.

Während des Entwickelns wurden die Anforderungen des Testsystems verifiziert. Da der Fokus der Arbeit nicht auf dem Testsystem liegt, werden diese Tests nicht ausführlich dargestellt.

6.3 Client-Systems

In dieser Arbeit wird das Client-System um die Funktionen *Testfälle* und *Burstmodus* erweitert. Damit kann das Client-System genutzt werden, um Testfälle automatisiert ablaufen zu lassen. Für die Testfälle wurde aus Zeitgründen keine automatisierte Auswertung implementiert, sodass die Ergebnisse für den Testdurchlauf anhand von Textausgaben ausgewertet werden.

Bevor jedoch das Client-System zum Testen des Server-Systems erweitert wird, müssen die grundlegenden Funktionen des Client-System verifiziert werden.

6.3.1 Verifikation des Client-Systems

Für die Verifikation der grundlegenden Funktionen des Client-Systems wurden die folgenden Testfälle ausgeführt:

- *TF_ C1* Indikation aktivieren.
- *TF_ C2* Eine einzelne Lese-Anfrage senden.
- *TF_ C3* Eine einzelne Schreib-Anfrage senden.
- *TF_ C4* Eine Indikation erhalten und ausgeben.

Die Testabläufe sind in der Beschreibung der Testfälle definiert. Diese sind im Anhang B zu finden.

Für die Verbindung vom Computer zum Modul *PAN1780AT* sowie die Verbindung vom Modul *PAN1780AT* zum Bluetooth Low Energy Server werden keine zusätzlichen Testfälle durchgeführt, da eine Überprüfung dieser Verbindungen in dem Python-Skript integriert ist. Die Prüfungen werden zu Beginn ausgeführt und informieren den Anwender über Fehlerzustände.

6.3.2 Auswertung des Client-Systems

Nachdem die Testfälle durchgeführt wurden, ohne einen Fehlerzustand hervorzurufen, kann das Client-System zum automatisierten Testen des Server-Systems genutzt werden.

6.3.3 Erweiterung des Client-Systems

Das bestehende Client-System wird um die folgenden Aktionen erweitert, damit das Server-System automatisiert getestet werden kann.

2) *Burstmodus*
   Diese Aktion sendet 5 bis 100 Anfragen nacheinander. Die Anzahl der Anfragen kann von dem Anwender bestimmt werden.

3) *Testfall Modus*
   Diese Aktion führt vorgegebene Testfälle aus.

6.4 Verifikation des Server-Systems

Im folgenden Abschnitt wird das Server-System mithilfe der Funktionen *Burstmodus* und *Testfälle* getestet. Das Ziel dieses Abschnittes ist es zu beweisen, dass das entwickelte Kommunikationsprotokoll des Server-Systems asynchrone Kommunikation zwischen einem Server- und Client-System ermöglicht. Zusätzlich werden die Anforderungen aus dem Unterabschnitt 5.1.1 durch Testfälle geprüft.

6.4.1 Burstmodus

Im Folgenden wurde die Testfunktion *Burstmodus* des Client-Systems ausgeführt. In dem Test wurden fünf Anfragen nacheinander an das Server-System gesendet. In der Abbildung 6.3 ist die Ausgabe von diesem Test dargestellt.

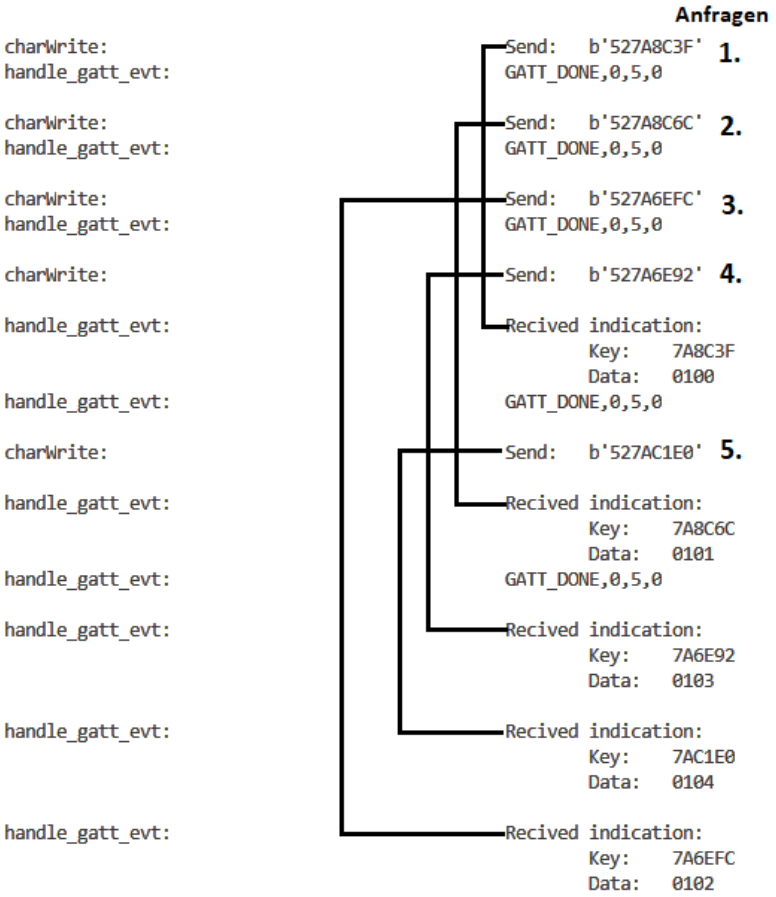

Abbildung 6.3: Der Burstmodus mit 5 Anfragen hintereinander

Es lässt sich anhand der Abbildung 6.3 beweisen, dass die Kommunikation asynchron ist. Zum einen führt die erste Anfrage nicht zum Blockieren weiterer Kommunikation, sodass die zweite, dritte und vierte Anfrage gestellt werden können, während die erste Anfrage auf die Beantwortung wartet. Zum anderen lässt sich die asynchrone Kommunikation durch die Reihenfolge der ankommenden Indikationen beweisen. Anhand der dritten Anfrage wird dies besonders deutlich. Durch die lange Latenzzeit der dritten Anfrage werden die vierte und fünfte Anfrage vor der dritten Anfrage beantwortet. Das Überholen der Anfragen beweist, dass das Kommunikationsprotokoll asynchron kommuniziert.

6.4.2 Testfälle

Die folgenden Testfälle werden im Modus Testfälle nacheinander ausgeführt. Die Beschreibungen der Testfälle ist im Anhang B enthalten. Im Anhang C wird die Ausgabe von diesem Test gezeigt.

- TF_S0 Senden einer Schreib-Anfrage
- TF_S1 Senden einer Lese-Anfrage
- TF_S2 Senden einer Anfrage mit falschem Bereich [O]
  - TF_S2_0 : Schreib-Anfrage mit einer Lesen-Operation
  - TF_S2_1 : Schreib-Anfrage mit einer ungültigen Operation
  - TF_S2_2 : Lese-Anfrage mit einer Schreib-Operation
  - TF_S2_3 : Lese-Anfrage mit einer ungültigen Operation
- TF_S3 Senden einer Anfrage mit falschem Bereich [KEY]
  - TF_S3_0 : Schreib-Anfrage mit einer zu langen Adresse
  - TF_S3_1 : Schreib-Anfrage mit einer ungültigen Adresse
  - TF_S3_2 : Schreib-Anfrage mit einer zu kurzen Adresse
  - TF_S3_3 : Lese-Anfrage mit einer zu langen Adresse
  - TF_S3_4 : Lese-Anfrage mit einer ungültigen Adresse
  - TF_S3_5 : Lese-Anfrage mit einer zu kurzen Adresse

- TF_S4 Senden einer Anfrage mit falschem Bereich [VALUE]
  - TF_S4_0 : Schreib-Anfrage mit zu vielen Daten
  - TF_S4_1 : Schreib-Anfrage mit zu wenigen Daten
  - TF_S4_2 : Schreib-Anfrage ohne Daten

Als Auswertung der Testfälle wurden die Ausgaben des Python-Skripts mit den erwarteten Ergebnissen der Testfälle verglichen. Jede Ausgabe der Testfälle beginnt mit einer Überschrift, diese identifiziert den Testfall. Als Beispiel wird der Testfall TF_S0 aus der Abbildung C.5 näher ausgeführt. Der dazugehörige Testfall ist im Unterabschnitt B.2.1 beschrieben. In dem Testfall TF_S0 wird das Ergebnis Received indication: Key: 875FC3 Status: 00 erhalten. Ein Vergleich zeigt, dass dieses Ergebnis dem erwarteten Ergebnis des Testfalls im Unterabschnitt B.2.1 entspricht. Dieser Testfall ist dementsprechend positiv ausgefallen. Durch den Testfall TF_S0 können die Anforderungen *A_S0*, A_S1, *A_S3*, *A_S5*, *A_S9* und *A_S10* geprüft werden.

Mit der Funktion *Testfälle* können automatisiert Testfälle durchlaufen werden. Die Ergebnisse der Testfälle müssen mit den erwarteten Ergebnissen verglichen werden. Mit den Testfällen werden die Anforderungen aus dem Unterabschnitt 5.1.1 getestet. Für das entwickelte Kommunikationsprotokoll des Server-Systems aus Abschnitt 5.3 sind die Testfälle alle positiv ausgefallen.

7 Bewertung und Schlussbetrachtung

In dieser Arbeit wurde ein asynchrones Kommunikationsprotokoll für den Standard Bluetooth Low Energy auf Basis der Ebene GATT entwickelt und implementiert.

Durch die Analyse der Verfahren des Standards Bluetooth Low Energy konnten Einschränkungen erarbeitet werden, die durch die direkte Kopplung von Anfrage und Antwort entstehen. Auf Grund dieser synchronen Kommunikation kann zwischen einer Anfrage und der Antwort keine weitere Kommunikation stattfinden, da die Bestätigung der Anfrage an die Antwort gekoppelt ist. Für Systeme mit langen Latenzen, wie bei einem Zugriff auf Daten, führt dies zum langen Blockieren der Kommunikation.

Das asynchrone Kommunikationsprotokoll entkoppelt die Bestätigung von der Antwort. Auf diese Weise ist es möglich, während des Wartens auf eine Antwort weitere Kommunikation zu führen. Sobald die Daten oder die Rückmeldung für die Antwort vorhanden sind, wird die Antwort gesendet. Somit werden schnelle Anfragen nicht von langsameren Anfragen beeinträchtigt und können diese sogar überholen.

Durch die Nutzung der standardisierten Verfahren kann die Umsetzung des asynchronen Kommunikationsprotokolls ohne eine Veränderung des Bluetooth Low Energy Host- oder Controller-Systems vorgenommen werden. Dies konnte in der Arbeit mit Hilfe der zwei unterschiedlichen Entwicklungsarten des Server- und Client-Systems gezeigt werden. Das Kommunikationsprotokoll für das Server-System konnte auf einem Mikrocontroller implementiert werden, der die Bluetooth Low Energy Funktionalität als Ein-Chip-System (System-on-a-Chip) realisiert. Außerdem konnte bei dem Client-System das Kommunikationsprotokoll auf einem System implementiert werden, welches die Bluetooth Low Energy Funktionalität in Form eines einfach zu nutzenden Moduls mit einer AT-Kommando-Schnittstelle bereitstellt.

7.1 Offene Punkte und Probleme

Neben der asynchronen Kommunikation werden in dieser Arbeit auch Probleme behandelt, die durch eine große Menge an Informationen entstehen. Werden viele Informationen bereitgestellt, müsste in einer klassischen Implementierung für jede Information eine eigene Characteristic erstellt werden. Daraus würde ein hoher Speicherverbrauch entstehen. Durch die Nutzung nur einer Characteristic und einer angepassten Adressierung kann der Speicherverbrauch stark reduziert werden.

In der Umsetzung des Kommunikationsprotokolls wurden für die Adressierung drei Byte gewählt, die über 16 Millionen unterschiedliche Adressen ermöglichen. Die Idee für diese viel zu große Adressierung entstand in den Anfangszeiten dieser Arbeit, sodass eine Änderung dieser Adressen zu einem hohen Aufwand geführt hätte. Allerdings hat die Adressierung kaum Einfluss auf die Struktur des asynchronen Kommunikationsprotokolls, sodass dies hier nur als Hinweis dient.

Für das Testsystem wurde ein Ansatz mit Threads gewählt. Jedoch zeigte sich in unterschiedlichen Tests, dass sich die Anzahl der maximalen Threads auf unterschiedlichen Systemen unterscheidet. Nach der maximalen Anzahl der Threads werden keine weiteren Anfragen von dem Testsystem bearbeitet. Die nicht bearbeiteten Anfragen des Testsystems werden von dem Server nach der Timeoutzeit mit dem Status $0xF3$ beantwortet. Dieses Phänomen tritt häufig während der Burstmodus-Tests auf. Ein Fehlerzustand des asynchrone Kommunikationsprotokoll ist dies nicht, da durch die Antwort der Anfrage mit dem Status $0xF3$ die Anfrage korrekt abgearbeitet wird. Dieser Fehler betrifft nur das Testsystem und nicht die Struktur des asynchronen Kommunikationsprotokolls, sodass die Suche nach der Fehlerursache zurückgestellt wurde. Durch das Neustarten des Python-Skripts wird die Anzahl der Threads zurückgesetzt und die Fehlerwirkung tritt erst nach dem erneuten erreichen der maximalen Anzahl der Threads auf.

7.2 Ausblicke

Anzumerken ist, dass durch die Umsetzung der asynchronen Verbindung davon auszugehen ist, dass mehr Daten pro Zeit versendet werden, als ohne die Umsetzung des asynchronen Kommunikationsprotokolls. Dies kann zu einem höheren Energieverbrauch führen. Da dies jedoch nur eine Vermutung ist, könnte eine weitere Arbeit untersuchen,

inwieweit sich der Energieverbrauch durch die Nutzung des asynchronen Kommunikationsprotokoll verändert. Daraufhin kann untersucht werden, welche Vorteile durch die Nutzung des asynchronen Kommunikationsprotokolls und der Kommunikation ohne das asynchrone Kommunikationsprotokoll entstehen. Diese Hybridlösung könnte zum einen zur Optimierung des Energieverbrauchs für nicht zeitkritische Anfragen und zum anderen zu einer verbesserten Performance für zeitkritische Anfragen führen.

Als Erweiterung wäre es möglich, Anfragen mit Prioritäten zu versehen, um dementsprechend darauf zu reagieren. Ein weiterer Ansatz wäre, mehrere Anfragen zusammenzufügen, sodass die Anfragen exakt der Standardgröße MTU von 20 Byte entsprechen. Auf diese Weise kann ein optimales Verhältnis von Paket-Overhead und Daten erreicht werden. Zusätzlich werden somit weniger Pakete versendet, was den Energieverbrauch verringert. Auf Basis des entwickelten asynchronen Kommunikationsprotokolls könnten in der Zukunft weitere interessante Arbeiten aufsetzen.

7.3 Fazit

In dieser Arbeit wird die Kommunikation der Ebene GATT des Standards Bluetooth Low Energy untersucht und mit den Ergebnissen ein Konzept für ein asynchrones Kommunikationsprotokoll entwickelt. Die Asynchronität, Funktionalität und Umsetzbarkeit des Kommunikationsprotokoll wird anhand einer konkreten Implementierung des Kommunikationsprotokolls und durch die Verifikation der Implementierung gezeigt.

Mit dem asynchronen Kommunikationsprotokoll für den Standard Bluetooth Low Energy können die Einschränkungen für Systeme, die beim Zugriff auf Daten Latenzen verursachen, behoben werden. Eine derartige Implementierung kann für modular aufgebaute Geräte oder für Geräte, die beim Zugriff auf Daten Latenzen erzeugen, genutzt werden, um das Problem der Kopplung von Anfrage zu Antwort zu lösen. Die Kopplung in Kombination mit den Latenzen der Geräte führt zu andauernden Blockaden der Kommunikation beim Standard Bluetooth Low Energy. Mit dem asynchronen Kommunikationsprotokoll ist asynchrone Kommunikation, die mit den standardisierten Verfahren der Ebene GATT erfolgt, möglich. Die asynchrone Kommunikation entkoppelt die Antwort von der Anfrage, sodass kein Blockieren weiterer Kommunikation entsteht und die Latenzzeiten parallel ablaufen können. Letztendlich führt das zu einer schnelleren Kommunikation für modular aufgebaute Geräte oder für Geräte, die beim Zugriff auf Daten Latenzen erzeugen.

Literaturverzeichnis

[1] Ieee standard for an 8-bit microcomputer bus system: Std bus. *IEEE Std 961-1987*, pages 1–72, 1988.

[2] Beate 1967 Bender, editor. *Pahl/Beitz Konstruktionslehre Methoden und Anwendung erfolgreicher Produktentwicklung*. Springer eBook Collection. Springer Vieweg, 9. auflage edition, 2021.

[3] Inc. BlueRadios. Software. Website. Online erhältlich unter https://www.blueradios.com/software.htm; abgerufen am 22. Januar 2021.

[4] Inc. Bluetooth SIG. Bluetooth Generic Attributes. Website, 2020. Online erhältlich unter https://www.bluetooth.com/de/specifications/gatt/; abgerufen am 22. Januar 2021.

[5] Core Specification Working Group. *Core Specification*, rev. 5.2 edition, 12 2019. Online erhältlich unter https://www.bluetooth.com/de/specifications/bluetooth-core-specification/; abgerufen am 7. Oktober 2020.

[6] Core Specification Working Group. *Core Specification*, 16-bit uuidnumbersdocument edition, 11 2020. Online erhältlich unter https://www.bluetooth.com/specifications/assigned-numbers/; abgerufen am 20. November 2020.

[7] Oliver Eitelwein, Stefanie Malz, and Jürgen Weber. Erfolg durch modularisierung. *Controlling & Management*, 56(2):79–84, Jun 2012.

[8] FreePik. Technology Icon-Paket. Website. Diese Abbildung wurde unter Verwendung von Ressourcen von Flaticon.com erstellt.; Online erhältlich unter https://www.flaticon.com/de/packs/technology-141; abgerufen am 18. Januar 2021.

[9] Panasonic Industry Europe GmbH. PAN1780 - High Performance and Long Range. Website, 2020. Online erhältlich unter https://industry.panasonic.eu/devices/wireless-connectivity/bluetooth-low-

energy-modules/pan1780-high-performance-and-long-range; abgerufen am 7. Oktober 2020.

[10] Panasonic Industry Europe GmbH. PAN1780AT. Website, 2020. Online erhältlich unter https://industry.panasonic.eu/devices/wireless-connectivity/bluetoothr-low-energy-modules/pan1780-high-performance-and-long-range/pan1780at; abgerufen am 7. Oktober 2020.

[11] SEGGER Microcontroller GmbH. Embedded Studio — The All-In-One Embedded Development Solution. Website. Online erhältlich unter https://www.segger.com/products/development-tools/embedded-studio/; abgerufen am 23. Dezember 2020.

[12] IIWissen.info. GFSK (gaussian frequency shift keying). Website, 2015. Online erhältlich unter https://www.itwissen.info/GFSK-gaussian-frequency-shift-keying-GFSK-Modulation.html; abgerufen am 9. November 2020.

[13] J. Liu and M. Cai. Gfsk modulation for ble baseband ic design. In *2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–2, 2017.

[14] Panasonic Industrial Devices Europe GmbH. *Module Integration Guide*, rev. 0.3 edition, 2 2020. Online erhältlich unter https://eu.industrial.panasonic.com/sites/default/pidseu/files/downloads/files/preliminary_wm_pan1780_module_integration_guide.pdf; abgerufen am 7. Oktober 2020.

[15] Nordic Semiconductor. nRF5 SDK Essential. Website. Online erhältlich unter https://www.nordicsemi.com/Software-and-tools/Software/nRF5-SDK; abgerufen am 13. Januar 2021.

[16] Nordic Semiconductor. nRF52840 System on Chip. Website. Online erhältlich unter https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840; abgerufen am 01. Februar 2021.

[17] Nordic Semiconductor. S140 SoftDevice. Website. Online erhältlich unter https://www.nordicsemi.com/Software-and-tools/Software/S140; abgerufen am 15. Januar 2021.

[18] Nordic Semiconductor. Template Application. Website. Online erhältlich unter https://infocenter.nordicsemi.com/index.jsp?topic=%2Fsdk_

nrf5_v16.0.0%2Fble_sdk_app_template.html; abgerufen am 12. Februar 2021.

[19] Nordic Semiconductor. SoftDevice stack architecture. Website, 2019. Online erhältlich unter https://infocenter.nordicsemi.com/topic/sds_s140/SDS/s1xx/ble_protocol_stack/ble_protocol_stack.html; abgerufen am 7. Oktober 2020.

[20] R. Tei, H. Yamazawa, and T. Shimizu. Ble power consumption estimation and its applications to smart manufacturing. In *2015 54th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, pages 148–153, 2015.

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Advertising Das Annoncieren des eigenen Dienstes.

Association model Zuordnungsmodell.

ATT Attribute Protocol.

Authentication Die Authentifizierung.

Callbacks Eine Rückruffunktion.

CCCD Client Characteristic Configuration Descriptor.

Characteristic Descriptor Discovery Das Auffinden des Descriptors einer Characteristic.

Characteristic Discovery Das Auffinden von Characteristics.

Characteristic Value Handles Die Nummer zur Identifizierung einer Characteristic auf dem Attribute-Server.

Client Der Klient.

Connection Die Verbindung.

Connection Handling Das Erstellen und Aufrechterhalten von Verbindungen.

CRC Cyclic Redundancy Check.

Descriptor Die Beschreibung der Characteristics.

Device discovery Die Verhaltensweisen und Methoden für die Geräteerkennung.

Embedded Systems Das eingebettete Systeme.

Encryption Die Verschlüsselung von Verbindungen.

Error Code Der Fehlercode.

Event Das Ereignis.

FiFo Der Ansatz First in First out.

GAP Generic Access Profile.

GATT Generic Attribute Profile.

GFSK Gaussian Frequency-Shift Keying.

IDE Integrated Development Environment.

Indicate Die Benachrichtigung mit Bestätigung.

L2CAP Logical link control and adaptation layer protocol.

LL Link Layer.

MIC Message Integrity Check.

MTU Maximum Transmission Unit.

Notification Die Benachrichtigung ohne Bestätigung.

Pairing Der Austausch und die Verteilung von Sicherheitsschlüsseln.

PHY Physical Layer.

Queue Die Warteschlangen.

Read Die Eigenschaft Lesbar.

Scanning Die Suche nach Kommunikationspartnern.

Security Die Sicherheit.

Sequential Request-Response Protocol Das sequenzielle Anfrage-Antwort Protokoll.

Service Discovery Die Diensterkennung.

SIG Special Interest Group.

SM Security Manager.

Softwarestack Der Programmstapel.

Task Eine Aufgabe, die in der Hauptroutine ausgeführt wird.

Thread Ein leichtgewichtiger Prozess.

Timeout Eine Zeitüberschreitung.

UUID Universally Unique Identifier.

Values Die Daten der Attribute.

Write Die Eigenschaft Schreibbar.

A Anhang

In diesem Anhang sind auf den folgenden Seiten das Bild für drei synchrone und für drei asynchrone Anfragen gezeigt. Damit der Inhalt sichtbar ist, werden die Bilder sehr groß dargestellt. Des weiteren beinhaltet dieser Anhang die Tabelle mit der Datenstruktur des Testsystems.

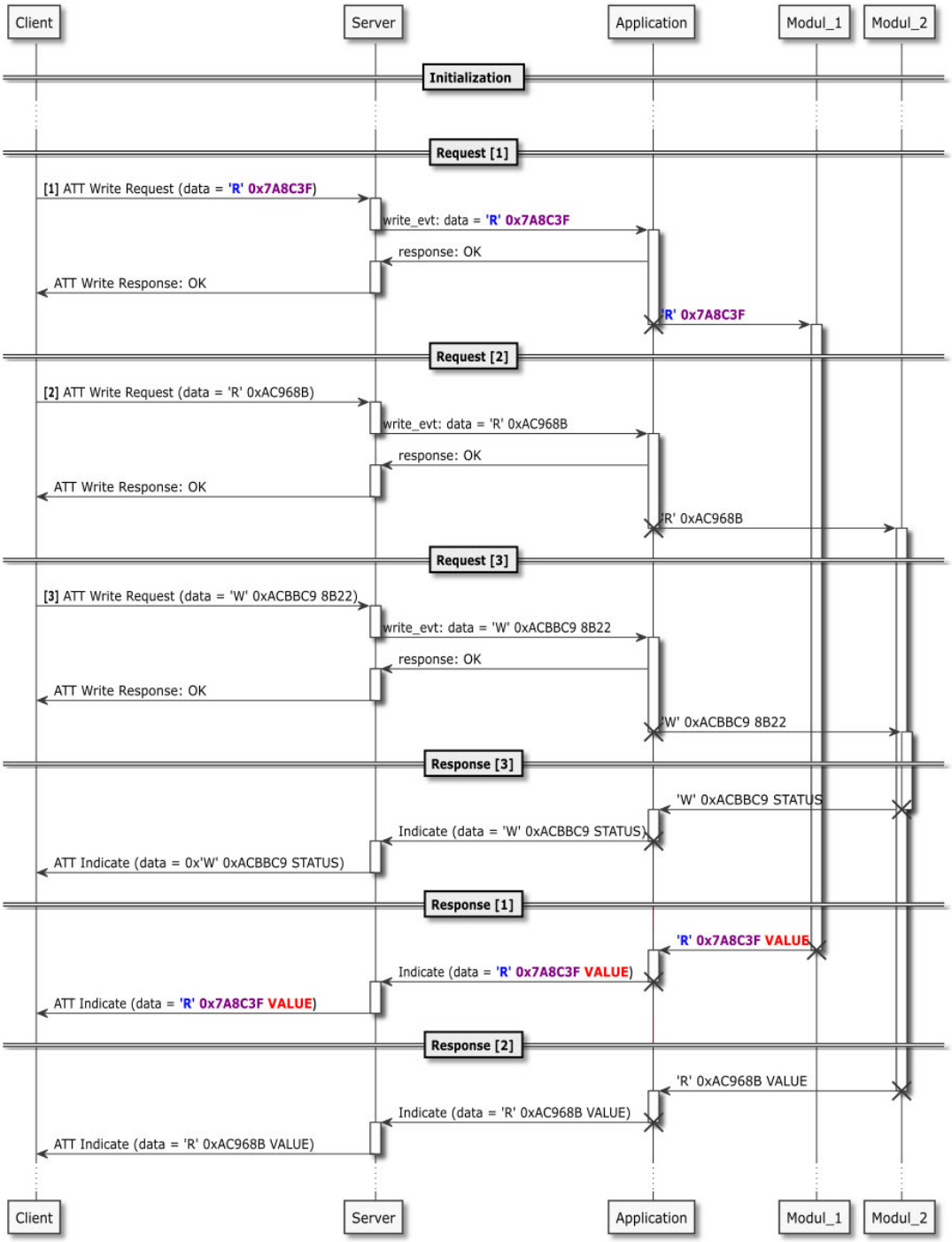

Abbildung A.1: Asynchrone Kommunikation von drei Anfragen.

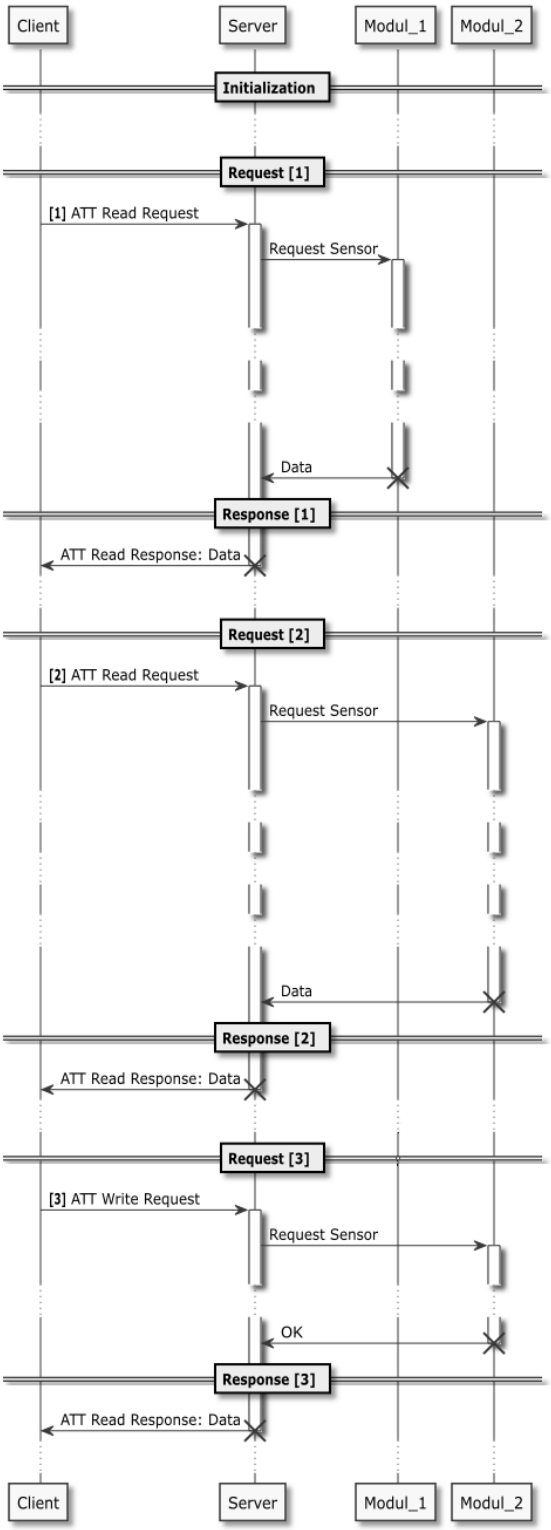

Abbildung A.2: Synchrone Kommunikation von drei Anfragen.

Modul	Addr	Submodul	Addr	Sensor	Addr	Addresse
Akkumodul	0x7a					
		1. Zelle	0x8c			
				1	0x3f	7A8C3F
				2	0x6c	7A8C6C
		2. Zelle	0x6e			
				1	0xfc	7A6EFC
				2	0x92	7A6E92
		3. Zelle	0xc1			
				1	0xe0	7AC1E0
				2	0x35	7AC135
		Temperatur	0x53			
				1	0x31	7A5331
				2	0x21	7A5321
Behältermodul	0xac					
		Füllstand	0xde			
				1	0xc3	ACDEC3
				2	0x8b	ACDE8B
		Feuchtigkeit	0x8b			
				1	0x58	AC8B58
				2	0x6c	AC8B6C
		Druck	0x96			
				1	0x8b	AC968B
				2	0xc1	AC96C1
		Temperatur	0xbb			
				1	0x96	ACBB96
				2	0xc9	ACBBC9
Bürstenmodul	0x87					
		Hall	0x0c			
				1	0x0c	870C0C
				2	0xbb	870CBB
		Feuchtigkeit	0x33			
				1	0x2d	87332D
				2	0xbb	8733BB
		Druck	0x5f			
				1	0x13	875F13
				2	0xc3	875FC3
		Temperatur	0x24			
				1	0x0e	87240E
				2	0xed	8724ED

Tabelle A.1: Datenstruktur für die Adressierung von Anfragen

B Testfälle

In diesem Anhang sind die Testfälle für den Unterabschnitt 6.4.2 enthalten.

In den Testfällen steht ein „X“ immer für eine beliebige Zahl. Das bedeutet, für den Erfolg des Testfalls ist es irrelevant, welche Zahl an der Stelle des „X“ steht.

B.1 Testfälle zum Testen des Clients

B.1.1 TF_C1 Indikation aktivieren

Vorbedingungen:

- Ausführen des Python Skripts BLE_test.py
- Serielle Verbindung zum BlueRadios Device vorhanden.
- Bluetooth Low Energy Verbindung zum Bluetooth Low Energy Server vorhanden.
- Im Eingabefenster des Python Skripts BLE_test.py steht: „Please enter a number:“

Ablauf:

1. Auswählen der Funktion „1) Enable indications“ durch das Eingeben der Nummer 1 in das Eingabefenster des Programms BLE_test.py.

Erwartetes Ergebnis:

GATT_DONE,X,5,0
und
Enabled indications.

B.1.2 TF_C2 Eine einzelne Lese-Anfrage senden

Vorbedingungen:

- Ausführen des Python Skripts BLE_test.py
- Serielle Verbindung zum BlueRadios Device vorhanden.
- Bluetooth Low Energy Verbindung zum Bluetooth Low Energy Server vorhanden.
- Im Eingabefenster des Python Skripts BLE_test.py steht: „Please enter a number:“

Ablauf:

1. Auswählen der Funktion „4) Send single request“ durch das Eingeben der Nummer 4 in das Eingabefenster des Programms BLE_test.py.
2. Rückgabe:
   Single request
   Choose an Operation: (R | W):
3. Auswählen der Lese-Anfrage durch das Eingeben des Buchstaben R in das Eingabefenster.
4. Rückgabe:
   Choose a key from 0 to 24:
5. Auswahl eines Keys von 0 bis 24 durch das Eingeben einer Zahl von 0 bis 24 in das Eingabefenster.
6. Rückgabe:
   Send: b’52XXXXXX’
   GATT_DONE,X,5,0
7. 500 ms Warten
8. Indikation erhalten

Erwartetes Ergebnis ohne Testsystem:

Recived indication:
Key: XXXXXX
Status: F3
oder, wenn das Testsystem aktiv ist:
Recived indication:
Key: XXXXXX
Data: XXXX

B.1.3 TF_C3 Eine einzelne Schreib-Anfrage senden

Vorbedingungen:

- Ausführen des Python Skripts BLE_test.py
- Serielle Verbindung zum BlueRadios Device vorhanden.
- Bluetooth Low Energy Verbindung zum Bluetooth Low Energy Server vorhanden.
- Im Eingabefenster des Python Skripts BLE_test.py steht: „Please enter a number:“

Ablauf:

1. Auswählen der Funktion „4) Send single request“ durch das Eingeben der Nummer 4 in das Eingabefenster des Programms BLE_test.py.
2. Rückgabe:
   Single request
   Choose an Operation: (R | W):
3. Auswählen der Schreib-Anfrage durch das Eingeben des Buchstaben W in das Eingabefenster.
4. Rückgabe:
   Choose a key from 0 to 24:

5. Auswahl eines Keys von 0 bis 24 durch das Eingeben einer Zahl von 0 bis 24 in das Eingabefenster.
6. Rückgabe:
   Send: b'57XXXXXXXXXX'
   GATT_DONE,X,5,0
7. 500 ms Warten
8. Indikation erhalten

Erwartetes Ergebnis ohne Testsystem:

Recived indication:
Key: XXXXXX
Status: F3
oder, wenn das Testsystem aktiv ist:
Recived indication:
Key: XXXXXX
Status: 00

B.1.4 TF_C4 Eine Indikation erhalten und ausgeben

Vorbedingungen:

- Ausführen des Python Skripts BLE_test.py
- Serielle Verbindung zum BlueRadios Device vorhanden.
- Bluetooth Low Energy Verbindung zum Bluetooth Low Energy Server vorhanden.
- Indikationen aktiviert.
- Im Eingabefenster des Python Skripts BLE_test.py steht: „Please enter a number:“

Ablauf:

1. Senden einer validen Anfrage.
2. Rückgabe:
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Recived indication:
Key: XXXXXX
Data: XXXX

B.2 Testfälle zum Testen des Servers

Zum Beschreiben der Testfälle werden die folgenden Variablen definiert:
tc_value = b"4200"
tc_key = b"875FC3"
asyn_write = b"57"
asyn_read = b"52"

B.2.1 TF_S0 Senden einer Schreib-Anfrage

Anforderungen:

A_S0, A_S1, *A_S3*, *A_S5*, *A_S9*, *A_S10*

Vorbedingungen:

- Serielle Verbindung zum BlueRadios Modul.
- Bluetooth Low Energy Verbindung zum Bluetooth Low Energy Server.
- Indikationen aktiviert.
- Testsystem Aktiv.

Ablauf:

1. Senden einer einzelnen Schreib-Anfrage mit den Daten:
   - **O** = asyn_write
   - **KEY** = tc_key
   - **VALUE** = tc_value
2. Rückgabe:
   Send: b'57875FC34200'
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Recived indication:
Key: 875FC3
Status: 00

B.2.2 TF_S1 Senden einer Lese-Anfrage

Vorbedingungen:

- Serielle Verbindung zum BlueRadios Device.
- Bluetooth Low Energy Verbindung zum Bluetooth Low Energy Server.
- Indikationen aktiviert.
- Testsystem Aktiv.

Ablauf:

1. Senden einer einzelnen Lese-Anfrage. mit den Daten:
   - **O** = asyn_read
   - **KEY** = tc_key
2. Rückgabe:
   Send: b'52875FC3'
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Recived indication:
Key: 875FC3
Data: XXXX

B.2.3 TF_S2 Senden einer Anfrage mit falschem Bereich [O]

Vorbedingungen:

- Serielle Verbindung zum BlueRadios Device.
- Bluetooth Low Energy Verbindung zum Bluetooth Low Energy Server.
- Indikationen aktiviert.
- Testsystem Aktiv.

Ablauf TF_S2_0 : Schreib-Anfrage mit einer Lesen-Operation

1. Senden einer einzelnen Schreib-Anfrage mit den Daten:
   - **O** = asyn_read
   - **KEY** = tc_key
   - **VALUE** = tc_value
2. Rückgabe:
   Send: b'52875FC34200'
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Recived indication:
Key: 875FC3
Status: 70

Ablauf TF_S2_1 : Schreib-Anfrage mit einer ungültigen Operation

1. Senden einer einzelnen Schreib-Anfrage mit den Daten:
   - **O** = b'42'
   - **KEY** = tc_key
   - **VALUE** = tc_value
2. Rückgabe:
   Send: b'42875FC34200'
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Recived indication:
Key: 875FC3
Status: 70

Ablauf TF_S2_2 : Lese-Anfrage mit einer Schreib-Operation

1. Senden einer einzelnen Lese-Anfrage. mit den Daten:
   - **O** = asyn_write
   - **KEY** = tc_key
2. Rückgabe:
   Send: b'57875FC3'
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Recived indication:
Key: 875FC3
Status: 70

Ablauf TF_S2_3 : Lese-Anfrage mit einer ungültigen Operation

1. Senden einer einzelnen Lese-Anfrage. mit den Daten:
   - **O** = b'42'
   - **KEY** = tc_key
2. Rückgabe:
   Send: b'42875FC3'
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Recived indication:
Key: 875FC3
Status: 70

B.2.4 TF_S3 Senden einer Anfrage mit falschem Bereich [KEY]

Vorbedingungen:

- Serielle Verbindung zum BlueRadios Modul.
- Bluetooth Low Energy Verbindung zum Bluetooth Low Energy Server.
- Indikationen aktiviert.

- Testsystem Aktiv.

Ablauf TF_S3_0 : Schreib-Anfrage mit einer zu langen Adresse

1. Senden einer einzelnen Schreib-Anfrage mit den Daten:
   - **O** = asyn_write
   - **KEY** = b'BDBDBDBD'
   - **VALUE** = tc_value
2. Rückgabe:
   Send: b'57BDBDBDBD4200'

Erwartetes Ergebnis:

GATT_DONE Error: Invalid Attribute Value Length

Ablauf TF_S3_1 : Schreib-Anfrage mit einer ungültigen Adresse

1. Senden einer einzelnen Schreib-Anfrage mit den Daten:
   - **O** = asyn_write
   - **KEY** = b'BDBDBD'
   - **VALUE** = tc_value
2. Rückgabe:
   Send: b'57BDBDBD4200'
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Ohne KEY-Check:
Recived indication:
Key: BDBDBD
Status: 00
Mit KEY-Check:
Recived indication:
Key: BDBDBD
Status: 71

Ablauf TF_S3_2 : Schreib-Anfrage mit einer zu kurzen Adresse

1. Senden einer einzelnen Schreib-Anfrage mit den Daten:
   - **O** = asyn_write
   - **KEY** = b'BD'
   - **VALUE** = tc_value
2. Rückgabe:
   Send: b'57BD4200'
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Recived indication:
Key: BD4200
Status: 70

Ablauf TF_S3_3 : Lese-Anfrage mit einer zu langen Adresse

1. Senden einer einzelnen Lese-Anfrage. mit den Daten:
   - **O** = asyn_read
   - **KEY** = b'BDBDBDBD'
2. Rückgabe:
   Send: b'52BDBDBDBD'

Erwartetes Ergebnis:

GATT_DONE Error: Invalid Attribute Value Length

Ablauf TF_S3_4 : Lese-Anfrage mit einer ungültigen Adresse

1. Senden einer einzelnen Lese-Anfrage. mit den Daten:
   - **O** = asyn_read
   - **KEY** = b'BDBDBD'
2. Rückgabe:
   Send: b'52BDBDBD'
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Ohne KEY-Check:
Recived indication:
Key: BDBDBD
Data: XXXX
Mit KEY-Check:
Recived indication:
Key: BDBDBD
Status: 71

Ablauf TF_S3_5 : Lese-Anfrage mit einer zu kurzen Adresse

1. Senden einer einzelnen Lese-Anfrage. mit den Daten:
   - **O** = asyn_read
   - **KEY** = b'BD'
2. Rückgabe:
   Send: b'52BD'

Erwartetes Ergebnis:

GATT_DONE Error: Invalid Attribute Value Length

B.2.5 TF_S4 Senden einer Anfrage mit falschem Bereich [VALUE]

Vorbedingungen:

- Serielle Verbindung zum BlueRadios Modul.
- Bluetooth Low Energy Verbindung zum Bluetooth Low Energy Server.
- Indikationen aktiviert.

- Testsystem Aktiv.

Ablauf TF_S4_0 : Schreib-Anfrage mit zu vielen Daten

1. Senden einer einzelnen Schreib-Anfrage mit den Daten:
   - **O** = asyn_write
   - **KEY** = tc_key
   - **VALUE** = b'BDBDBD'
2. Rückgabe:
   Send: b'57875FC3BDBDBD'

Erwartetes Ergebnis:

GATT_DONE Error: Invalid Attribute Value Length

Ablauf TF_S4_1 : Schreib-Anfrage mit zu wenigen Daten

1. Senden einer einzelnen Schreib-Anfrage mit den Daten:
   - **O** = asyn_write
   - **KEY** = tc_key
   - **VALUE** = b'BD'
2. Rückgabe:
   Send: b'57875FC3BD'

Erwartetes Ergebnis:

GATT_DONE Error: Invalid Attribute Value Length

Ablauf TF_S4_2 : Schreib-Anfrage ohne Daten

1. Senden einer einzelnen Schreib-Anfrage mit den Daten:
   - **O** = asyn_write
   - **KEY** = tc_key
   - **VALUE** = b”
2. Rückgabe:
   Send: b’57875FC3’
   GATT_DONE,X,5,0
3. 500 ms Warten
4. Indikation erhalten

Erwartetes Ergebnis:

Ohne KEY-Check:
Recived indication:
Key: 875FC3
Status: 70
Mit KEY-Check:
Recived indication:
Key: 875FC3
Status: 71

C Dokumentation der Testfälle

In diesem Anhang werden die Ergebnisse der Testfälle für den Unterabschnitt 6.4.2 dargestellt.

C.1 Testfälle des Clients

```
__main__:       Please enter a number: 1



charWrite:                                      Send:   b'2'
handle_gatt_evt:                                GATT_DONE,0,5,0
__main__:       Enabled indications
```

Abbildung C.1: Testfall TC_1 Aktivieren einer Indikation

```
__main__:       Please enter a number: 4


------------------------------------------------------------------------
__main__:       Single request

Choose an Operation: (R | W): R
Choose a key from 0 to 24: 0

charWrite:                                      Send:   b'527A8C3F'
handle_gatt_evt:                                GATT_DONE,0,5,0

handle_gatt_evt:                                Recived indication:
                                                        Key:    7A8C3F
                                                        Data:   0102
------------------------------------------------------------------------
```

Abbildung C.2: Testfall TC_2 Senden einer Lese-Anfrage

```
__main__:       Please enter a number: 4


------------------------------------------------------------------------
__main__:       Single request

Choose an Operation: (R | W): W
Choose a key from 0 to 24: 0

charWrite:                                      Send:   b'577A8C3F4200'
handle_gatt_evt:                                GATT_DONE,0,5,0

handle_gatt_evt:                                Recived indication:
                                                        Key:    7A8C3F
                                                        Status: 00
------------------------------------------------------------------------
```

Abbildung C.3: Testfall TC_3 Senden einer Schreib-Anfrage

C.2 Testfälle des Servers

```
------------------------------------------------------------------------
                        0) Print Menu

                        1) Enable indications

                        2) Burst Test

                        3) Test modus

                        4) Send single request

                        5) Reset BlueRadios

                        6) Reconnect to BLE device

                        7)

                        8) Disconnect and exit

------------------------------------------------------------------------
__main__:       Please enter a number: 3


------------------------------------------------------------------------
__main__:       Testcase mode
```

Abbildung C.4: Starten der Funktion *Testfälle*

```
                                    TF_S0
                                   ----------

tc_send_write_req: START


charWrite:                                  Send:   b'57875FC34200'
handle_gatt_evt:                            GATT_DONE,0,5,0

handle_gatt_evt:                            Recived indication:
                                                Key:    875FC3
                                                Status: 00
tc_send_write_req: END


                                    TF_S1
                                   ----------

tc_send_read_req: START


charWrite:                                  Send:   b'52875FC3'
handle_gatt_evt:                            GATT_DONE,0,5,0

handle_gatt_evt:                            Recived indication:
                                                Key:    875FC3
                                                Data:   0101
tc_send_read_req: END
```

Abbildung C.5: Testfall TF_S0 und TF_S1

```
                                    TF_S2_0
                                   ----------

 tc_send_write_req: START


 charWrite:                                  Send:   b'52875FC34200'
 handle_gatt_evt:                            GATT_DONE,0,5,0

 handle_gatt_evt:                            Recived indication:
                                                 Key:    875FC3
                                                 Status: 70
 tc_send_write_req: END


                                    TF_S2_1
                                   ----------

 tc_send_write_req: START


 charWrite:                                  Send:   b'42875FC34200'
 handle_gatt_evt:                            GATT_DONE,0,5,0

 handle_gatt_evt:                            Recived indication:
                                                 Key:    875FC3
                                                 Status: 70
 tc_send_write_req: END
```

Abbildung C.6: Testfall TF_S2_0 und TF_S2_1

```
                                    TF_S2_2
                                   ----------

tc_send_read_req: START


charWrite:                                  Send:   b'57875FC3'
handle_gatt_evt:                            GATT_DONE,0,5,0

handle_gatt_evt:                            Recived indication:
                                                Key:    875FC3
                                                Status: 70
tc_send_read_req: END


                                    TF_S2_3
                                   ----------

tc_send_read_req: START


charWrite:                                  Send:   b'42875FC3'
handle_gatt_evt:                            GATT_DONE,0,5,0

handle_gatt_evt:                            Recived indication:
                                                Key:    875FC3
                                                Status: 70
tc_send_read_req: END
```

Abbildung C.7: Testfall TF_S2_2 und TF_S2_3

```
                                    TF_S3_0
                                   ----------

tc_send_write_req: START


charWrite:                                  Send:   b'57BDBDBDBD4200'

handle_gatt_evt:                            GATT_DONE Error: Invalid Attribute Value Length
tc_send_write_req: END


                                    TF_S3_1
                                   ----------

tc_send_write_req: START


charWrite:                                  Send:   b'57BDBDBD4200'
handle_gatt_evt:                            GATT_DONE,0,5,0

handle_gatt_evt:                            Recived indication:
                                                Key:    BDBDBD
                                                Status: 00
tc_send_write_req: END
```

Abbildung C.8: Testfall TF_S3_0 und TF_S3_1

```
                                    TF_S3_2
                                   ----------

tc_send_write_req: START


charWrite:                                  Send:   b'57BD4200'
handle_gatt_evt:                            GATT_DONE,0,5,0

handle_gatt_evt:                            Recived indication:
                                                Key:    BD4200
                                                Status: 70
tc_send_write_req: END


                                    TF_S3_3
                                   ----------

tc_send_read_req: START


charWrite:                                  Send:   b'52BDBDBDBD'

handle_gatt_evt:                                    GATT_DONE Error: Invalid Attribute Value Length
tc_send_read_req: END
```

Abbildung C.9: Testfall TF_S3_2 und TF_S3_3

```
                                    TF_S3_4
                                   ----------

tc_send_read_req: START


charWrite:                                  Send:   b'52BDBDBD'
handle_gatt_evt:                            GATT_DONE,0,5,0

handle_gatt_evt:                            Recived indication:
                                                Key:    BDBDBD
                                                Data:   0000
tc_send_read_req: END


                                    TF_S3_5
                                   ----------

tc_send_read_req: START


charWrite:                                  Send:   b'52BD'

handle_gatt_evt:                                    GATT_DONE Error: Invalid Attribute Value Length
tc_send_read_req: END
```

Abbildung C.10: Testfall TF_S3_4 und TF_S3_5

```
                                    TF_S4_0
                                   ----------

tc_send_write_req: START


charWrite:                                  Send:   b'57875FC3BDBDBD'

handle_gatt_evt:                                    GATT_DONE Error: Invalid Attribute Value Length
tc_send_write_req: END


                                    TF_S4_1
                                   ----------

tc_send_write_req: START


charWrite:                                  Send:   b'57875FC3BD'

handle_gatt_evt:                                    GATT_DONE Error: Invalid Attribute Value Length
tc_send_write_req: END
```

Abbildung C.11: Testfall TF_S4_0 und TF_S4_1

```
                                    TF_S4_2
                                   ----------

tc_send_write_req: START


charWrite:                                  Send:   b'57875FC3'
handle_gatt_evt:                            GATT_DONE,0,5,0

handle_gatt_evt:                            Recived indication:
                                                Key:    875FC3
                                                Status: 70

tc_send_write_req: END
```

Abbildung C.12: Testfall TF_S4_2

D Programme

In diesem Anhang sind die Quellcodes enthalten. Die Dokumentation zu den Quellcodes ist in dem Kapitel 5 enthalten.

D.1 Bluetooth Low Energy Client Testsystem

Der Anhang zur Arbeit befindet sich auf der CD und kann beim Erstgutachter eingesehen werden.

```
1  # Autor: M. Pyka
2  # Date: 27.12.20
3  # Name: BLE_test.py
4  # Description:
5  # This programme provides a user interface to the PAN1780AT
6  # BLE module. The programme can be used to perform asynchronous
7  # read and write requests for BLE. In order for the request to
8  # be processed by the BLE server, the BLE server must be
9  # implemented as described in the bachelor thesis
10 # by Martin Pyka. The focus of this programme is to
11 # functionally test the asynchronous communication using
12 # the PAN1780AT.
13
14 import serial
15 import time
16 import threading
17 import sys
18
19 CR_CMD = 0x0D .to_bytes(1, "big")  # CR command in bytes.
20 UART_LOCK = threading.RLock()
```

```
21 # Connection specific AT-Commands.
22 RESET = b"ATRST"
23 STATE = b"ATSLE?"
24 CONNECT = b"ATDMLE"
25 DISCONNECT = b"ATDH,0"
26 RECONNECT = b"ATDMLLE"
27 DISCOVERY = b"ATDILE"
28 LASTCONNADDR = b"ATLCALE?"
29 CANCLE_CON = b"ATDC"
30 ADVERTISING = b"ATDSLE"
31 #               Ble information.
32 # Address of the ble server.
33 BLE_SERVER_ADDR = b"C50A1B6F5899:1"
34 COM_PORT = "COM10"
35 # Handle number for the asyn ble service.
36 BLE_ASYN_HA_NUM = b"16"
37 # Indication handle number for the Power Service.
38 BLE_INDICATION_HA_NUM = b"17"
39
40 #               GATT Specific
41 GATT_READ = b"ATGR,0,"  # Make an GATT read request.
42 GATT_WRITE = b"ATGW,0,"  # Make an GATT write request.
43 # The read and write KEYs for the asynchronous read or write.
44 ASYN_READ = b"52"  # Make an asynchron read request.
45 ASYN_WRITE = b"57"  # Make an asynchron write request.
46 # Value type of an Indication GATT event.
47 value_event_indication = "2"
48
49 #                   KEYs
50 VALID_KEY1 = b"875FC3"  # Buerstenmodul, Druck, Sensor 2.
51 VALID_KEY2 = b"AC968B"  # Behaeltermodul, Druck, Sensor 1.
52 VALID_KEY3 = b"ACBBC9"  # Behaeltermodul, Temperatur, Sensor 2.
53 VALID_KEY4 = b"7AC1E0"  # Akkumodul, 3. Zelle, Sensor 1.
54 VALID_KEY5 = b"7A8C6C"  # Akkumodul, 1. Zelle, Sensor 2.
55 VALID_KEY6 = b"870C0C"  # Buerstenmodul, Hall, Sensor 1.
56 INVALID_KEY1 = b"87"
```

```
57 INVALID_KEY2 = b"875FC3AC968B"
58 INVALID_KEY3 = b"375EC3"
59 INVALID_KEY4 = b"57875FC3"
60 INVALID_KEY5 = b"AC9"
61 # This dict is filled with a combination of the moduls,
62 # sub_moduls and sensors.
63 key_list = [
64     "7A8C3F",
65     "7A8C6C",
66     "7A6EFC",
67     "7A6E92",
68     "7AC1E0",
69     "7AC135",
70     "7A5331",
71     "7A5321",
72     "ACDEC3",
73     "ACDE8B",
74     "AC8B58",
75     "AC8B6C",
76     "AC968B",
77     "AC96C1",
78     "ACBB96",
79     "ACBBC9",
80     "870C0C",
81     "870CBB",
82     "87332D",
83     "8733BB",
84     "875F13",
85     "875FC3",
86     "87240E",
87     "8724ED",
88 ]
89 #               Testcase constants
90 tc_value = b"4200"  # Constant value for the asyn writes.
91 tc_key = b"875FC3"  # Constant key for the asyn reads/writes.
92
```

```
93 #                    FLAGs
94 wait_for_data = False  # Flag showing that a task is waiting
       for an answer.
95 evt_read = False  # Flag showing the wait_for_event is reading
       an event.
96 wait_evt_off = False  # Flag to shutdown the wait_for_event.
97 is_connected = False  # Flag showing the connection state.
98 gatt_done = True  # Some operations need a GATT_DONE to end
       correctly.
99
100 # This function checks if the connection to the ble modul is
       established.
101 def testconnection(ser) -> bool:
102     uart_data = bytes(0)
103     print(sys._getframe().f_code.co_name + ":\tSending AT
       cmd\n")
104     while ser.in_waiting:
105         ser.read()
106     # The AT prefix by itself can be used to check
       communication with the module.
107     ser.write(b"AT" + CR_CMD)
108     time.sleep(1)
109     uart_data = portRead(ser)
110     # It will always respond with an OK.
111     for x in uart_data:
112         if "OK" in x:
113             print(sys._getframe().f_code.co_name + ":\tresult:
       Success!\n")
114             return True
115     print(sys._getframe().f_code.co_name + ":\tresult:
       Failed!\n")
116     return False
117
118
119 # Function for showing an error from the ble modul.
120 def error_resp(error: int) -> str:
```

```
121     error_msg = "Unknown Error"
122     if error == 1:
123         error_msg = "Invalid Parameters"
124     if error == 2:
125         error_msg = "Invalid Role"
126     if error == 3:
127         error_msg = "Invalid State"
128     if error == 4:
129         error_msg = "Invalid Password"
130     if error == 5:
131         error_msg = "Invalid Connection Handle"
132     if error == 6:
133         error_msg = "Configuration Locked"
134     if error == 7:
135         error_msg = "List Error"
136     if error == 8:
137         error_msg = "Hardware Error"
138     if error == 9:
139         error_msg = "No Address Stored"
140     if error == 10:
141         error_msg = "Bluetooth Error"
142     if error == 11:
143         error_msg = "Memory Allocation Error"
144     if error == 12:
145         error_msg = "GATT Request Pending"
146     if error == 13:
147         error_msg = "Not Supported"
148     return error_msg
149
150
151 # Function for showing a gatt specific error.
152 def gatt_done_error(error: int) -> str:
153     error_msg = ""
154     if error == 1:
155         error_msg = "Invalid Attribute Handle"
156     if error == 2:
```

```
157         error_msg = "Read Not Permitted"
158     if error == 3:
159         error_msg = "Write Not Permitted"
160     if error == 4:
161         error_msg = "Invalid PDU"
162     if error == 5:
163         error_msg = "Insufficient Authentication"
164     if error == 6:
165         error_msg = "Request Not Supported"
166     if error == 7:
167         error_msg = "Invalid Offset"
168     if error == 8:
169         error_msg = "Insufficient Authorization"
170     if error == 9:
171         error_msg = "Prepare Queue Full"
172     if error == 10:
173         error_msg = "Attribute Not Found"
174     if error == 11:
175         error_msg = "Attribute Not Long"
176     if error == 12:
177         error_msg = "Insufficient Encryption Key Size"
178     if error == 13:
179         error_msg = "Invalid Attribute Value Length"
180     if error == 14:
181         error_msg = "Unlikely Error"
182     if error == 15:
183         error_msg = "Insufficient Encryption"
184     if error == 16:
185         error_msg = "Unsupported Group Type"
186     if error == 17:
187         error_msg = "Insufficient Resources"
188     if error > 128:
189         error_msg = "Service Specific Error " + str(error)
190     return error_msg
191
192
```

```
193 # Function to handle the known GATT events.
194 def handle_gatt_evt(msg: str):
195     global gatt_done
196 
197     if "GATT_VAL" in msg:  # The ble server send a message.
198         evt_data = msg.split(",")  # Split the message into
    events.
199         value_event_type = evt_data[3]  # Get the event type.
200         value = evt_data[6]  # Get the event value.
201         # Checks if the message is an indication event.
202         if value_event_type == value_event_indication:
203             # Getting the key of the message
204             key = value[2] + value[3] + value[4] + value[5]
205             key = key + value[6] + value[7]
206             print(
207                 "\n"
208                 + sys._getframe().f_code.co_name
209                 + ":\t\t\t\tRecived indication:\n"
210                 + "\t\t\t\t\t\t\tKey:\t"
211                 + key
212             )
213             # The ble server can send an indication with a
    status.
214             if len(value) == 10:
215                 status = value[8] + value[9]
216                 print("\t\t\t\t\t\t\tStatus:\t" + status)
217             # Or the ble server can send an indication with
    data.
218             elif len(value) == 12:
219                 data = value[8] + value[9] + value[10]
220                 data = data + value[11]
221                 print("\t\t\t\t\t\t\tData:\t" + data)
222     # Checks if the message is a GATT_DONE event.
223     if "GATT_DONE" in msg:
224         evt_data = msg.split(",")  # Splits the message.
225         gatt_done = True  # reset the flag.
```

```
226         # Cheks if the GATT_DONE event contains an error.
227         if len(evt_data) > 3:
228             error = evt_data[3]
229             try:
230                 if int(error) != 0:
231                     print(
232                         "\n"
233                         + sys._getframe().f_code.co_name
234                         + ":\t\t\t\t\tGATT_DONE Error: "
235                         + gatt_done_error(int(error))
236                     )
237                     return
238             except:
239                 print("\nError: 2133\n")
240         print(sys._getframe().f_code.co_name)
241         print(":\t\t\t\t" + msg)
242 
243 
244 # Checks if there are known prefixes and handles them
245 def prefix_check(data: list) -> list:
246     global is_connected
247     msg_new = list()
248 
249     for x in data:  # Go through the data and checks for
    prefixes.
250         if "ERROR" in x:
251             err = x.replace("ERROR,", "")
252             print("\n" + sys._getframe().f_code.co_name,
    end="")
253             print(":\t\t\tRecived Error from BlueRadios: ",
    end="")
254             print(error_resp(int(err)))
255         elif "GATT_VAL" in x:  # Check if its a gatt event.
256             handle_gatt_evt(x)
257         elif "GATT_DONE" in x:  # Check if its a gatt event.
258             handle_gatt_evt(x)
```

```
259         elif "CONNECT" in x:  # Check if its a connection
    event.
260             print("\n" + sys._getframe().f_code.co_name,
    end="")
261             print(":\t\t\t\t\t\t" + x)
262             is_connected = True
263         elif "DISCONNECT" in x:  # Check if its a disconncet
    event.
264             print("\n" + sys._getframe().f_code.co_name,
    end="")
265             print(":\t\t\t\t\t\t" + x)
266             is_connected = False
267         elif "MTU" in x:
268             pass
269         elif "BRSP" in x:
270             pass
271         elif "RESET" in x:
272             print("\n" + sys._getframe().f_code.co_name,
    end="")
273             print(":\t\t\t\t\t\tModul reseted")
274             time.sleep(1)
275         elif "CPU" in x:
276             pass
277         else:
278             # If the data is no known prefix return it.
279             msg_new.append(x)
280     # Filter out any empty string.
281     msg_new = list(filter(None, msg_new))
282     return msg_new
283
284
285 # Function to read the uart port.
286 # The known prefixes will be handled automatically.
287 def portRead(ser) -> list:
288     data = bytes(0)  # Clear data buffer.
289
```

```
290     with UART_LOCK:
291         while ser.in_waiting:  # Wait for new uart data.
292             uart_data = ser.read()  # Read port.
293             data += uart_data  # Save to data buffer.
294             time.sleep(0.005)
295     data_string = data.decode("utf-8")  # Convert to string.
296     event_list = data_string.splitlines()  # Split the lines
    and remove line boundaries.
297     event_list = list(filter(None, event_list))  # Filter out
    any empty string.
298     event_list = prefix_check(event_list)  # Filter out any
    known events.
299
300     return event_list
301
302
303 # Function to send any command to the ble modul over uart.
304 def anycmd(ser, ble_cmd: bytes) -> list:
305     global wait_evt_off
306     global wait_for_data
307     global evt_read
308     global gatt_done
309
310     if not ser.is_open:
311         return list("Error")
312     # If user wants to disconnect
313     if ble_cmd == DISCONNECT:
314         wait_evt_off = True  # end the wait_for_event()
    function.
315     wait_for_data = True  # Entering critical section.
316     while evt_read:  # Wait till the critical section is
    reached.
317         time.sleep(0.1)
318     with UART_LOCK:  # Waits for the uart to be free.
319         ser.write(ble_cmd + CR_CMD)  # Sending the command to
    the ble modul.
```

```
320     time.sleep(0.02)
321     readout = portRead(ser)
322     wait_for_data = False  # Leaving critical section.
323 
324     return readout
325 
326 
327 # Function to send any data to a characteristic of the ble
    server.
328 def charWrite(ser, handle: bytes, tpe: bytes, data: bytes):
329     global gatt_done
330     # Wraping the data to the AT cmds of the ble modul.
331     asyn_ble_data = GATT_WRITE + handle + b"," + tpe + b"," +
    data
332 
333     gatt_done = False  # Characteristic write needs a
    GATT_DONE to go on.
334     print("\n" + sys._getframe().f_code.co_name, end="")
335     print(":\t\t\t\t\tSend:\t" + str(data))
336     response = anycmd(ser, asyn_ble_data)  # Sending command.
337     while not gatt_done:  # Some cmd need a GATT_DONE to end
    correctly.
338         response.extend(portRead(ser))  # This takes care of
    that.
339         if "Error" in response:
340             while 1:
341                 print("\n" + sys._getframe().f_code.co_name,
    end="")
342                 print(":\tPort not open\n")
343                 time.sleep(1)
344 
345 
346 # Function that waits for the uart data of the ble modul.
347 def wait_for_event(ser):
348     global evt_read
349     global wait_evt_off
```

```
350     global wait_for_data
351
352     while True:
353         data = bytes(0)
354         if wait_evt_off:
355             return
356         if not wait_for_data:
357             if ser.inWaiting() > 0:
358                 evt_read = True
359                 with UART_LOCK:
360                     while ser.in_waiting:  # Wait for data.
361                         uart_data = ser.read()  # Read port.
362                         data += uart_data  # Save to data
    buffer.
363                         time.sleep(0.005)
364                 data_string = data.decode("utf-8")
365                 event_list = data_string.splitlines()
366                 event_list = list(
367                     filter(None, event_list)
368                 )  # Filter out any empty strings.
369                 event_list = prefix_check(event_list)  #
    Filter out any known events.
370                 if len(event_list) != 0:
371                     # Print out command if unknown events
    occour.
372                     print(
373                         "\n"
374                         + sys._getframe().f_code.co_name
375                         + "\t\t\t\t\tUnknown Command: "
376                         + data_string
377                     )
378                 evt_read = False
379         time.sleep(0.05)
380
381
382 # Function to send a single asynchron read or write request.
```

```
383 def single_request(ser, num: int, operation: bytes):
384     x = num % len(key_list)  # protects from overranged num
385     if operation == ASYN_READ:
386         charWrite(ser, BLE_ASYN_HA_NUM, b"0", ASYN_READ +
    bytes(key_list[x], "utf-8"))
387     elif operation == ASYN_WRITE:
388         charWrite(
389             ser,
390             BLE_ASYN_HA_NUM,
391             b"0",
392             ASYN_WRITE + bytes(key_list[x], "utf-8") +
    tc_value,
393         )
394
395
396 ############ TEST Cases ############
397 # Function to perform a asynchron write request with special
    printing functions.
398 def tc_send_write_req(TC: str, ser, o: bytes, key: bytes,
    value: bytes):
399     print("\n\t\t\t\t\t  " + TC)
400     print("\t\t\t\t\t" + "----------" + "\n")
401     print(sys._getframe().f_code.co_name + ": START\n")
402
403     charWrite(ser, BLE_ASYN_HA_NUM, b"0", o + key + value)
404     time.sleep(1)
405     print(sys._getframe().f_code.co_name + ": END \n")
406
407
408 # Function to perform a asynchron read request with special
    printing functions.
409 def tc_send_read_req(TC: str, ser, o: bytes, key: bytes):
410     print("\n\t\t\t\t\t  " + TC)
411     print("\t\t\t\t\t" + "----------" + "\n")
412     print(sys._getframe().f_code.co_name + ": START\n")
```

```
413     charWrite(ser, BLE_ASYN_HA_NUM, b"0", o + key)   # Send
    valid request
414     time.sleep(1)
415     print(sys._getframe().f_code.co_name + ": END \n")
416
417
418 # Function to perform a bunch of asynchron read/write request
419 # with special printing functions.
420 def tc_burst_test(ser, num: int, operation: bytes):
421     print("\n\n\n\t\t\t\t\t" + sys._getframe().f_code.co_name
    + ": START\n")
422     for i in range(num):
423         x = i % len(key_list)
424         single_request(ser, x, operation)
425     time.sleep(4)
426     print("\n\n\n\t\t\t\t\t" + sys._getframe().f_code.co_name
    + ": END \n")
427
428
429 # Function to print out the user menu.
430 def menu_print():
431     print("\n\n")
432     print("-------------------------------------")
433     print("\t\t\t0) Print Menu\n")
434     print("\t\t\t1) Enable indications\n")
435     print("\t\t\t2) Burst Test\n")
436     print("\t\t\t3) Test modus\n")
437     print("\t\t\t4) Send single request\n")
438     print("\t\t\t5) Reset BlueRadios\n")
439     print("\t\t\t6) Reconnect to BLE device\n")
440     print("\t\t\t7) \n")
441     print("\t\t\t8) Disconnect and exit\n")
442     print("-------------------------------------")
443
444
445 if __name__ == "__main__":
```

```
446     try:
447         ser = serial.Serial(COM_PORT, 115200, rtscts=True)
448     except:
449         print("No Device connected to: " + COM_PORT)
450         input("Terminal will be closed. Click enter.")
451         sys.exit()
452 
453     ok = False  # Helping variable
454     # Testing the serial connection to the ble modul.
455     while testconnection(ser) == False:
456         if anycmd(ser, RESET) == "Error":  # Resets the ble
    modul.
457             print("\nPort not Open")
458             ser.close()  # Closes port to ble modul.
459             time.sleep(2)  # Waits for a short time.
460             ser.open()  # Reopens connection to ble modul.
461         time.sleep(1)
462 
463     anycmd(ser, CONNECT + b"," + BLE_SERVER_ADDR)  # Try to
    connect to ble server.
464     time.sleep(2)
465     while not is_connected:  # Check for connecttion
466         response = anycmd(ser, STATE)  # Gets the current LE
    state of the module.
467         if len(response) != 0:
468             for x in response:
469                 if "OK" in x:
470                     ok = True
471                 elif ok == True:
472                     state = x
473                     ok = False
474             if state[6] == "1":  # If in 'Connecting' process
475                 anycmd(ser, CANCLE_CON)  # Kill process
476             if state[8] == "1":  # If in 'Connected' process
477                 is_connected = True
478                 break
```

```
479             anycmd(
480                 ser, CONNECT + b"," + BLE_SERVER_ADDR
481             )  # Try to connect to ble server.
482             time.sleep(2)
483 
484     wait_thr = threading.Thread(target=wait_for_event,
    args=(ser,))
485     wait_thr.start()
486 
487     menu_print()  # print Menu
488     while True:
489         option = ""
490         option = input(__name__ + ":\tPlease enter a number: ")
491         print("\n")
492         if option == "0":  # Menu print
493             menu_print()
494         elif option == "1":  # Enable indications for response
    to a request
495             # 1. b"2" = uint16_t. 2. b"2" = Enable indications
    = (0x0002) and notification = (0x0001)
496             charWrite(ser, BLE_INDICATION_HA_NUM, b"2", b"2")
497             print(__name__ + ":\tEnabled indications\n")
498         elif option == "2":  # Some Burst test
499             print("----------------------------------------")
500             print(__name__ + ":\tBurst test\n")
501             while True:
502                 try:
503                     operation = input("Choose an Operation: (R
    | W): ")
504                     if "R" == operation:
505                         op = ASYN_READ
506                         break
507                     elif "W" == operation:
508                         op = ASYN_WRITE
509                         break
510                     else:
```

```
511                         print("Sorry, I didn't understand
    that.\n")
512                         continue
513                 except:
514                     print("Sorry, I didn't understand that.\n")
515                     continue
516             while True:
517                 try:
518                     num = int(input("Choose a number of
    requests from 5 to 100: "))
519                 except:
520                     print("Sorry, I didn't understand that.\n")
521                     continue
522                 else:
523                     break
524             tc_burst_test(ser, num, op)
525             print("----------------------------------------")
526         elif option == "3":  # Some Test modes
527             print("----------------------------------------")
528             print(__name__ + ":\tTestcase mode\n")
529
530             ### TF_S0 ### Write req
531             # write correct
532             tc_send_write_req("TF_S0", ser, ASYN_WRITE,
    tc_key, tc_value)
533
534             ### TF_S1 ### Read req
535             # read correct
536             tc_send_read_req("TF_S1", ser, ASYN_READ, tc_key)
537
538             ### TF_S2 ### wrong Operation
539             # write req with read O
540             tc_send_write_req("TF_S2_0", ser, ASYN_READ,
    tc_key, tc_value)
541             # write req with not allowed O
```

```
542             tc_send_write_req("TF_S2_1", ser, b"42", tc_key,
    tc_value)
543             # read req with write 0
544             tc_send_read_req("TF_S2_2", ser, ASYN_WRITE,
    tc_key)
545             # read req with not allowed 0
546             tc_send_read_req("TF_S2_3", ser, b"42", tc_key)
547
548             ### TF_S3 ### Wrong key
549             # key to long
550             tc_send_write_req("TF_S3_0", ser, ASYN_WRITE,
    b"BDBDBDBD", tc_value)
551             # key not in system
552             tc_send_write_req("TF_S3_1", ser, ASYN_WRITE,
    b"BDBDBD", tc_value)
553             # key to short
554             tc_send_write_req("TF_S3_2", ser, ASYN_WRITE,
    b"BD", tc_value)
555             # key to long
556             tc_send_read_req("TF_S3_3", ser, ASYN_READ,
    b"BDBDBDBD")
557             # key not in system
558             tc_send_read_req("TF_S3_4", ser, ASYN_READ,
    b"BDBDBD")
559             # key to short
560             tc_send_read_req("TF_S3_5", ser, ASYN_READ, b"BD")
561
562             ### TF_S4 ### Wrong Value
563             # value to long
564             tc_send_write_req("TF_S4_0", ser, ASYN_WRITE,
    tc_key, b"BDBDBD")
565             # value to short
566             tc_send_write_req("TF_S4_1", ser, ASYN_WRITE,
    tc_key, b"BD")
567             # no value
```

```
568             tc_send_write_req("TF_S4_2", ser, ASYN_WRITE,
    tc_key, b"")
569             print("----------------------------------------")
570         elif option == "4":  # Choosing a single Request
571             print("----------------------------------------")
572             print(__name__ + ":\tSingle request\n")
573             while True:
574                 try:
575                     operation = input("Choose an Operation: (R
    | W): ")
576                     if "R" == operation:
577                         op = ASYN_READ
578                         break
579                     elif "W" == operation:
580                         op = ASYN_WRITE
581                         break
582                     else:
583                         print("Sorry, I didn't understand
    that.\n")
584                         continue
585                 except:
586                     print("Sorry, I didn't understand that.\n")
587                     continue
588             while True:
589                 try:
590                     num = int(input("Choose a key from 0 to
    24: "))
591                 except:
592                     print("Sorry, I didn't understand that.\n")
593                     continue
594                 else:
595                     break
596             single_request(ser, num, op)
597             time.sleep(3.5)
598             print("----------------------------------------")
599         elif option == "5":  # Reset BlueRadios
```

```
600             anycmd(ser, RESET)
601         elif option == "6":  # Reconnect to last ble device
602             anycmd(ser, RECONNECT)
603         elif option == "7":
604             pass
605         elif (
606             option == "8"
607         ):  # Disconnect from Bluetooth Low Energy and close
    Serial connection
608             try:
609                 if is_connected == True:
610                     anycmd(ser, DISCONNECT)  # Disconnect
611                 ser.close()
612             except:
613                 print("Error closing serial port.")
614                 sys.exit()
615             else:
616                 input("Terminal will be closed. Click enter.")
617                 sys.exit()
618         else:
619             print("Invalid Operation!\n")
```

Listing D.1: Quellprogramm des Bluetooth Low Energy Testsystems

D.2 Simulieren von Latenzen

Der Anhang zur Arbeit befindet sich auf der CD und kann beim Erstgutachter eingesehen werden.

```
1 # Autor: M. Pyka
2 # Date: 27.12.20
3 # Name: sim_sensor.py
4 # Description:
5 # This programme provides an interface to the PAN1780 BLE
     module.
6 # The programme is used to simulate sensors. Requests to the
```

```
7  # sensors are received and answered via a Uart interface.
8  # The aim of this programme is to simulate latencies in
9  # the acquisition of sensor data.
10
11 import serial
12 import time
13 import threading
14 import sys
15
16 comport = "COM5"
17 baudrate = 115200
18 kill = False
19 counter = 255
20 UART_LOCK = threading.RLock()
21 # This dict is filled with a combination of the moduls,
       sub_moduls
22 # and sensors. They are mapped to a processing time in ms.
23 key_dict = {
24     "0x7a8c3f": 90,
25     "0x7a8c6c": 50,
26     "0x7a6efc": 90,
27     "0x7a6e92": 20,
28     "0x7ac1e0": 5,
29     "0x7ac135": 3,
30     "0x7a5331": 4,
31     "0x7a5321": 68,
32     "0xacdec3": 62,
33     "0xacde8b": 59,
34     "0xac8b58": 9,
35     "0xac8b6c": 27,
36     "0xac968b": 67,
37     "0xac96c1": 17,
38     "0xacbb96": 90,
39     "0xacbbc9": 5,
40     "0x870c0c": 53,
41     "0x870cbb": 6,
```

```
42      "0x87332d": 87,
43      "0x8733bb": 1,
44      "0x875f13": 68,
45      "0x875fc3": 9,
46      "0x87240e": 1,
47      "0x8724ed": 5,
48  }
49  # The read and write KEYs for the asynchronous read or write.
50  ASYN_READ = 0x52
51  ASYN_WRITE = 0x57
52
53  # This function waits for the time that is assigned to the
        sensor
54  def sensorRead(request):
55      global counter
56
57      counter = counter + 1
58      r_value = counter
59      l_request = hex(request)
60
61      # Checks whether the transfered request is part of
62      # the permitted KEYs.
63      if l_request in key_dict:
64          # Gives out the time that has to be waited.
65          print("\n")
66          print(
67              sys._getframe().f_code.co_name + ":\t Sensor_time:
        \t",
68              key_dict.get(l_request),
69          )
70          print("\t\t" + l_request)
71          # Waits the time until the sensor responds.
72          time.sleep(key_dict.get(l_request) / 100)
73      # Or whether the transferred request is not part of
74      # the allowed KEYs.
75      else:
```

```
76          # Than do nothing.
77          return 0
78      return r_value
79
80
81  # Sends the sensors response to the BLE server.
82  def responseSensor(port, operation, key, resp):
83      try:
84          if operation == ASYN_READ:
85              # A read resquest is answered as follows.
86              # |8 Bit Operation | 24 Bit Key | 16 Bit sensor
    data |
87              data = (operation << 40) + (key << 16) + resp
88              # Sends the response to the BLE server
89              port.write(data.to_bytes(6, byteorder="big"))
90          elif operation == ASYN_WRITE:
91              # A write resquest is answered as follows.
92              # |8 Bit Operation | 24 Bit Key | 8 Bit sensor
    data |
93              # The respone data answer will be the fixed value
    129
94              resp = 129
95              data = (operation << 32) + (key << 8) + resp
96              # Sends the response to the BLE server
97              port.write(data.to_bytes(5, byteorder="big"))
98
99          print("\n")
100         print(sys._getframe().f_code.co_name)
101         print("(): \t\t\t\t\tData OUT: ")
102         print(hex(data))
103         # This flushes the port.
104         port.flush()
105     except:
106         print("\n")
107         print(sys._getframe().f_code.co_name)
108         print("(): Error sending")
```

```
109         # This flushes the port
110         port.flush()
111
112
113 # This function separates the message from the BLE server.
114 def req_separate(data):
115     value = 0
116     # The operation defines if its a read or wwrite operation.
117     operation = int.from_bytes(data, byteorder="little") &
    int.from_bytes(
118         bytes.fromhex("FF"), byteorder="little"
119     )
120
121     # The modul, sub_modul and sensor is contained
122     # in the last 3 bytes of the message.
123     modul = int.from_bytes(data, byteorder="little") &
    int.from_bytes(
124         bytes.fromhex("00FF"), byteorder="little"
125     )
126     key = modul << 8
127     submodul = int.from_bytes(data, byteorder="little") &
    int.from_bytes(
128         bytes.fromhex("0000FF"), byteorder="little"
129     )
130     key += submodul >> 8
131     sensor = int.from_bytes(data, byteorder="little") &
    int.from_bytes(
132         bytes.fromhex("000000FF"), byteorder="little"
133     )
134     key += sensor >> 24
135
136     if operation == ASYN_READ:
137         return operation, key, value
138     elif operation == ASYN_WRITE:
139         value = int.from_bytes(data, byteorder="big") &
    int.from_bytes(
```

```
140             bytes.fromhex("FFFF"), byteorder="big"
141         )
142         return operation, key, value
143     return
144
145
146 # This function has the task of processing the requested data
147 def thread_Sensor(port, data):
148     # Gets the operation and the destination for the request.
149     operation, key, value = req_separate(data)
150     if operation == ASYN_READ:
151         print("\n\t OPERATION READ !!!\n")
152         # Gets the sensor response.
153         response = sensorRead(key)
154         with UART_LOCK:
155             responseSensor(port, operation, key, response)
156
157     elif operation == ASYN_WRITE:
158         print("\n\t OPERATION WRITE !!!\n")
159         # Gets the sensor response
160         response = sensorRead(key)
161         with UART_LOCK:
162             responseSensor(port, operation, key, response)
163     return
164
165
166 # This function has the task of checking the port for new
      requests.
167 # Waits for a request on the port
168 def read_Thread(ser):
169     # While loop with the possibility to kill the thread.
170     while not kill:
171         if ser.inWaiting() > 0:
172             b_data = bytes(0)
173             with UART_LOCK:
```

```
174                 while ser.in_waiting:  # Reads the port till
    its empty.
175                     ch = ser.read()
176                     b_data += ch  # Saves the read data
    temporary.
177             # Start a new thread that takes care of the
    request.
178             req_thr = threading.Thread(
179                 target=thread_Sensor,
180                 args=(
181                     ser,
182                     b_data,
183                 ),
184             )
185             req_thr.setDaemon(True)
186             req_thr.start()
187         time.sleep(0.1)
188
189
190 if __name__ == "__main__":
191     # This opens the serial port
192     ser = serial.Serial(comport, baudrate, xonxoff=True)
193     # This creates a thread that waits for requests and
    handles them.
194     wait_thr = threading.Thread(
195         target=read_Thread,
196         args=(ser,),
197     )
198     wait_thr.setDaemon(True)
199     wait_thr.start()  #
200
201     while ser.is_open:
202         value = 0
203         value = input("Press 5 to exit: ")
204         print("\n")
```

```
205         if value == "5":  # The programm will shut down if 5
    is pressed.
206             kill = True  # Activate the kill process.
207             time.sleep(2)  # Wait a time so the thread can end.
208             ser.close()  # Close serial connection.
209             break
```

Listing D.2: Quellprogramm des Systems zum simulieren von Latenzen

D.3 Bluetooth Low Energy Server

Der Anhang zur Arbeit befindet sich auf der CD und kann beim Erstgutachter eingesehen werden.

D.3.1 Main Programm

Der folgende Quellcode enthält nur die Codeabschnitte, die für die Integration in eine Vorlage erforderlich sind. Für die Entwicklung wurde die Vorlage ble_app_template [18] aus dem Software Entwicklungskit nRF5 in der Version 16.0.0 verwendet.

```
1  /**
2   * Copyright (c) 2014 - 2019, Nordic Semiconductor ASA
3   *
4   * All rights reserved.
5   *
6   * Redistribution and use in source and binary forms, with or
      without modification,
7   * are permitted provided that the following conditions are
      met:
8   *
9   * 1. Redistributions of source code must retain the above
      copyright notice, this
10  *    list of conditions and the following disclaimer.
11  *
```

```
12  * 2. Redistributions in binary form, except as embedded into
      a Nordic
13  *    Semiconductor ASA integrated circuit in a product or a
      software update for
14  *    such product, must reproduce the above copyright notice,
      this list of
15  *    conditions and the following disclaimer in the
      documentation and/or other
16  *    materials provided with the distribution.
17  *
18  * 3. Neither the name of Nordic Semiconductor ASA nor the
      names of its
19  *    contributors may be used to endorse or promote products
      derived from this
20  *    software without specific prior written permission.
21  *
22  * 4. This software, with or without modification, must only
      be used with a
23  *    Nordic Semiconductor ASA integrated circuit.
24  *
25  * 5. Any software provided in binary form under this license
      must not be reverse
26  *    engineered, decompiled, modified and/or disassembled.
27  *
28  * THIS SOFTWARE IS PROVIDED BY NORDIC SEMICONDUCTOR ASA "AS
      IS" AND ANY EXPRESS
29  * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
      IMPLIED WARRANTIES
30  * OF MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A
      PARTICULAR PURPOSE ARE
31  * DISCLAIMED. IN NO EVENT SHALL NORDIC SEMICONDUCTOR ASA OR
      CONTRIBUTORS BE
32  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
      EXEMPLARY, OR
33  * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
      PROCUREMENT OF SUBSTITUTE
```

```
34  * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
      BUSINESS INTERRUPTION)
35  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
      CONTRACT, STRICT
36  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
      ARISING IN ANY WAY OUT
37  * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
      POSSIBILITY OF SUCH DAMAGE.
38  *
39  */
40 /** @file
41  *
42  * @defgroup ble_sdk_app_template_main main.c
43  * @{
44  * @ingroup ble_sdk_app_template
45  * @brief Template project main file.
46  *
47  * This file contains a template for creating a new
      application. It has the code necessary to wakeup
48  * from button, advertise, get a connection restart
      advertising on disconnect and if no new
49  * connection created go back to system-off mode.
50  * It can easily be used as a starting point for creating a
      new application, the comments identified
51  * with 'YOUR_JOB' indicates where and how you can customize.
52  */
53 ...
54 #include "ble_asyn_protocol.h"              /* 0) Adding my
      function headers. */
55 ...
56 BLE_FU_DEF(m_pos);                        /* 3) My service
      structure. */
57 ...
58 /**@brief Function for the Timer initialization.
59  *
```

```
60  * @details Initializes the timer module. This creates and
      starts application timers.
61  */
62 static void timers_init(void)
63 {
64     ...
65     // Create timers.
66
67     fu_timer_create();  /* 2) Adding the Timer function. */
68 }
69 ...
70 /**@brief Function for initializing services that will be used
      by the application.
71  */
72 static void services_init(void)
73 {
74     ...
75
76 /* YOUR_JOB: */
77     fu_service_init(&m_pos);          /* 3) Adding the service
      init function. */
78 }
79 ...
80 /**@brief  Function for initializing the UART module.
81  */
82 /**@snippet [UART Initialization] */
83 static void uart_init(void)
84 {
85 ...
86
87     APP_UART_FIFO_INIT(&comm_params,
88                        UART_RX_BUF_SIZE,
89                        UART_TX_BUF_SIZE,
90                        asyn_uart_evt,           /* 1) Adding
      the uart evt handler. */
91                        APP_IRQ_PRIORITY_LOWEST,
```

```
92                          err_code);
93     APP_ERROR_CHECK(err_code);
94 }
95 /**@snippet [UART Initialization] */
96 ...
97 /**@brief Function for application main entry.
98  */
99 int main(void)
100 {
101     bool erase_bonds;
102 
103     // Initialize.
104     uart_init();            /* 1) Adding the UART init stuff.
    */
105     timers_init();          /* 2) Adding the Timer stuff. */
106     buttons_leds_init(&erase_bonds);
107     power_management_init();
108     ble_stack_init();
109     gap_params_init();
110     gatt_init();
111     advertising_init();
112     services_init();        /* 3) Adding the service
    initialization. */
113     conn_params_init();
114     peer_manager_init();
115     // Start execution.
116     NRF_LOG_INFO("Template example started.");
117     application_timers_start();
118 
119     advertising_start(erase_bonds);
120 
121     fu_testsystem_init();         /* 4) Adding the key list,
    if needed. */
122     // Enter main loop.
123     for (;;)
124     {
```

```
125         idle_state_handle();
126         asyn_processing_BLE(); /* 5) Adding the async process.
    */
127     }
128 }
```

Listing D.3: Ausschnitte aus dem Quellprogramm der Vorlage ble_app_template [18] des Software Entwicklungskit nRF5 in der Version 16.0.0 mit den benötigten Änderungen

D.3.2 Ble asynchrone Protokoll

```
1  /*
2  # Autor: M. Pyka
3  # Date: 27.12.20
4  # Name: ble_asyn_protocol.c
5  # Description:
6  # This programme module handles all asynchronous
7  # Bluetooth Low Energy communication. Asynchronous
8  # communication via ble can only be done with a ble
9  # client that has implemented the asynchronous communication
10 # protocol. This programme module is to be considered as
11 # a prove of concept.
12 */
13
14 #include <stdint.h>
15 #include <string.h>
16 #include "ble_srv_common.h"
17 #include "nrf_delay.h"
18 #include "nrf_drv_rng.h"
19 #include "nrf_assert.h"
20 #include "app_error.h"
21 #include "app_timer.h"
22 #include "app_uart.h"
23 #include "nrf_drv_clock.h"
24 #if defined (UART_PRESENT)
```

```
25 #include "nrf_uart.h"
26 #endif
27 #if defined (UARTE_PRESENT)
28 #include "nrf_uarte.h"
29 #endif
30
31 #include "ble_asyn_protocol.h"
32 /****** DEBUGG ******/
33 #define DEBUG
34 #ifdef DEBUG
35 #define dprintf(...) \
36 do { \
37 printf("%s(): ", __func__); \
38 printf(__VA_ARGS__); \
39 } \
40 while (0)
41 #else
42 #define dprintf(...)
43 #endif
44
45 /****** TIMER ******/
46 APP_TIMER_DEF(my_timer_id1);
47 APP_TIMER_DEF(my_timer_id2);
48 APP_TIMER_DEF(my_timer_id3);
49 APP_TIMER_DEF(my_timer_id4);
50 APP_TIMER_DEF(my_timer_id5);
51 APP_TIMER_DEF(my_timer_id6);
52 APP_TIMER_DEF(my_timer_id7);
53 APP_TIMER_DEF(my_timer_id8);
54 APP_TIMER_DEF(my_timer_id9);
55 APP_TIMER_DEF(my_timer_id10);
56 const app_timer_t * my_timers[10] = {my_timer_id1,
    my_timer_id2, my_timer_id3, my_timer_id4, my_timer_id5,
    my_timer_id6, my_timer_id7, my_timer_id8, my_timer_id9,
    my_timer_id10};
57 /****** QUEUE ******/
```

```
58 /* ble_raw_input */
59 NRF_QUEUE_DEF(ble_request_raw_data_t, ble_raw_input_queue,
       QUEUE_BLE_MAX, NRF_QUEUE_MODE_NO_OVERFLOW);
60 /* uart_input */
61 NRF_QUEUE_DEF(uart_response_raw_data_t, uart_raw_input_queue,
       QUEUE_UART_MAX, NRF_QUEUE_MODE_NO_OVERFLOW);
62 /* prcessing */
63 NRF_QUEUE_DEF(processing_data_t, processing_queue,
       QUEUE_PROCESS_MAX,NRF_QUEUE_MODE_NO_OVERFLOW);
64 /* timer */
65 NRF_QUEUE_DEF(void *, free_timer_queue,
       QUEUE_PROCESS_MAX,NRF_QUEUE_MODE_NO_OVERFLOW);
66 NRF_QUEUE_DEF(void *, fired_timer_queue,
       QUEUE_PROCESS_MAX,NRF_QUEUE_MODE_NO_OVERFLOW);
67
68 /****** GLOBAL_VAR ******/
69 g_ble_request_data_t * g_uart_conn_handle = NULL;
70
71
72 /*************************************
73 *
74 *                  FU_INIT
75 *
76 *************************************/
77 /**@brief Function for adding a new characterstic to
       "fu_service".
78  *
79  * @param[in]   g_ble_request_data_t        p_session structure
80  *
81  */
82 static void fu_char_add(g_ble_request_data_t* p_session)
83 {
84   uint32_t            err_code;
85   ble_uuid_t          char_uuid;
86   ble_uuid128_t       base_uuid = BLE_UUID_FUNC_BASE_UUID;
87   char_uuid.uuid      = BLE_UUID_FUNC_CHARACTERISTC_UUID;
```

```
88  err_code = sd_ble_uuid_vs_add(&base_uuid, &char_uuid.type);
89  APP_ERROR_CHECK(err_code);
90
91  ble_gatts_attr_md_t attr_md;
92  memset(&attr_md, 0, sizeof(attr_md));
93  attr_md.vloc        = BLE_GATTS_VLOC_STACK;
94  attr_md.vlen        = 1;
95  attr_md.wr_auth     = 1;
96
97  ble_gatts_attr_t    attr_char_value;
98  memset(&attr_char_value, 0, sizeof(attr_char_value));
99  attr_char_value.p_uuid      = &char_uuid;
100 attr_char_value.p_attr_md   = &attr_md;
101 attr_char_value.max_len     = 20;
102 attr_char_value.init_len    = 20;
103
104 ble_gatts_char_md_t char_md;
105 memset(&char_md, 0, sizeof(char_md));
106 char_md.char_props.read = 0;
107 char_md.char_props.write = 1;
108
109 BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);
110 BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.write_perm);
111
112 ble_gatts_attr_md_t cccd_md;
113 memset(&cccd_md, 0, sizeof(cccd_md));
114 BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.read_perm);
115 BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.write_perm);
116 cccd_md.vloc                = BLE_GATTS_VLOC_STACK;
117 cccd_md.rd_auth = 1;
118
119 char_md.p_cccd_md           = &cccd_md;
120 char_md.char_props.indicate = 1;
121 err_code =
     sd_ble_gatts_characteristic_add(p_session->service_handle,
122                                 &char_md,
```

```
123                                 &attr_char_value,
124                                 &p_session->char_handles);
125   APP_ERROR_CHECK(err_code);
126
127   err_code = nrf_drv_rng_init(NULL);
128   APP_ERROR_CHECK(err_code);
129 }
130
131 /**@brief Function for initiating the new fu_service.
132  *
133  * @param[in]   g_ble_request_data_t        p_session
      structure.
134  */
135 void fu_service_init(g_ble_request_data_t* p_session)
136 {
137   // STEP 3: Declare 16 bit service and 128 bit base UUIDs and
       add them to BLE stack table
138   uint32_t err_code;
139   ble_uuid_t    service_uuid;
140   ble_uuid128_t       base_uuid = BLE_UUID_FUNC_BASE_UUID;
141   service_uuid.uuid = BLE_UUID_FUNC_SERV;
142   err_code = sd_ble_uuid_vs_add(&base_uuid,
       &service_uuid.type);
143   APP_ERROR_CHECK(err_code);
144   // STEP 4: Add our service
145   err_code =
       sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY,
146                                 &service_uuid,
147                                 &p_session->service_handle);
148   APP_ERROR_CHECK(err_code);
149   // Initialisation of the characteristics
150   fu_char_add(p_session);
151 }
152 void fu_timer_create(void)
153 {
154   uint32_t err_code;
```

```
155  /*** Creat Timer ***/
156    err_code = app_timer_create(&my_timer_id1,
157                                APP_TIMER_MODE_SINGLE_SHOT,
158                                asyn_timeout_evt);
159    APP_ERROR_CHECK(err_code);
160    /** Add timer to free timer queue **/
161    err_code = nrf_queue_push(&free_timer_queue,
       &my_timer_id1);
162    err_code = app_timer_create(&my_timer_id2,
163                                APP_TIMER_MODE_SINGLE_SHOT,
164                                asyn_timeout_evt);
165    err_code = nrf_queue_push(&free_timer_queue,
       &my_timer_id2);
166    err_code = app_timer_create(&my_timer_id3,
167                                APP_TIMER_MODE_SINGLE_SHOT,
168                                asyn_timeout_evt);
169    err_code = nrf_queue_push(&free_timer_queue,
       &my_timer_id3);
170    err_code = app_timer_create(&my_timer_id4,
171                                APP_TIMER_MODE_SINGLE_SHOT,
172                                asyn_timeout_evt);
173    err_code = nrf_queue_push(&free_timer_queue,
       &my_timer_id4);
174    err_code = app_timer_create(&my_timer_id5,
175                                APP_TIMER_MODE_SINGLE_SHOT,
176                                asyn_timeout_evt);
177    err_code = nrf_queue_push(&free_timer_queue,
       &my_timer_id5);
178    err_code = app_timer_create(&my_timer_id6,
179                                APP_TIMER_MODE_SINGLE_SHOT,
180                                asyn_timeout_evt);
181    err_code = nrf_queue_push(&free_timer_queue,
       &my_timer_id6);
182    err_code = app_timer_create(&my_timer_id7,
183                                APP_TIMER_MODE_SINGLE_SHOT,
184                                asyn_timeout_evt);
```

```
185     err_code = nrf_queue_push(&free_timer_queue,
    &my_timer_id7);
186     err_code = app_timer_create(&my_timer_id8,
187                                 APP_TIMER_MODE_SINGLE_SHOT,
188                                 asyn_timeout_evt);
189     err_code = nrf_queue_push(&free_timer_queue,
    &my_timer_id8);
190     err_code = app_timer_create(&my_timer_id9,
191                                 APP_TIMER_MODE_SINGLE_SHOT,
192                                 asyn_timeout_evt);
193     err_code = nrf_queue_push(&free_timer_queue,
    &my_timer_id9);
194     err_code = app_timer_create(&my_timer_id10,
195                                 APP_TIMER_MODE_SINGLE_SHOT,
196                                 asyn_timeout_evt);
197     err_code = nrf_queue_push(&free_timer_queue,
    &my_timer_id10);
198     APP_ERROR_CHECK(err_code);
199 }
200 void fu_testsystem_init(void)
201 {
202   keylist_fill();
203 }
204
205 /*************************************
206 *
207 *               TASK
208 *
209 *************************************/
210
211 /***BLE*****************************/
212 static void task_ble(void)
213 {
214   dprintf(" IN\n");
215   processing_data_t process_data;
216   ble_request_raw_data_t queue_raw_data;
```

```
217   uint32_t error_code;
218   uint8_t async_err;
219   static uint32_t timeout_ms = 3000;    // max Timeout time in
       ms
220   void * p_timer_id;
221
222 /*** Queue pop ***/
223   nrf_queue_generic_pop(&ble_raw_input_queue, &queue_raw_data,
       false);  /*** getting first element. ***/
224
225 /*** Check ***/
226   async_err = req_check(&queue_raw_data, &process_data);
227
228 /*** Send: Uart, Queue | Error ***/
229   if(async_err != CHECK_SUCCESS)
230   {
231     ble_indication_data_t indication_data;
232     indication_data.op = process_data.request.op;
233
234     indication_data.key.modul = process_data.request.key.modul;
235     indication_data.key.submodul =
      process_data.request.key.submodul;
236     indication_data.key.sensor =
      process_data.request.key.sensor;
237     indication_data.response.status = async_err;
238     UNUSED_VARIABLE(send_indication_status(&indication_data));
239     return;
240   }
241
242   /** send request **/
243   error_code = send_uart(&process_data);
244   APP_ERROR_CHECK(error_code);
245   /** Getting one free timer **/
246   error_code = nrf_queue_generic_pop(&free_timer_queue,
       &p_timer_id, false);  /*** getting first element. ***/
247   APP_ERROR_CHECK(error_code);
```

```
248   /** Save request to processing_queue **/
249   process_data.p_timer_id = p_timer_id;
250   error_code = nrf_queue_push(&processing_queue,
       &process_data);
251   APP_ERROR_CHECK(error_code);
252   /** Increase keylist counter **/
253   increase_keylist_counter(process_data.request.key.modul);
254   /*** Start AppTimer ***/
255   dprintf("%p\n", p_timer_id);
256   error_code = app_timer_start(p_timer_id,
       APP_TIMER_TICKS(timeout_ms), p_timer_id);
257   APP_ERROR_CHECK(error_code);
258 }
259 /***UART*****************************/
260 static void task_uart(void)
261 {
262    dprintf(" IN\n");
263    processing_data_t processing_data;
264    ble_indication_data_t indication_data;
265    uart_response_raw_data_t uart_raw_data;
266    size_t num_queue;
267
268    /*** Getting first element. ***/
269    nrf_queue_generic_pop(&uart_raw_input_queue,
       &uart_raw_data, false);
270    indication_data.op = uart_raw_data.buf[0];
271
272    indication_data.key.modul = uart_raw_data.buf[1];
273    indication_data.key.submodul = uart_raw_data.buf[2];
274    indication_data.key.sensor = uart_raw_data.buf[3];
275    if(indication_data.op == BLE_ASYNC_READ)
276    {
277       indication_data.response.value[0] = uart_raw_data.buf[4];
278       indication_data.response.value[1] = uart_raw_data.buf[5];
279    }
280    else if (indication_data.op == BLE_ASYNC_WRITE)
```

```
281     {
282        indication_data.response.status = IND_CORRECT;
283     }
284     /*** Match to processing queue ***/
285     num_queue = nrf_queue_utilization_get(&processing_queue);
286     for(int i = 0; i < num_queue; i++)
287     {
288        nrf_queue_generic_pop(&processing_queue,
        &processing_data, false);  /*** getting first element. ***/
289        if (processing_data.request.op == indication_data.op)
290        {
291          if((processing_data.request.key.modul ==
        indication_data.key.modul) &&
        (processing_data.request.key.submodul ==
        indication_data.key.submodul) &&
        (processing_data.request.key.sensor ==
        indication_data.key.sensor))
292          {
293     /*** Send Indication ***/
294            /** stop Timer */
295            app_timer_stop(processing_data.p_timer_id);
296            /** adding timer to free timer queue **/
297            nrf_queue_push(&free_timer_queue,
        &processing_data.p_timer_id);
298            dprintf("%p\n", processing_data.p_timer_id);
299            /** send Indication */
300            dprintf("Sending indication\n");
301            if(indication_data.op == BLE_ASYNC_READ)
302            {
303
        UNUSED_VARIABLE(send_indication_value(&indication_data));
304            }
305            else if (indication_data.op == BLE_ASYNC_WRITE)
306            {
307
        UNUSED_VARIABLE(send_indication_status(&indication_data));
```

```
308             }
309         #ifdef WITH_KEY_CHECK
310         /* decrease the keylist counter */
311             decrease_keylist_counter(indication_data.key.modul);
312         #endif
313             break;
314           }
315           else
316           {
317             nrf_queue_push(&processing_queue, &processing_data);
                        /* Add element back to Queue */
318           }
319         }
320         else
321         {
322           nrf_queue_push(&processing_queue, &processing_data);
                      /* Add element back to Queue */
323         }
324     }
325 }
326 /***TIMEOUT*************************/
327 static void task_timeout(void)
328 {
329     void * p_timer;
330     size_t num_queue;
331     processing_data_t processing_data;
332     ble_indication_data_t indication_data;
333 
334     while (nrf_queue_utilization_get(&fired_timer_queue) > 0)
335     {
336       dprintf(" IN\n");
337       nrf_queue_generic_pop(&fired_timer_queue, &p_timer,
    false);  /*** getting first element. ***/
338   /*** Match to processing queue ***/
339       num_queue = nrf_queue_utilization_get(&processing_queue);
340       for(int i = 0; i < num_queue; i++)
```

```
341     {
342         nrf_queue_generic_pop(&processing_queue,
    &processing_data, false);  /*** getting first element. ***/
343         if (processing_data.p_timer_id == p_timer)
344         {
345       /*** Send Indication ***/
346           dprintf("%d\n",
    nrf_queue_utilization_get(&free_timer_queue));
347           /** adding timer to free timer queue **/
348           nrf_queue_push(&free_timer_queue,
    &processing_data.p_timer_id);
349           dprintf("%p\n",processing_data.p_timer_id);
350           /** Fill indication with informations **/
351           indication_data.op = processing_data.request.op;
352
353           indication_data.key.modul =
    processing_data.request.key.modul;
354           indication_data.key.submodul =
    processing_data.request.key.submodul;
355           indication_data.key.sensor =
    processing_data.request.key.sensor;
356           indication_data.response.status = IND_TIMEOUT;
357           /** send Indication */
358           dprintf("Sending TIME OUT\n");
359
    UNUSED_VARIABLE(send_indication_status(&indication_data));
360           #ifdef WITH_KEY_CHECK
361           /* decrease the keylist counter */
362           decrease_keylist_counter(indication_data.key.modul);
363           #endif
364           break;
365         }
366         else
367         {
368           nrf_queue_push(&processing_queue, &processing_data);
                    /* Add element back to Queue */
```

```
369         }
370       }
371     }
372 }
373
374
375 /*************************************
376 *
377 *                  EVENT
378 *
379 *************************************/
380 /***BLE*********************************/
381 void asyn_ble_evt(ble_evt_t const * p_ble_evt, void *
      p_context)
382 {
383   /*store context of service handle , characteristic handle
       and write handler for event*/
384   g_ble_request_data_t * p_session = (g_ble_request_data_t
       *)p_context;
385
386   switch (p_ble_evt->header.evt_id)
387   {
388     case BLE_GAP_EVT_CONNECTED:
389         dprintf("BLE_GAP_EVT_CONNECTED\n");
390         p_session->conn_handle =
      p_ble_evt->evt.gap_evt.conn_handle;
391         g_uart_conn_handle = p_session;
392         break;
393     case BLE_GAP_EVT_DISCONNECTED:
394         p_session->conn_handle = BLE_CONN_HANDLE_INVALID;
395         g_uart_conn_handle = NULL;
396         break;
397     case BLE_GATTS_EVT_RW_AUTHORIZE_REQUEST:
398       handle_ble_write_evt(p_ble_evt);
399       break;// BLE_GATTS_EVT_RW_AUTHORIZE_REQUEST
400     default:
```

```
401         // No implementation needed.
402         break;
403   }
404 }
405 #define UART_MAX_DATA_LEN 6
406
407 /***UART*********************************/
408 /**@brief   Function for handling app_uart events.
409  *
410  * @details This function will receive bytes from the databus
      and append it to
411  *          the data form. The data will than be sent over BLE
      to the BLE peripheral.
412  */
413 /**@snippet [Handling the data received over UART] */
414
415 void asyn_uart_evt(app_uart_evt_t * p_event)
416 {
417     static uint8_t data_array[UART_MAX_DATA_LEN];
418     static uint8_t index = 0;
419     static uint8_t op = 0;
420     static uint8_t end = UART_MAX_DATA_LEN;
421     uint32_t       err_code;
422     uart_response_raw_data_t l_data;
423
424     switch (p_event->evt_type)
425     {
426         case APP_UART_DATA_READY:
427         /***  Data collect ***/
428             UNUSED_VARIABLE(app_uart_get(&data_array[index]));
429             if(index == 0) /** First data indicates Read or
      Write **/
430             {
431               op = data_array[0];
432               /** A Read operation returns a value **/
433               if (op == BLE_ASYNC_READ)
```

```
434                 {
435                   end = LENG_INDICATE_VALUE;
436                 /** A Write operation returns a status **/
437                 }else if (op == BLE_ASYNC_WRITE)
438                 {
439                   end = LENG_INDICATE_STATUS;
440                 }
441               }
442               index ++;
443               index %= end;
444
445               if(index == 0)
446               {
447               /** Data complete **/
448                   l_data.len = end;
449                   for(int i = 0; i < end; i++)
450                   {
451                      l_data.buf[i] = data_array[i];
452                   }
453         /***  Add Data to queue ***/
454                   nrf_queue_push(&uart_raw_input_queue,
    &l_data);                  /* Add element to Queue */
455               }
456               break;
457         case APP_UART_COMMUNICATION_ERROR:
458
    APP_ERROR_HANDLER(p_event->data.error_communication);
459               break;
460         case APP_UART_FIFO_ERROR:
461               APP_ERROR_HANDLER(p_event->data.error_code);
462               break;
463         default:
464               break;
465     }
466 }
467 /***TIMEOUT*************************/
```

```
468 void asyn_timeout_evt(void * p_fired_timer)
469 {
470   dprintf(" IN\n");
471   /** Add fired Timer to fired timer queue **/
472   nrf_queue_push(&fired_timer_queue, &p_fired_timer);
473 }
474
475
476 /**************************************
477 *
478 *                                    HELPING_FUNCTIONS
479 *
480 **************************************/
481 static void handle_ble_write_evt(ble_evt_t const * p_ble_evt)
482 {
483   const ble_gatts_evt_rw_authorize_request_t *req;
484   ble_gatts_rw_authorize_reply_params_t auth_reply;
485   memset(&auth_reply, 0, sizeof(auth_reply));
486   ble_request_raw_data_t raw_data;
487   uint32_t err_code;
488
489   req = &p_ble_evt->evt.gatts_evt.params.authorize_request;
490   if (req->type == BLE_GATTS_AUTHORIZE_TYPE_WRITE)
491   {
492     switch(incoming_inspection(&req->request.write))
493     {
494       case INS_CORRECT:
495           raw_data.len = req->request.write.len;
496           memcpy(&raw_data.buf[0],
    &req->request.write.data[0], req->request.write.len);
497
498           err_code = nrf_queue_push(&ble_raw_input_queue,
    &raw_data);
499           APP_ERROR_CHECK(err_code);
500           auth_reply.params.write.gatt_status =
    BLE_GATT_STATUS_SUCCESS;          /* Success */
```

```
501             break;
502         case INS_QUEUE_FULL:
503             auth_reply.params.write.gatt_status =
    BLE_GATT_STATUS_ATTERR_CPS_PROC_ALR_IN_PROG; /* Queue is
    full */
504             break;
505         case INS_LEN_N_CORRECT:
506             auth_reply.params.write.gatt_status =
    BLE_GATT_STATUS_ATTERR_INVALID_ATT_VAL_LENGTH; /* length
    not correct */
507             break;
508         default:
509             break;
510       }
511       auth_reply.type = BLE_GATTS_AUTHORIZE_TYPE_WRITE;
512       auth_reply.params.write.update = 1;
513       auth_reply.params.write.offset = 0;
514       auth_reply.params.write.len = req->request.write.len;
515       auth_reply.params.write.p_data = req->request.write.data;
516     }
517     else
518     {
519       auth_reply.type = BLE_GATTS_AUTHORIZE_TYPE_READ;
520       auth_reply.params.read.gatt_status =
      BLE_GATT_STATUS_ATTERR_APP_BEGIN;
521     }
522     err_code =
      sd_ble_gatts_rw_authorize_reply(p_ble_evt->evt.gatts_evt.conn_handle,
523                                                 &auth_reply);
524     APP_ERROR_CHECK(err_code);
525 }
526
527 static uint8_t incoming_inspection(ble_gatts_evt_write_t
      const* p_evt_write)
528 {
529     uint16_t len = (p_evt_write->len);
```

```
530
531   /* first of all lenght of request is checked */
532   if ((len != LENG_WRITE_REQ) && (len != LENG_READ_REQ))
533   {
534     dprintf("INS_LEN_N_CORRECT\n");
535     return INS_LEN_N_CORRECT;
536   }/* than Queues are checked */
537   else if(nrf_queue_is_full(&ble_raw_input_queue) ||
       nrf_queue_is_full(&processing_queue))
538   {
539     dprintf("INS_QUEUE_FULL\n");
540     return INS_QUEUE_FULL;
541   }
542   else
543   {
544     return INS_CORRECT;
545   }
546 }
547
548 static uint8_t req_check(ble_request_raw_data_t *raw_data,
      processing_data_t *process)
549 {
550   uint8_t async_err = CHECK_OP_ERROR;
551
552   process->request.op = raw_data->buf[0];
553   process->request.key.modul = raw_data->buf[1];
554   process->request.key.submodul = raw_data->buf[2];
555   process->request.key.sensor = raw_data->buf[3];
556   switch(raw_data->buf[0])
557   {
558     case BLE_ASYNC_WRITE:
559       if(raw_data->len != LENG_WRITE_REQ)
560       {
561         return async_err;
562         break;
563       }
```

```
564       process->request.value =
      (TYPE_REQ_VALUE)(raw_data->buf[4] << 8);
565       process->request.value |= (TYPE_REQ_VALUE)
      raw_data->buf[5];
566       async_err = CHECK_SUCCESS;
567       break;
568     case BLE_ASYNC_READ:
569       if(raw_data->len != LENG_READ_REQ)
570       {
571         return async_err;
572         break;
573       }
574       async_err = CHECK_SUCCESS;
575       break;
576     default:
577       dprintf("Operation ERROR\n");
578       return async_err;
579       break;
580   }
581   /*** Key ***/
582 #ifdef WITH_KEY_CHECK
583       async_err = keylist_check(&process->request.key);
584 #endif
585   return async_err;
586 }
587
588 static uint32_t send_indication_status(ble_indication_data_t
      *indication_data)
589 {
590   uint32_t error_code = NRF_SUCCESS;
591   TYPE_IND_STATUS l_data[LENG_INDICATE_STATUS];
592   uint16_t len = LENG_INDICATE_STATUS;              // len = 5
593   ble_gatts_hvx_params_t p_hvx_params;
594   if(g_uart_conn_handle->conn_handle !=
      BLE_CONN_HANDLE_INVALID)
595   {
```

```
596     memset(&p_hvx_params, 0, sizeof(p_hvx_params));
597
598     l_data[0] = indication_data->op;
599     l_data[1] = indication_data->key.modul;
600     l_data[2] = indication_data->key.submodul;
601     l_data[3] = indication_data->key.sensor;
602     l_data[4] = indication_data->response.status;
603
604     p_hvx_params.type     = BLE_GATT_HVX_INDICATION;
605     p_hvx_params.handle   =
      g_uart_conn_handle->char_handles.value_handle;
606     p_hvx_params.p_data   = l_data;
607     p_hvx_params.p_len    = &len;
608
609     do{
610         error_code =
      sd_ble_gatts_hvx(g_uart_conn_handle->conn_handle,
      &p_hvx_params);
611     }while(error_code == NRF_ERROR_BUSY);
612     if(error_code != NRF_SUCCESS)
613     {
614       dprintf("Error, sending Indication!\n");
615     }
616   }
617   return error_code;
618 }
619
620 static uint32_t send_indication_value(ble_indication_data_t
      *indication_data)
621 {
622   uint32_t error_code = NRF_SUCCESS;
623   TYPE_IND_STATUS l_data[LENG_INDICATE_VALUE];
624   uint16_t len = LENG_INDICATE_VALUE;           // len = 6
625   ble_gatts_hvx_params_t p_hvx_params;
626   if(g_uart_conn_handle->conn_handle !=
      BLE_CONN_HANDLE_INVALID)
```

```
627   {
628     memset(&p_hvx_params, 0, sizeof(p_hvx_params));
629 
630     l_data[0] = indication_data->op;
631     l_data[1] = indication_data->key.modul;
632     l_data[2] = indication_data->key.submodul;
633     l_data[3] = indication_data->key.sensor;
634     l_data[4] = indication_data->response.value[0];
635     l_data[5] = indication_data->response.value[1];
636 
637     p_hvx_params.type     = BLE_GATT_HVX_INDICATION;
638     p_hvx_params.offset   = 0;
639     p_hvx_params.handle   =
     g_uart_conn_handle->char_handles.value_handle;
640     p_hvx_params.p_data   = l_data;
641     p_hvx_params.p_len    = &len;
642 
643     do{
644         error_code =
     sd_ble_gatts_hvx(g_uart_conn_handle->conn_handle,
     &p_hvx_params);
645     }while(error_code == NRF_ERROR_BUSY);
646 
647     if(error_code != NRF_SUCCESS)
648     {
649       dprintf("Error, sending Indication!\n");
650     }
651   }
652   return error_code;
653 }
654 
655 static uint32_t send_uart(processing_data_t *process)
656 {
657   uint32_t error_code;
658 
659   error_code = app_uart_put((uint8_t) process->request.op);
```

```
660   if(error_code != NRF_SUCCESS){return error_code;}
661   error_code = app_uart_put((uint8_t)
       process->request.key.modul);
662   if(error_code != NRF_SUCCESS){return error_code;}
663   error_code = app_uart_put((uint8_t)
       process->request.key.submodul);
664   if(error_code != NRF_SUCCESS){return error_code;}
665   error_code = app_uart_put((uint8_t)
       process->request.key.sensor);
666   if(error_code != NRF_SUCCESS){return error_code;}
667
668   if(process->request.op == BLE_ASYNC_WRITE)
669   {
670    error_code = app_uart_put((uint8_t) (process->request.value
       >>8));
671    if(error_code != NRF_SUCCESS){return error_code;}
672    error_code = app_uart_put((uint8_t) process->request.value);
673    if(error_code != NRF_SUCCESS){return error_code;}
674   }
675   return NRF_SUCCESS;
676 }
677
678
679
680 /*************************************
681 *
682 *       ASYNC_MAIN
683 *
684 *************************************/
685 void asyn_processing_BLE(void)
686 {
687   if(g_uart_conn_handle == NULL) /* if no connection, return.
       */
688   {
689     return;
690   }
```

```
691   volatile size_t num_queue_uart = 0;
692   volatile size_t num_queue_ble = 0;
693   while(1)
694   {
695     num_queue_uart =
      nrf_queue_utilization_get(&uart_raw_input_queue);
696     num_queue_ble  =
      nrf_queue_utilization_get(&ble_raw_input_queue);
697     task_timeout();
698     if((num_queue_uart >= QUEUE_UART_MAX/2) && (num_queue_ble
      <= QUEUE_BLE_MAX/2))
699     {
700       task_uart();
701     }
702     else if((num_queue_ble >= QUEUE_BLE_MAX/2) &&
      (num_queue_uart <= QUEUE_UART_MAX/2))
703     {
704       task_ble();
705     }
706     else if(num_queue_uart > 0)
707     {
708       task_uart();
709     }
710     else if(num_queue_ble > 0)
711     {
712       task_ble();
713     }
714     else
715     {
716       return;
717     }
718   }
719 }
```

Listing D.4: Quellcode des asynchronen Kommunikationsprotokolls für den Bluetooth Low Energy Server

```
1  /*
2  # Autor: M. Pyka
3  # Date: 27.12.20
4  # Name: ble_asyn_protocol.h
5  */
6  #ifndef BLE_ASYN_PROTOCOL_H__
7  #define BLE_ASYN_PROTOCOL_H__
8
9  #include <stdint.h>
10 #include "ble.h"
11 #include "ble_srv_common.h"
12 #include "sdk_errors.h"
13 #include "app_uart.h"
14 #include "nrf_queue.h"
15 #if defined (UART_PRESENT)
16 #include "nrf_uart.h"
17 #endif
18 #if defined (UARTE_PRESENT)
19 #include "nrf_uarte.h"
20 #endif
21 #include "test_system.h"
22
23 #define BLE_UUID_FUNC_BASE_UUID              {0x07, 0x75,
       0x73, 0x58, 0x96, 0x93, 0x59, 0x95, 0x65, 0x41, 0x7d, 0x09,
       0x02, 0xa5, 0xe6, 0xc3} // 128-bit base UUID
24 #define BLE_UUID_FUNC_SERV                   0xAF0D // Just a
       random, but recognizable value
25 #define BLE_UUID_FUNC_CHARACTERISTC_UUID     0x0DAD
26
27 #define BLE_FU_DEF(_name)
                                                \
28 static g_ble_request_data_t _name;
                                                \
29 NRF_SDH_BLE_OBSERVER(_name ## _obs,
                                                \
```

```
30                      1,
                                        \
31                      asyn_ble_evt, &_name)
32 /* ---------------------------------- DEFINES
     ------------------------------------------------------ */
33 /* Types */
34   /***** @ref BLE_ASYNC_TYPES *****/
35   #define BLE_ASYNC_READ  0x52
36   #define BLE_ASYNC_WRITE 0x57
37   /***** Request *****/
38 #define TYPE_REQ_OPERATION  uint8_t
39 #define TYPE_REQ_KEY    SYS_KEY_t
40 #define TYPE_REQ_VALUE    uint16_t
41   /***** Uart *****/
42 #define TYPE_UART_KEY   SYS_KEY_t
43 #define TYPE_UART_RESPONSE  uint16_t
44   /***** Indication *****/
45 #define TYPE_IND_OPERATION  uint8_t
46 #define TYPE_IND_KEY    SYS_KEY_t
47 #define TYPE_IND_VALUE    uint8_t
48 #define TYPE_IND_STATUS   uint8_t
49 /* Lenght */
50   /***** Request *****/
51 #define LENG_READ_REQ     (sizeof(TYPE_REQ_OPERATION) +
     sizeof(TYPE_REQ_KEY))
52 #define LENG_WRITE_REQ    (sizeof(TYPE_REQ_OPERATION) +
     sizeof(TYPE_REQ_KEY) + sizeof(TYPE_REQ_VALUE))
53   /***** UART *****/
54 #define LENG_UART_RESP    (sizeof(TYPE_UART_KEY) +
     sizeof(TYPE_UART_RESPONSE))
55   /***** Indication *****/
56 #define LENG_INDICATE_VALUE   (sizeof(TYPE_IND_OPERATION) +
     sizeof(TYPE_IND_KEY) + sizeof(TYPE_IND_VALUE) +
     sizeof(TYPE_IND_VALUE))
57 #define LENG_INDICATE_STATUS  (sizeof(TYPE_IND_OPERATION) +
     sizeof(TYPE_IND_KEY) + sizeof(TYPE_IND_STATUS))
```

```
58          /*** Queue ***/
59 #define QUEUE_BLE_MAX 10
60 #define QUEUE_UART_MAX 10
61 #define QUEUE_PROCESS_MAX 10
62 /* Errors */
63   /***** Incoming Inspection *****/
64 #define INS_CORRECT 0x01
65 #define INS_QUEUE_FULL 0x02
66 #define INS_LEN_N_CORRECT 0x03
67          /***** Indication Status *****/
68 #define IND_CORRECT 0x00
69 #define IND_TIMEOUT 0xF3
70
71 /* ------------------------ STRUCTS
       ------------------------------------ */
72 typedef struct
73 {
74   uint16_t               len;
75   uint8_t                buf[LENG_WRITE_REQ+1];
76 } ble_request_raw_data_t;    /**< BLE_WRITE_EVT -->
       ble_raw_input_queue */
77
78 typedef struct
79 {
80   uint16_t              len;
81   uint8_t               buf[LENG_UART_RESP];
82 } uart_response_raw_data_t;    /**< APP_UART_DATA_READY -->
       uart_input_queue */
83
84 typedef struct
85 {
86   TYPE_REQ_OPERATION    op; /**< OPERATION */
87   TYPE_REQ_KEY    key;  /**< KEY */
88   TYPE_REQ_VALUE  value;  /**< VALUE */
89 } income_request_data_t;
90
```

```
91 typedef struct
92 {
93   //uint8_t     type;             /**< Type of operation, see
      @ref BLE_ASYNC_TYPES. */
94   income_request_data_t          request;          /**< Request
      Parameters. */
95   void *                         p_timer_id;      /**< Variable
      for app_timer */
96 } processing_data_t; /**< task_ble --> processing_queue */
97
98 typedef struct
99 {
100   TYPE_IND_OPERATION    op;    /**< OPERATION */
101   TYPE_IND_KEY      key;   /**< KEY */
102   union {
103     TYPE_IND_VALUE    value[2]; /**< VALUE */
104     TYPE_IND_STATUS     status; /**< STATUS */
105   } response;
106 } ble_indication_data_t;
107
108
109
110 typedef struct
111 {
112   uint16_t                    conn_handle;
113   uint16_t                    service_handle;     /**< Handle
      of POWER Service (as provided by the BLE stack). */
114   ble_gatts_char_handles_t    char_handles;
115 } g_ble_request_data_t;
116
117 /*** INIT ***/
118 static void fu_char_add(g_ble_request_data_t *
     p_power_service);
119 void fu_service_init(g_ble_request_data_t * p_power_service);
120 void fu_timer_create(void);
121 void fu_testsystem_init(void);
```

```
122
123 /*** TASK ***/
124 static void task_ble(void);
125 static void task_uart(void);
126 static void task_timeout(void);
127
128 /*** EVT ***/
129 void asyn_uart_evt(app_uart_evt_t * p_event);
130 void asyn_ble_evt(ble_evt_t const * p_ble_evt, void *
      p_context);
131 void asyn_timeout_evt(void * p_fired_timer);
132
133 /*** HELPING FUNCTIONS ***/
134 static void handle_ble_write_evt(ble_evt_t const * p_ble_evt);
135 static uint8_t incoming_inspection(ble_gatts_evt_write_t
      const* p_evt_write);
136 static uint8_t req_check(ble_request_raw_data_t *raw_data,
      processing_data_t *process);
137 static uint32_t send_indication_status(ble_indication_data_t
      *indication_data);
138 static uint32_t send_indication_value(ble_indication_data_t
      *indication_data);
139 static uint32_t send_uart(processing_data_t *process);
140 /*** LOOP ***/
141 void asyn_processing_BLE(void);
142
143 #endif  /* _BLE_ASYN_PROTOCOL_H__ */
```

Listing D.5: Header des asynchronen Kommunikationsprotokolls für den Bluetooth Low Energy Server

D.3.3 Testsystem

```
1 /*
2 # Autor: M. Pyka
3 # Date: 27.12.20
4 # Name: test_system.h
```

```
5 */
6
7 #ifndef TEST_SYSTEM_H__
8 #define TEST_SYSTEM_H__
9
10 #include <stdint.h>
11 #include "SEGGER_RTT.h"
12
13 /* Configuration */
14 //#define WITH_KEY_CHECK    /* Key_check checks the existence
      of the key. */
15
16 /***** Detailed Inspection *****/
17 #define CHECK_SUCCESS          0x00
18 #define CHECK_OP_ERROR         0x70
19 #define CHECK_KEY_ERROR        0x71
20 #define CHECK_KEY_ERROR_MODUL 0x72
21
22 /* Checklist Parameter */
23 #define MAX_MODULS 3  /* maximal number of moduls */
24 #define MAX_SUB_MODULS 4  /* maximal number of submoduls */
25 #define MAX_SENSORS 2  /* maximal number of sensors */
26
27 /****** Data Structure Checklist ******/
28 /* Modul */
29 #define TYPE_MODUL       uint8_t /* datatype of modul */
30 /* Sup-modul */
31 #define TYPE_SUP_MODUL   uint8_t /* datatype of sup-modul */
32 /* Sensor */
33 #define TYPE_SENSOR      uint8_t /* datatype of sensor */
34 typedef struct
35 {
36   TYPE_SENSOR     sensors[MAX_SENSORS];
37   TYPE_SUP_MODUL  data;
38 } submodul_t;
39 typedef struct
```

```
40 {
41   submodul_t   submodul[MAX_SUB_MODULS];
42   TYPE_MODUL   data;
43   uint32_t     max_req;
44   uint32_t     num_act_req;
45 } modul_t;
46
47 typedef struct
48 {
49   modul_t       modul[MAX_MODULS];
50 } keylist_t;
51
52
53 typedef struct
54 {
55   TYPE_MODUL         modul;
56   TYPE_SUP_MODUL     submodul;
57   TYPE_SENSOR        sensor;
58 } SYS_KEY_t;
59
60 void keylist_fill(void);
61 uint8_t keylist_check(SYS_KEY_t* p_request);
62 void decrease_keylist_counter(TYPE_MODUL modul);
63 void increase_keylist_counter(TYPE_MODUL modul);
64 #endif  /* TEST_SYSTEM_H__ */
```

Listing D.6: Header des Testsystems für den Bluetooth Low Energy Server

```
1 /*
2 # Autor: M. Pyka
3 # Date: 27.12.20
4 # Name: test_system.c
5 */
6 #include "test_system.h"
7 /********************** DEBUGG
      ************************************/
8 #define DEBUG
```

```
9  #ifdef DEBUG
10 #define dprintf(...) \
11 do { \
12 printf("%s(): ", __func__); \
13 printf(__VA_ARGS__); \
14 } \
15 while (0)
16 #else
17 #define dprintf(...)
18 #endif
19 volatile keylist_t g_keylist_t;
20 /**@brief Function for initializing the keylist.
21  *
22  * @details Initializes the keylist.
23  */
24 void keylist_fill(void)
25 {
26 #ifdef WITH_KEY_CHECK
27   /* Akkumodul */
28   g_keylist_t.modul[0].data = 0x7a;
29   g_keylist_t.modul[0].max_req = 0x07;
30       /* 1. Zelle */
31     g_keylist_t.modul[0].submodul[0].data = 0x8c;
32       g_keylist_t.modul[0].submodul[0].sensors[0] = 0x3f;
33       g_keylist_t.modul[0].submodul[0].sensors[1] = 0x6c;
34       /* 2. Zelle */
35     g_keylist_t.modul[0].submodul[1].data = 0x6e;
36       g_keylist_t.modul[0].submodul[1].sensors[0] = 0xfc;
37       g_keylist_t.modul[0].submodul[1].sensors[1] = 0x92;
38       /* 3. Zelle */
39     g_keylist_t.modul[0].submodul[2].data = 0xc1;
40       g_keylist_t.modul[0].submodul[2].sensors[0] = 0xe0;
41       g_keylist_t.modul[0].submodul[2].sensors[1] = 0x35;
42       /* Temperatur */
43     g_keylist_t.modul[0].submodul[3].data = 0x53;
44       g_keylist_t.modul[0].submodul[3].sensors[0] = 0x31;
```

```
45      g_keylist_t.modul[0].submodul[3].sensors[1] = 0x21;
46
47  /* Behaeltermodul */
48  g_keylist_t.modul[1].data = 0xac;
49  g_keylist_t.modul[1].max_req = 0x03;
50      /* Fuellstand */
51    g_keylist_t.modul[1].submodul[0].data = 0xde;
52      g_keylist_t.modul[1].submodul[0].sensors[0] = 0xc3;
53      g_keylist_t.modul[1].submodul[0].sensors[1] = 0x8b;
54      /* Feuchtigkeit */
55    g_keylist_t.modul[1].submodul[1].data = 0x8b;
56      g_keylist_t.modul[1].submodul[1].sensors[0] = 0x58;
57      g_keylist_t.modul[1].submodul[1].sensors[1] = 0x6c;
58      /* Druck */
59    g_keylist_t.modul[1].submodul[2].data = 0x96;
60      g_keylist_t.modul[1].submodul[2].sensors[0] = 0x8b;
61      g_keylist_t.modul[1].submodul[2].sensors[1] = 0xc1;
62      /* Temperatur */
63    g_keylist_t.modul[1].submodul[3].data = 0xbb;
64      g_keylist_t.modul[1].submodul[3].sensors[0] = 0x96;
65      g_keylist_t.modul[1].submodul[3].sensors[1] = 0xc9;
66
67  /* Buerstenmodul */
68  g_keylist_t.modul[2].data = 0x87;
69  g_keylist_t.modul[2].max_req = 0x10;
70      /* Hall */
71    g_keylist_t.modul[2].submodul[0].data = 0x0c;
72      g_keylist_t.modul[2].submodul[0].sensors[0] = 0x0c;
73      g_keylist_t.modul[2].submodul[0].sensors[1] = 0xbb;
74      /* Feuchtigkeit */
75    g_keylist_t.modul[2].submodul[1].data = 0x33;
76      g_keylist_t.modul[2].submodul[1].sensors[0] = 0x2d;
77      g_keylist_t.modul[2].submodul[1].sensors[1] = 0xbb;
78      /* Druck */
79    g_keylist_t.modul[2].submodul[2].data = 0x5f;
80      g_keylist_t.modul[2].submodul[2].sensors[0] = 0x13;
```

```
81        g_keylist_t.modul[2].submodul[2].sensors[1] = 0xc3;
82        /* Temperatur */
83      g_keylist_t.modul[2].submodul[3].data = 0x24;
84        g_keylist_t.modul[2].submodul[3].sensors[0] = 0x0e;
85        g_keylist_t.modul[2].submodul[3].sensors[1] = 0xed;
86 #endif
87 }
88
89 uint8_t keylist_check(SYS_KEY_t* p_request)
90 {
91 #ifdef WITH_KEY_CHECK
92     TYPE_MODUL i = 0;
93     for(i = 0; i < MAX_MODULS; i++)
94     {
95       /* 1. Check if modul is in keylist. */
96       if(p_request->modul == g_keylist_t.modul[i].data)
97       {
98         if(g_keylist_t.modul[i].num_act_req >=
    g_keylist_t.modul[i].max_req)
99         {
100          return CHECK_KEY_ERROR_MODUL;
101        }
102        TYPE_SUP_MODUL j = 0;
103        for(j = 0; j < MAX_SUB_MODULS; j++)
104        {
105          /* 2. Check if sub Modul exist in keylist. */
106          if(p_request->submodul ==
    g_keylist_t.modul[i].submodul[j].data)
107          {
108            TYPE_SENSOR k = 0;
109            for(k = 0; k < MAX_SENSORS; k++)
110            {
111              /* 3. Check if request exists in keylist. */
112              if(p_request->sensor ==
    g_keylist_t.modul[i].submodul[j].sensors[k])
113              {
```

```
114                 /* return success */
115                 dprintf("Keylist: Success \n");
116                 return CHECK_SUCCESS;
117               }
118             }
119             break;
120           }
121         }
122         break;
123       }
124     }
125     dprintf("Keylist: Error \n");
126     dprintf("\t\tModul:\t%X \n",p_request->modul);
127     dprintf("\t\tSubModul:\t%X \n",p_request->submodul);
128     dprintf("\t\tRequest:\t%X \n",p_request->sensor);
129     dprintf("Not in Keylist\n");
130     /* return error */
131     return CHECK_KEY_ERROR;
132 #endif
133 }
134 void increase_keylist_counter(TYPE_MODUL modul)
135 {
136 #ifdef WITH_KEY_CHECK
137   TYPE_MODUL i = 0;
138   for(i = 0; i < MAX_MODULS; i++)
139   {
140       /* 1. Check if modul is in checklist. */
141       if(modul == g_keylist_t.modul[i].data)
142       {
143           dprintf("numactreq: %d\n",
     g_keylist_t.modul[i].num_act_req);
144           /** Save some Data for request limitation **/
145           g_keylist_t.modul[i].num_act_req += 1;
146           return;
147       }
148   }
```

```
149 #endif
150 }
151
152 void decrease_keylist_counter(TYPE_MODUL modul)
153 {
154 #ifdef WITH_KEY_CHECK
155   TYPE_MODUL i = 0;
156   for(i = 0; i < MAX_MODULS; i++)
157   {
158       /* 1. Check if modul is in checklist. */
159       if(modul == g_keylist_t.modul[i].data)
160       {
161          dprintf("numactreq:
    %d\n",g_keylist_t.modul[i].num_act_req);
162          if(g_keylist_t.modul[i].num_act_req > 0)
163          {
164             g_keylist_t.modul[i].num_act_req -= 1;
165          }
166          return;
167       }
168   }
169 #endif
170 }
```

Listing D.7: Quellcode des Testsystems für den Bluetooth Low Energy Server

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort Datum Unterschrift im Original