Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Michel Rottleuthner

Interprozesskommunikation zwischen Android- und Mikrocontrollersystemen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Michel Rottleuthner

Interprozesskommunikation zwischen Android- und Mikrocontrollersystemen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Thomas Lehmann
Zweitgutachter: Prof. Dr. Hans Heinrich Heitmann

Eingereicht am: 9. Januar 2015

Michel Rottleuthner

Thema der Arbeit
Interprozesskommunikation zwischen Android- und Mikrocontrollersystemen

Stichworte
Interprozesskommunikation, IPC, Android, Accessory, USB, MCU, Mikrocontroller, Embedded

Kurzzusammenfassung
Diese Thesis zeigt die Entwicklung eines Frameworks zur Interprozesskommunikation zwischen Android- und Mikrocontrollersystemen. Es werden die Grundlagen der verwendeten Technologien dargelegt und Anforderungen an das System definiert. Daraufhin werden zur Verfügung stehende Werkzeuge und ähnliche Projekte analysiert um ein Konzept zu erarbeiten. Mit diesem als Basis wird im Anschluss die Implementierung durchgeführt, woraufhin das System direkt in Beispielanwendungen angewandt wird. Den Abschluss der Arbeit bildet die Evaluierung um die Einsatz- und Leistungsfähigkeit des Frameworks zu prüfen und die Diskussion, in welcher Ergebnisse und mögliche Verbesserungen dargelegt werden. Die für diese Thesis gesetzten Ziele wurden erreicht und somit wurde ein funktionierendes System realisiert.

Michel Rottleuthner

Title of the paper
Interprocesscommunication between Android- and microcontroller systems

Keywords
inter-process communication, IPC, Android, accessory, USB, MCU, microcontroller, embedded

Abstract
This thesis presents the development of a framework for inter-process communication between Android and microcontroller systems. It describes the basics of the used technologies and the requirements for the system are defined. After that available tools and similar projects are analyzed to create a concept. With that as a basis, the following implementation is executed, after which the system is applied directly on sample applications. The work is finished with the evaluation of the use and performance of the framework and the discussion where the results and possible improvements are noted. The objectives of this thesis were achieved and therefore a working system was implemented.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

Eingebettete Systeme sind mittlerweile durch die Vielzahl an Einsatzmöglichkeiten und die stetig günstiger werdende Fertigung allgegenwärtig und werden in unzähligen Systemen, von Steueranlagen in der Industrie über Unterhaltungselektronik, bis hin zu Küchengeräten eingesetzt. Viele dieser Anwendungsbereiche verlangen nach einer Möglichkeit mit anderen Systemen zu kommunizieren um so Einfluss auf deren Steuerung oder Zugriff auf deren Daten zu erhalten.

1.1. Motivation

Die in Steuerungs- und Sensorsystemen häufig eingesetzten Mikrocontroller unterliegen bedingt durch deren Architektur oder deren Einsatzzweck häufig Einschränkungen, was deren Dynamik und oft auch Leistungsfähigkeit betrifft. Diese Einschränkungen können beispielsweise durch Anforderungen wie Echtzeitfähigkeit, Autarkie (z.B. bei Batteriebetrieb) oder anderen Anforderungen entstehen. Bei Echtzeitsystemen werden Einschränkungen welche ein System statisch gestalten teilweise wohlwollend in Kauf genommen um einfacher, qualitativ hochwertige Aussagen und Vorhersagen zu deren Verhalten treffen und garantieren zu können. Um nun Anwendungen auszuführen, welche durch die genannten Einschränkungen nicht möglich sind, oder das Echtzeitverhalten eines Systems zerstören, ist ein möglicher Ansatz die kritischen Systeme zu kapseln und mit anderen externen Systemen zu koppeln um diese darüber mit zeitlich unkritischen Funktionalitäten zu erweitern. Eine potentiell sehr interessante Plattform für diesen Zweck stellen aktuelle Smartphones und ähnliche mobile Geräte dar. Diese wurden in den letzten Jahren in Bezug auf deren Leistungsfähigkeit zu performanten kleinst-Computern weiterentwickelt, welche verglichen mit kleinen eingebetteten Systemen, auch aufwändige Berechnungen und verhältnismäßig große Datenmengen problemlos verarbeiten können. Des Weiteren haben heutige Mobilgeräte in der Regel eine Vielzahl von Kommunikationsschnittstellen (Bluetooth, NFC, WLAN, GSM, UMTS, LTE, etc.), diverse Sensoren (Gyro-, Accelero-, Hygro- und Barometer, GPS, Licht, Temperatur, etc.) und Peripheriegeräte (Kamera, Mikrofon, Lautsprecher, Touchscreen, Knöpfe, etc.), welche sich für unzählige Anwendungen als praktikabel erweisen. Durch den vermehrten Einsatz von

gleichen Betriebssystemlösungen wie Android, bei unterschiedlichen Herstellern, ist weiterhin eine einheitliche und stark verbreitete Softwareplattform entstanden. Diese ermöglicht es Programme zu entwickeln die mit einer sehr großen Zahl an Geräten kompatibel ist, ohne sie für jeden Hersteller speziell anpassen zu müssen. Für Entwickler ergibt sich dadurch außerdem der Vorteil, Software in einer Hochsprache und mit Unterstützung umfangreicher Bibliotheken entwickeln zu können, welche ein hohes Maß an Abstraktion zulassen und den Entwicklungsprozess erheblich beschleunigen können.

1.2. Ziel

Im Rahmen dieser Arbeit soll ein System zur gleichberechtigten Kommunikation von eng gekoppelten Software-Komponenten zwischen einem Android-Gerät und einem Mikrocontroller entworfen werden. Dieses soll ermöglichen Daten zur Laufzeit zwischen Anwendungen auf einem Android-Gerät und einem Programm auf einem Mikrocontroller auszutauschen. Sowohl eine synchrone als auch asynchrone Datenübertragung ist zu unterstützen. Es ist abzuwägen ob ein Nachrichtenbasiertes System oder ein RPC-Ansatz besser zur Umsetzung geeignet ist. Der verwendete Kommunikationskanal muss austauschbar sein. Auf Basis des Systems sind anschließend Beispielanwendungen zu entwerfen und implementieren. Mit einer der Beispielanwendungen ist eine Zeitsynchronisation auf dem Mikrocontroller zu realisieren. Diese und weitere Anwendungen sollen der allgemeinen Veranschaulichung des Einbindens der erstellten Software und deren Funktionsweise dienen. Im Anschluss ist unter anderem mithilfe der Beispielanwendungen und geeigneten Werkzeugen das System zu evaluieren.

2. Grundlagen

In diesem Kapitel wird in die Grundlagen der verwendeten Technologien eingeführt. Dazu werden erst einige Begriffe allgemein erläutert und im Anschluss wird deren Bedeutung innerhalb der eingesetzten Plattformen aufgezeigt. Um die folgenden Kapitel leichter verständlich zu machen, werden zunächst diverse Grundlagen in Bezug auf den Kontext dieser Arbeit dargelegt.

2.1. Begriffe

2.1.1. Prozess

Da in dieser Thesis Aspekte der Interprozesskommunikation beleuchtet werden, ist zunächst zu klären wie der Begriff Prozess im Rahmen dieser Arbeit zu deuten ist. Im Kontext eines Betriebssystems ist ein Prozess vereinfacht ausgedrückt ein Programm während seiner Ausführung. (vgl. [29]). Diese grobe Beschreibung unter Berücksichtigung der Eigenschaft, dass ein Prozess nur Zugriff auf seinen eigenen Speicherbereich hat, ist ausreichend. Des Weiteren soll die Ausführung eines Programms auf einem Mikrocontroller in diesem Sinne ebenfalls als Prozess verstanden werden, auch wenn auf diesem kein Betriebssystem vorhanden ist.

2.2. Interprozesskommunikation

Interprozesskommunikation bezeichnet somit eine spezielle Art der Interaktion zwischen mindestens zwei Prozessen. Um den Begriff der Interprozesskommunikation genauer abzugrenzen, dient die folgende Definition: „Wenn Prozesse untereinander kommunizieren, aufeinander warten oder Daten und Ressourcen austauschen müssen, werden so genannte Interprozesskommunikationen verwendet“ ([29], Kap. 9).
Das zu realisierende System soll hauptsächlich zur Kommunikation und dem Austausch von Daten genutzt werden.

Als gängige Hilfsmittel zur Interprozesskommunikation die schwerpunktmäßig nicht nur dazu eingesetzt werden, zwischen Prozessen auf dem selben Physikalischen System zu kom-

munizieren, zählen unter anderem Message Queues, Sockets, und allgemein Streams (vgl. [29], Kap. 9.1).

Für diese Arbeit wird angestrebt ein Nachrichtenbasiertes System zu realisieren, das Stream-orientierte Übertragungsmechanismen verwendet. Bei der Kommunikation in verteilten Systemen mittels Nachrichten, werden viele unterschiedliche Konzepte verwendet. Man kann hierbei grob unterscheiden zwischen persistenter und transienter Kommunikation. Wobei diese weiter in asynchron und synchron unterteilt werden können. Ein einfaches Beispiel für eine persistente Kommunikation ist ein Dienst zum Versand von E-Mails. Bei diesem ist es wichtig, dass die zu übertragenden Informationen solange von dem Dienst vorgehalten werden, bis sie an den Empfänger übermittelt werden. Durch diese Vorgehensweise muss der Sender die Daten nur an den Dienst übergeben und nicht darauf warten, dass dieser die Nachricht an den Empfänger übermittelt hat. In Umgekehrter Betrachtungsrichtung gilt dies auch für den Empfänger, der dadurch nicht zwingend erreichbar sein muss, während der Sender eine Nachricht absetzt (siehe Abbildung 2.1).

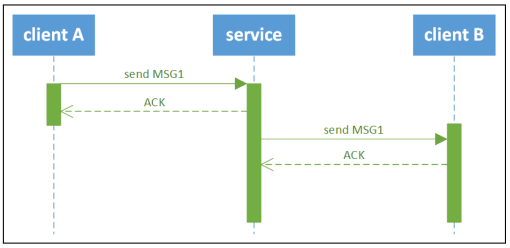


Abbildung 2.1.: Sequenzdiagramm persistente Kommunikation

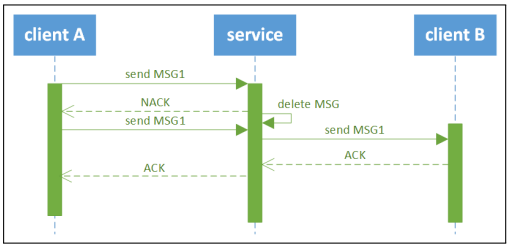


Abbildung 2.2.: Sequenzdiagramm transiente Kommunikation

Anders als bei persistenter Kommunikation wird bei der transienten Kommunikation eine Nachricht nur vorübergehend für die Zeit gespeichert, in der Beide Kommunikationspartner verfügbar sind (siehe Abbildung 2.2). Ist also der Empfänger einer Nachricht momentan nicht erreichbar wird sie verworfen.

Der Unterschied zwischen einer synchronen und asynchronen Kommunikation zeigt sich darin ob der Programmablauf der Anwendung während dem Ausführen von Sende- oder Empfangsroutinen blockiert wird. Bei Asynchroner Kommunikation wird vom sendenden Programm lediglich das Übertragen von Daten angefordert und sofort mit dem Programmablauf fortgefahren. Häufig werden dazu die Daten bei der Anforderung zunächst kopiert damit das Übertragungssystem parallel zum regulären Programmablauf mit dem Senden beginnen kann.

Bei synchroner Kommunikation wird erst mit der Programmausführung des Senders fortgefahren, sobald die Anforderung akzeptiert wurde. Dieses Akzeptieren kann je nach Anwendungsfall unterschiedlich bestimmt sein, wobei meistens eines der folgenden drei Verfahren genutzt wird. Beispielsweise kann es bereits ausreichen, dass die Anforderung zur Übertragung erfolgreich an eine Middleware weitergeben wurde, welche sich um die eigentliche Übertragung kümmert. Als zweite Variante wird das Blockieren erst dann beendet, wenn die Nachricht vollständig an den Empfänger übertragen wurde. Die dritte Möglichkeit wird bei der Umsetzung von Remote Procedure Calls (RPC) angewandt. Dabei wird blockiert, bis die Verarbeitung der Nachricht auf der Empfängerseite vollständig abgeschlossen ist. (vgl. [20] S.148 - 150)

2.3. Android

Android ist eine weit verbreitete Software Plattform die auf mobile Endgeräte, Fernseher, Wearable-Computer und Fahrzeug-Multimedia spezialisiert ist. Sie wird vom „Android Open Source Project“ entwickelt, das von der Firma Google geleitet wird. Der bekannteste Ableger des Projektes ist das Android Betriebssystem für Mobilgeräte wie Smartphones oder Tablet-Computer. Mit über einer Million neu aktivierten Geräten pro Tag und einer Verbreitung in mehr als 190 Ländern bietet Android den Nutzern und Entwicklern von Applikationen eine ausgereifte und breit aufgestellte Plattform (vgl. [2]).

2.3.1. Architektur

Das Android Betriebssystem basiert auf einem Linux System für mehrere Benutzer, wobei jede installierte App einen Nutzer darstellt. Durch diesen Aufbau können die unter Linux verwendeten Nutzer-Zugriffsrechte dazu verwendet werden jeder App eigene Berechtigungen zu erteilen. Standardmäßig läuft jede App in einem eigenen Linux Prozess und jeder Prozess läuft in einer eigenen Virtuellen Maschine. Dadurch werden Apps voneinander Isoliert ausgeführt. Die Linux Prozesse werden vom Betriebssystem verwaltet und somit z. B. gestartet wenn eine App ausgeführt werden soll oder beendet, wenn Ressourcen freigegeben werden müssen (vgl. [6]).

2.3.2. Rechteverwaltung

Die Rechteverwaltung umfasst neben reinen Dateizugriffsrechten auch erweiterte Rechte wie z. B. Zugriff auf Kamera, Bluetooth, Kontakte, Nachrichten und weitere. Jede App die zur

Ausführung bestimmte Rechte benötigt, muss diese vorher vom Betriebssystem einholen. Die geforderten Rechte werden bei der Installation einer App direkt dem Benutzer angezeigt und sind von diesem zu bestätigen. Die möglichen Berechtigungen beschränken sich nicht auf die vom Betriebssystem vorgegebenen, denn Apps können auch selbst spezielle Berechtigungen einführen und diese für bestimmte Aktionen einfordern (vgl. [9]).

2.3.3. Komponenten

Eine Android-App kann im Wesentlichen aus vier Arten von Komponenten bestehen, wobei nicht jede davon zwingend erforderlich ist. Es existieren Activities, Services, Content provider und Broadcast receiver. Diese Komponenten unterscheiden sich grundlegend voneinander und sind jeweils für einen bestimmten Einsatzzweck gedacht.

Activities bilden die Basis für die GUI (Graphical User Interface) einer App. Sie werden verwendet um einzelne, meist den Bildschirm füllende Anzeigen, zu realisieren. Jede Activity stellt dem Nutzer einen bestimmten Teil der Benutzerschnittstelle zur Verfügung. Ein Beispiel hierfür wäre eine Activity zum Verfassen einer neuen E-Mail, welche Eingabefelder für E-Mail Adressen den Nachrichtentext als auch Buttons zum Anhängen einer Datei oder dem Versenden der E-Mail beinhalten könnte. Eine weitere Activity könnte eine Liste zur Auswahl der Kontakte präsentieren. Aus mehreren solcher Activities besteht aus Nutzersicht eine fest zusammen hängende grafische Oberfläche, durch die frei navigiert werden kann. Technisch gesehen sind einzelne Activities unabhängig voneinander und können beliebig gestartet werden, was in Hinsicht auf die Benutzerfreundlichkeit zu Problemen führen kann. Activities können nicht nur von der App gestartet werden, von der sie ein Bestandteil sind, sondern auch von fremden Apps, wenn es diesen gestattet wird. Das kann besonders dann sinnvoll sein, wenn eine bestimmte Funktion einer App auch anderen Apps zur Verfügung stehen soll. In Bezug auf das obige Beispiel könnten fremde Apps die E-Mail Activity direkt starten um eine Datei anzuhängen.

Services werden verwendet um zeitaufwändige Aufgaben im Hintergrund zu erledigen, die selbst keine grafische Oberfläche besitzen. Ein einfacher Anwendungsfall in dem der Einsatz eines Services Sinn macht ist zum Beispiel ein Dateidownload der im Hintergrund ausgeführt werden soll, auch wenn die laufende App vor Fertigstellung des Downloads verlassen wird. Zwei besonders wichtige Fakten über Services aus Entwicklersicht sind, dass ein Service weder

das Äquivalent zu einem Prozess noch zu einem Thread darstellt. Ein Service läuft im Normalfall im selben Prozess wie die App zu der er gehört (vgl. [17]).

2.3.4. IPC unter Android

Um zwischen Prozessen unter Android zu kommunizieren gibt es diverse Ansätze. Die Hilfsmittel bilden hierbei Intents, Binder, Messenger, Broadcast Receiver und Content Provider. Es gelten allerdings einige Einschränkungen, wenn man über Prozessgrenzen hinweg kommunizieren möchte. Dies ist für Anwendungen die nicht in derselben apk-Datei geliefert werden unumgänglich, da diese jeweils in ihrem eigenen Prozessraum ausgeführt werden.

Intents werden unter Android für eine stark abstrahierte Form der Interprozesskommunikation verwendet. Ein Intent dient dazu einen Vorgang zu beschreiben der ausgeführt werden soll. Ein solcher Vorgang kann z.B. das Starten einer Activity oder eines Services sein, aber auch abstrakte Anweisungen um z. B. dem Nutzer ein Bild anzuzeigen, können über Intents modelliert werden. Sie können gleichzeitig dazu verwendet werden Daten zwischen Prozessen auszutauschen (vgl. [16]).

Content Provider werden verwendet um verschiedenen Apps Zugriff auf gemeinsame Daten zu ermöglichen. Dabei können die geteilten Daten über einen URI (Uniform Resource Identifier) identifiziert und über Prozessgrenzen hinweg übertragen werden. Notwendig wird ein solcher Mechanismus durch die grundlegende Datenkapselung einzelner Apps und der gleichzeitigen Anforderung dennoch eine einfache Möglichkeit zum gemeinsamen Datenzugriff zwischen diesen zu erreichen. Die Daten in einem Content Provider werden dabei ähnlich wie bei Relationalen Datenbanken als Tabelle strukturiert und können in vergleichbarer Weise über eine vorgegebene API (Application Programming Interface) abgefragt und gegebenenfalls verändert werden. Ein Beispiel für einen Content Provider liefert das integrierte Wörterbuch von Android. In dieses können vom Nutzer z.B. eigene Wörter hinzugefügt werden, die dem System zuvor unbekannt waren. Durch die Nutzung des systemeigenen Content Providers für das Wörterbuch, kann in jeder Anwendung dieselbe Konsistente Datenbasis verwendet und dadurch eine nahtlose Integration der Daten erreicht werden. Weitere Beispiele für Einsatzmöglichkeiten sind eine zentrale Speicherung von Kontakten oder Einstellungen (vgl. [9] und [3]).

Binder ist eine Basisimplementierung für entfernt zugreifbare Objekte innerhalb eines Android-Systems. Es stellt grundlegende Funktionen zur Implementierung eigener RPC-Protokolle bereit. Dabei werden zur Datenübertragung Parcelable-Objekte verwendet, die laut Dokumentation für hoch performanten IPC-Transport entworfen wurden (vgl. [8]). Parcelable-Objekte werden ähnlich wie von Serializable-Objekten aus Java bekannt verwendet. Die Serialisierungsvorgänge müssen aber im Gegensatz zu Serializable manuell implementiert werden. Der Vorteil von Parcelable besteht in der höheren Geschwindigkeit (vgl. [15] und [14]).

AIDL steht für „Android Interface Definition Language“ und ist, wie der Name zeigt, eine Schnittstellenbeschreibungssprache. Möchte man fremden Anwendungen Funktionalität zur Verfügung stellen, muss der Zugriff darauf einheitlich beschrieben werden. Mit der AIDL und zusätzlichen Funktionen der bereitgestellten Android Development Tools wird die Möglichkeit eröffnet, automatisiert aus den Schnittstellendefinitionen entsprechende Java-Dateien zu generieren, welche die Funktionalität zur Übertragung der Übergabe- und Rückgabeparameter implementieren. Zusätzlich wird ein Java-Interface deklariert, auf welchem die konkrete Implementierung der definierten Methoden aufgebaut werden kann. Generierte Klassen verwenden intern die o.g. Binder-Funktionalitäten und basieren somit ebenfalls auf der performanten Datenübertragung mittels Parcelable-Objekten. Ein Vorteil gegenüber der direkten Verwendung der Binder-Klasse besteht darin, kein eigenes RPC-Protokoll implementieren zu müssen. Gleichzeit bleibt aber die gute Performanz der direkten Verwendung des Binders vorhanden. Entfernte Aufrufe über AIDL werden dabei an einen Thread-Pool innerhalb des Ziel-Prozesses delegiert und synchron ausgeführt. Hierdurch entsteht zusätzlich der Vorteil, dass mehrere Anwendungen parallel auf die Funktionen zugreifen können.

Messenger ist unter Android ein Mechanismus der ebenfalls auf Binder aufbaut, jedoch etwas einfacher gestaltet ist als AIDL. Ein Messenger wird auch dazu verwendet zwischen unterschiedlichen Prozessen zu Kommunizieren, bietet jedoch kein Multithreading und kann ausschließlich über Nachrichten Kommunizieren.

Broadcast Receiver sind Komponenten die Intents empfangen können die als Broadcast versendet wurden. Diese Broadcasts werden häufig vom System versendet, wenn ein bestimmtes Ereignis eintritt an dem potentiell viele Empfänger interessiert sind. So werden sie z.B. eingesetzt um systemweit mitzuteilen, dass der Akku schwach ist oder der Bildschirm abgeschaltet wurde. Broadcasts können aber nicht nur vom System selbst sondern auch von eigenen Anwendungen versendet werden. Sie bieten somit eine einfache Form der Interprozesskommunikation zwischen verschiedenen Android Applikationen. Abgesehen von systemweiten

Broadcasts lassen sich auch lokal innerhalb des eigenen Prozesses Broadcasts versenden die nur von Komponenten der eigenen Anwendung empfangen werden können. Diese lassen sich dann auch für das Übertragen von privaten Nutzerdaten einsetzen bei denen es kritisch wäre, wenn sie von anderen Anwendungen im System abgefangen werden (vgl. [9] und [7]).

2.3.5. Prozessmodell unter Android

Das Prozessmodell von Android unterscheidet sich grundlegend von denen auf Linux oder Windowssystemen. Bei aktuellen Desktop Betriebssystemen mit Multitasking wird der Lebenszyklus eines Prozesses hauptsächlich durch das Scheduling und deren Eigenschaften bestimmt. Dadurch lässt sich der Lebenszyklus eines Prozesses mit einem Modell darstellen, in dem der Prozess, abhängig von der Verfügbarkeit benötigter Ressourcen, zwischen den Zuständen running, runnable, sleeping etc. wechselt. Unter Android liegt darauf eine weitere Abstraktionsschicht, die das Laufzeitverhalten der Anwendungen erheblich beeinflusst. Ein besonders großer Unterschied besteht darin, dass die Lebenszeit eines Prozesses nicht direkt von der Anwendung sondern vom System gesteuert wird (vgl. [5]). So gibt es nicht wie bei Desktop-Applikationen üblich einen Zentralen Einsprungpunkt in dem eine Dauerschleife die Prozesslaufzeit darstellt und bei deren Verlassen das Programm beendet wird, sondern der Lebenszyklus eines Prozesses wird vom Betriebssystem mittels Events bzw. Callbacks gesteuert. In Abbildung 2.3 ist die Steuerung Mittels Events dargestellt.

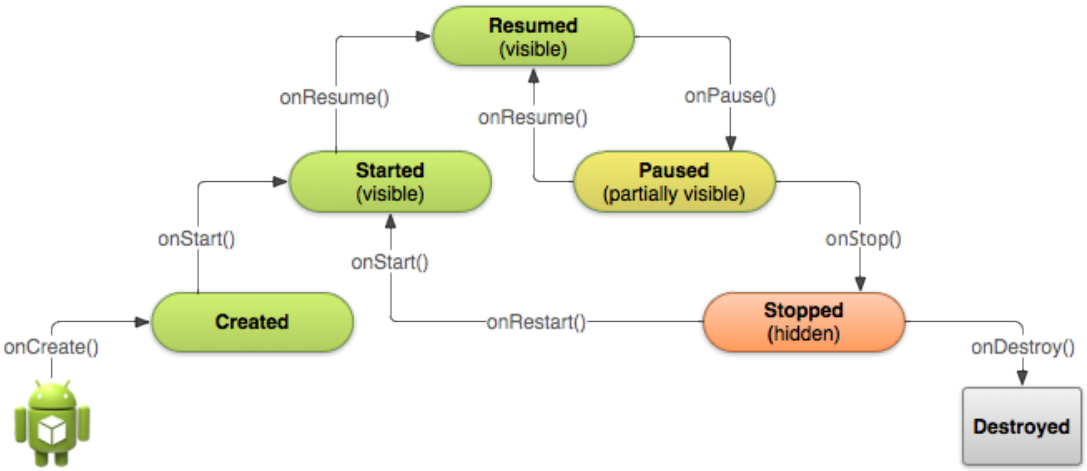


Abbildung 2.3.: Lebenszyklus einer Android Activity (Quelle:[12])

Zu beachten ist dabei, dass eine Anwendung die sich im Paused oder Stopped Zustand befindet keinen Code ausführen kann. Durch diese Unterschiede müssen besondere Rahmenbedingungen eingehalten werden. Eine normale Benutzeranwendung unter Android kann jederzeit vom Nutzer oder vom System selbst pausiert oder unter bestimmten Umständen ganz beendet werden. Dies ist auf mehrere Gründe zurückzuführen, wie z.B. dem Sparen von Ressourcen wie CPU-Zeit und Arbeitsspeicher und dadurch einhergehend die in Mobilgeräten besonders wichtige Akkukapazität. Aber auch ein flexibles und dennoch konsistentes Nutzererlebnis erfordert gewisse Einschnitte die sich merklich auf das Prozessverhalten unter Android auswirken. So soll es einem Benutzer beispielsweise immer durch eine einfache Eingabe möglich sein die momentane Anwendung zu verlassen, um die Oberfläche in die für den Nutzer gewohnte Ausgangsposition zu versetzen.

2.4. Hardware

Abschließend wird erläutert welche Hardware für die Umsetzung verwendet werden soll. Das verwendete Android-Gerät kann nahezu beliebig gewählt werden da Abhängigkeiten und Kompatibilität sich bis auf Ausnahmen nur durch das Betriebssystem ergeben, welches mindestens Version 2.3.4 (API level 10) entsprechen muss. Zur Durchführung der Arbeit wurde hauptsächlich ein Samsung Galaxy Tab 3 7.0 mit Android Version 4.4.2 verwendet.

Als Mikrocontroller-Plattform wird aus mehreren Gründen das ARM Cortex-M4 basierte STM32F4DISCOVERY Entwicklerbord benutzt. Es bietet 1 MB Flash, 192 KB RAM und ein integriertes ST-LINK/V2-Modul zum Flashen und Debuggen. Aktuell werden STM32-Basierte Plattformen an der HAW-Hamburg in diversen Projekten (POs / WPs) und anderen Praktika eingesetzt. Eines dieser Projekte ist das interdisziplläre Airborne Embedded Systems-Projekt in welchem das zu entwickelnde System später potentiell eingesetzt werden soll. Des weiteren sollen durch die Wahl dieser Plattform die Voraussetzungen für einen späteren produktiven Einsatz verbessert werden indem eine den Studenten bekannte Umgebung verwendet wird.

2.5. Werkzeuge

Zur Entwicklung für die MCU stehen diverse IDEs zur Verfügung. Für diese Thesis wurde CoIDE von CooCox gewählt welche eine unproblematische Konfiguration, aktive Unterstützung durch den Hersteller und eine zu vielen Systemen kompatible IDE verspricht.

Die Implementierung der Android Komponenten soll mit den Android Developer Tools stattfinden, welche aktuell den Standard zur Entwicklung für Android bilden. Im wesentlichen bestehen die Developer Tools aus einer angepassten Eclipse Variante als IDE, welche für die Android-Entwicklung zusätzliche Plugins und weitere Anwendungen wie einen Geräteemulator beinhaltet. Während der praktischen Arbeit an dieser Thesis wurde mit dem Android Studio die offizielle IDE abgelöst[1]. Dies wurde nicht berücksichtigt, da zum Zeitpunkt der Werkzeugwahl das Android Studio ausschließlich in einer Betaversion erhältlich war und Wert auf eine ausgereifte Entwicklungsumgebung gelegt wurde.

[1] http://tools.android.com/recent/androidstudio10released

3. Anforderungen

In diesem Kapitel werden Anforderungen erarbeitet, die das zu entwickelnde Framework erfüllen soll. Dazu wird zunächst der Einsatzbereich genauer erläutert und im Anschluss daraus funktionale und nicht funktionale Anforderungen abgeleitet.

3.1. Einsatz

Das zu entwickelnde Framework soll hauptsächlich für eine enge Kopplung von Systemen konzipiert werden, deren Einsatzzweck und Funktionsweise in der Regel schon beim Einbinden des Frameworks bekannt ist. Hochdynamische Systeme welche zur Laufzeit mit neuen Funktionen ausgestattet werden, werden nicht primär adressiert. Das System soll vielmehr die Entwicklung der Kommunikationskomponente eines logisch eng gekoppelten Systems erleichtern das zwischen Beiden Plattformen kommunizieren soll. Durch die Fokussierung auf eng gekoppelte Systeme wird die Annahme getroffen, dass die kommunizierenden Komponenten sich im Voraus bekannt sind. Außerdem soll das Framework schwerpunktmäßig zur Kopplung von Systemen entwickelt werden, die sich lokal am selben Standort befinden. Eine Verwendung zur Kopplung von Systemen die sich an weit voneinander entfernten Standorten befinden wird zwar nicht ausgeschlossen, aber für das Festlegen der Anforderungen sollte der Haupteinsatzzweck stärker beachtet werden.

3.2. Nicht funktionale Anforderungen

3.2.1. Kompatibilität

Die Kompatibilitätsanforderung betrifft vor allem die Android-Implementierung und Komponenten des Mikrocontrollers die nicht notwendigerweise plattformspezifisch sind. Es soll sichergestellt werden, dass das entwickelte Framework mit möglichst vielen Android-Geräten kompatibel ist. Die Komponenten auf der MCU sollen nach Möglichkeit einfach auf andere Systeme portierbar sein.

3.2.2. Modularität

Die Struktur des Frameworks soll Modular aufgebaut sein dadurch kann die allgemeine Erweiterbarkeit und Wiederverwendbarkeit einzelner Komponenten erreicht werden.

3.2.3. Effizienz

Es wird ein effizientes Einbinden des Frameworks in neue und bestehende Projekte gefordert.

3.2.4. Performanz

Der Overhead für die Übermittlung der Daten ist in Bezug auf CPU-Zeit als auch Bandbreite möglichst klein zu halten. Die Geschwindigkeit und Reaktionszeit sollen schnell genug sein um eine Zeitsynchronisation auf Basis des Frameworks umsetzen zu können. Außerdem soll es möglich sein regelmäßig innerhalb von 20 ms einen Datenblock mit der Größe von 512 Byte in einen Prozess auf einem anderen System zu übertragen.

3.3. Funktionale Anforderungen

Das System soll synchrone und asynchrone, Nachrichten-basierte Kommunikation ermöglichen. Der verwendete Kommunikationskanal soll austauschbar sein. Zielsysteme sind ein Android-System und eine 32-Bit MCU ohne Betriebssystem.

3.4. Verteiltes System

Verteilte Systeme werden von Experten sehr unterschiedlich definiert da sie sehr vielseitig sind und für die unterschiedlichsten Einsatzzwecke verwendet werden. In dieser Arbeit wird aufgrund ihrer Einfachheit die Definition nach Tanenbaum gewählt: “Ein verteiltes System ist eine Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes kohärentes System erscheinen.“ ([20]) (S.19). Damit die Anwendung eines Verteilten Systems für eine Anwendung sinnvoll ist, nennt Tanenbaum vier Ziele, die erfüllt sein sollten. Ein leichter Zugriff auf Ressourcen sollte möglich sein und das Verteilte System sollte weiterhin transparent, offen und skalierbar sein (vgl. [20]). Das Framework, das im Zuge dieser Arbeit entwickelt wird sollte nach Möglichkeit das Erreichen dieser Ziele nicht behindern um später als Basis für ein Verteiltes System dienen zu können.

4. Analyse

In diesem Kapitels werden bekannte Techniken und Verfahren erläutert die sich potentiell zur Realisierung des Systems eignen. Im Besonderen soll hierbei auf die zuvor definierten Anforderungen acht gegeben werden damit die Erkenntnisse aus der Analyse direkt in den Konzeptionellen Teil einfließen können.

4.1. Kopplung von Eingebetteten Systemen

Im Folgenden wird auf die Kopplung von eingebetteten Systemen eingegangen. Dazu wird eine Unterteilung in zwei Perspektiven vorgenommen, wovon sich eine nur auf die hardwarespezifischen und die andere auf die logischen bzw. konzeptuellen Grundlagen der Kopplung beziehen. Aus Sicht der Hardware sind auf Eingebetteten System wie einem Mikrocontroller, Schnittstellen die sich zur Kopplung mit anderen Systemen eignen im großen Umfang vorhanden. In der Regel gibt es diverse verfügbare Anschlüsse die sich als GPIO (General Purpose Input/Output), oder entsprechend konfiguriert, als spezielle standardisierte Hardware-Schnittstellen verwenden lassen. Die verschiedenen Protokolle werden dabei meist mithilfe eines im Prozessor integrierten Bausteins realisiert, der je nach Protokoll und Komplexität die benötigten Funktionalitäten direkt auf Transistorebene bereitstellt. Oft werden diese Komponenten mit zusätzlichen Bibliotheken erweitert, um deren Benutzbarkeit für den Entwickler zu erhöhen oder weitere Funktionen in Form von Software bereitzustellen. Bekannte Beispiele hierfür bilden die Schnittstellen U(S)ART (RS-232 etc.), SPI und Bussysteme wie CAN oder I2C. Des Weiteren können über die genannten Schnittstellen zusätzliche Komponenten an einen Mikrocontroller angebunden werden, welche kabellose Verbindungen über WLAN, Bluetooth, ZigBee, RFID / NFC und andere Standards ermöglichen.
Während auf leistungsstarken Systemen aus Anwendungssicht sehr abstrakte Verfahren zur Kopplung verwendet werden können, sind die verfügbaren Mittel auf eingebetteten Systemen häufig eingeschränkt, da die Ressourcen knapp sind. Zu Gunsten der Performanz wird versucht auf stark abstrahierende Verfahren zu verzichten und somit möglichst nah an der Hardware programmiert. Durch diesen Umstand werden zur Kommunikation zwischen eingebetteten

Systemen häufig Speziallösungen entwickelt, welche die gewünschte Funktionalität möglichst effizient realisieren.

4.2. Konnektivität

Zur Kopplung eines Android Gerätes mit einem Mikrocontroller stehen potentiell einige Möglichkeiten zur Auswahl. Auf der MCU-Seite steht wie bereits unter 4.1 gezeigt ein großes Repertoire an Schnittstellen bereit, das zusätzlich fast beliebig erweitert werden kann. Im Gegensatz dazu fällt bei Android basierten Endgeräten die Anzahl an verfügbaren Schnittstellen und die Erweiterbarkeit wesentlich geringer aus. Die gängigsten, da vom Betriebssystem direkt unterstützten, Schnittstellen sind WLAN (WIFI P2P (Peer-to-Peer)), Bluetooth (inkl. Bluetooth Low Energy), NFC, USB und GSM/UMTS (vgl. [4]).

Um abzuwägen welche von den genannten Schnittstellen sich auch tatsächlich verwenden lassen, muss ein Blick auf die Anforderungen und den Einsatz des Systems geworfen werden. Um die Auswahl weiter einzuschränken werden zunächst offensichtlich ungeeignete Kandidaten ausgeschlossen. Dazu zählt z.B. die Verbindung über das GSM/UMTS Netz. Dies wäre technisch realisierbar jedoch hinge die Kommunikationsfunktion dadurch wesentlich von der Verfügbarkeit der Netzt-Infrastruktur eines externen Betreibers ab und würde Probleme wie eine stark schwankende Bandbreite mit sich bringen. Zusätzlich müssten für einen Datentarif regelmäßige Kosten in Kauf genommen werden.

4.3. USB

4.3.1. Abgrenzung und Allgemeines

Im Folgenden wird die Funktionsweise des Universal Serial Bus (USB) beschrieben und auf dessen Details eingegangen. Die Spezifikationen rund um den USB werden von einer gemeinnützigen Organisation namens „USB Implementers Forum, Inc.“ erarbeitet und verwaltet. Die zugrunde liegende Spezifikation wird stetig aktualisiert und dem technischen Fortschritt entsprechend erweitert. Die im Moment aktuelle Spezifikation liegt in der Version 3.1 vom 26 Juli 2013 vor (vgl. [19]). Sie umfasst ein Dokument von 631 Seiten und baut auf den vorherigen Versionen auf, welche zusätzlich berücksichtigt werden müssen. Diese Tatsache allein stellt bereits sehr anschaulich dar, dass es sich beim USB um einen sehr komplexen und vielseitigen Bus handelt. Da eine vollständige Beschreibung der Spezifikation zu Umfangreich ist, wird nur auf den für diese Arbeit relevanten Teil Bezug genommen. Im Besonderen werden Bestandteile des Standards, die in der Regel bereits direkt von der USB-Hardware umgesetzt werden

nicht weiter erläutert, da Entwickler selten damit in Kontakt treten und dies für diese Arbeit keinen Mehrwert bietet. Diese Beschreibung des USB erhebt in keiner Weise den Anspruch vollständig in Bezug auf das Protokoll oder die Spezifikation zu sein, sondern dient vielmehr dem allgemeinen Verständnis des Bussystems und der Entscheidungen die im Rahmen dieser Arbeit gefällt wurden. Der USB ist aus unterschiedlichen Gründen eine sehr häufig eingesetzte Kommunikationsschnittstelle. Mit über zwei Milliarden neu installierten Geräten pro Jahr ist USB bereits die erfolgreichste PC-Schnittstelle (vgl. [21]). Diese derart hohe Verbreitung liegt unter anderem daran, dass USB zuverlässig, vielseitig einsetzbar und kostengünstig ist. Wie diese Vorteile im Detail aussehen wird in diesem Kapitel gezeigt. USB ist, wie auch der Name vermuten lässt, ein Serieller Bus und überträgt Daten im Half-Duplex Verfahren. An einem USB-Controller können bis zu 127 Geräte angeschlossen werden. (vgl. [21], Table 1-1). Ein verbundenes Gerät kann über den Anschluss mit Energie versorgt werden, wobei laut Version 2.0 Der USB-Spezifikation jedem Gerät 500 mA bei 5V zur Verfügung stehen. USB Unterstützt derzeit vier unterschiedliche Geschwindigkeiten: 5Gbps, 480Mbps, 12Mbps, und 1,5Mbps. Für diese Arbeit ist hauptsächlich der sogenannte Full Speed (12Mbps) relevant, da nur dieser in vollem Umfang von der verwendeten Hardware (STM32F4-DISCOVERY) unterstützt wird. Dennoch wird zum Vergleich an einigen Stellen auf Unterschiede zum nächst schnelleren Hi-Speed Standard hingewiesen. USB funktioniert nach einem Master-Slave Verfahren bei dem der sogenannte Host den Master darstellt und angeschlossene Geräte die Slave-Rolle einnehmen. Jeder Transfer von Daten auf dem Bus wird ausschließlich vom Host initiiert, er regelt wann, wie und wohin Daten übertragen werden. Ein Gerät reagiert nur auf Anweisungen die der Host vorgibt. Außerdem kann über die relevante USB-Version eine Kommunikation immer nur zwischen einem Host und einem Gerät stattfinden. (vgl. [21], S.31)

4.3.2. Endpoints

Konkret werden zur Datenübertragung sogenannte Endpoints verwendet, welche die Enden eines logischen Kommunikationskanals bilden der sich zwischen einem Host und einem Gerät erstreckt. Ein Endpoint wird dabei durch eine Nummer von 0 bis 15 und die Richtung in der die Daten übertragen werden adressiert. Die Richtung wird immer aus Sicht vom Host beschrieben. „IN“ bedeutet vom Gerät zum Host, „OUT“ vom Host zum Gerät. Ein Endpoint den jedes Gerät laut USB-Standard besitzen muss, ist immer auf der Adresse 0 zu finden. Dieser dient zur Konfiguration des Gerätes und kann in beide Richtungen kommunizieren. Ob und welche anderen Arten von Endpoints in einem Gerät Verwendung finden, ist Sache der Gerätehersteller bzw. des Firmwareentwicklers.

4.3.3. Aufbau eines Transfers

Über die Endpoints werden beim USB sogenannte Transfers abgewickelt. Ein Transfer kann im Kontext der USB-Kommunikation abstrakt als ein einzelner, geschlossener Vorgang zur Übertragung von Daten betrachtet werden. Um die generelle Funktionsweise der USB-Kommunikation besser verstehen zu können ist es zunächst von Vorteil die allgemeine Struktur von Transfers zu kennen.

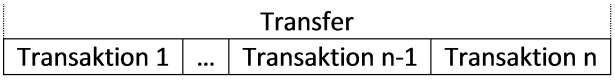


Abbildung 4.1.: Allgemeiner Aufbau eines Transfers (frei nach [21],S. 40 Abb. 2-1)

Jeder Transfer besteht aus einer oder mehreren Transaktionen (siehe Abbildung 4.1), welche wiederum aus verschiedenen Paketen zusammengesetzt werden (siehe Abbildung 4.2).

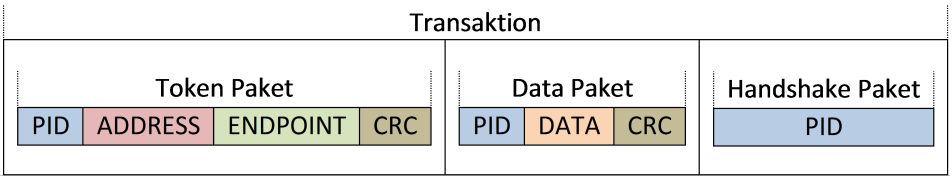


Abbildung 4.2.: Allgemeiner Aufbau einer Transaktion (frei nach [21], S. 40 Abb. 2-1)

Die verschiedenen Pakete einer Transaktion bilden beim USB die kleinste logische Einheit die zur Datenübertragung verwendet wird. Es gibt drei unterschiedliche Arten von Paketen: Token, Data, und Handshake Pakete, von denen nicht alle in jeder Transaktion benötigt werden. Wie aus Abbildung 4.2 ersichtlich enthält ein Paket immer eine Paketidentifikationsnummer (PID) und kann je nach Paket Typ weitere zur Übertragung notwendige Informationen, wie eine Adresse oder Daten, enthalten. Transaktionen werden wie bereits die Endpoints durch Ihre Datenübertragungsrichtung unterschieden. Eine zusätzliche Unterscheidung wird bei Setup-Transaktionen vorgenommen, welche wie normale OUT-Transaktionen Daten vom Host zum Device übertragen. Das ist notwendig da Setup Transaktionen dazu verwendet werden Control Transfers zu initiieren, diese müssen von jedem Gerät akzeptiert und beantwortet werden. Um den unterschiedlichen Anforderungen an die Art der Datenübertragung und verschiedenen Einsatzmöglichkeiten gerecht zu werden, unterstützt der USB vier unterschiedliche Varianten von Transfers. Es gibt Control-, Bulk-, Interrupt- und Isochronous Transfers, deren Unterschiede werden im Folgenden erklärt.

4.3.4. Control Transfer

Dies ist die einzige Art der Übertragung, die jedes USB-Gerät zwangsläufig unterstützen muss. Das liegt daran, dass über Control-Transfers Funktionen bereitgestellt werden, die von der USB-Spezifikation vorgegeben sind. Sie sind jedoch nicht darauf beschränkt und es können weitere gerätespezifische Funktionen mittels Control-Transfer realisiert werden. Mit einem Control-Transfer kann der Host z.B. einem verbundenen Gerät eine Adresse zuweisen und dieses konfigurieren. Damit der Host zuverlässig Control Transfers dazu verwenden kann die angeschlossenen Geräte zu konfigurieren, sind bis Full Speed (12 Mbps) 10%, ab Hi-Speed (480 Mbps) 20% der Bandbreite für diese reserviert.

4.3.5. Bulk Transfer

Bulk Transfers dienen der Übertragung größerer Datenmengen die nicht zeitkritisch sind, aber vollständig und korrekt übertragen werden müssen. Hierzu sind in der Spezifikation Fehlerkorrektur Maßnahmen vorgesehen. Eine für diese Arbeit wichtige Eigenschaft des Bulk Transfers ist, dass er auf einem Bus der keine anderen Pakete übermittelt die schnellste Übertragungsart darstellt.

4.3.6. Interrupt Transfer

Wenn durch die Anwendung die Anforderung besteht, dass Informationen mit einer möglichst kleinen Latenz übertragen werden sollen, eignen sich Interrupt Transfers am besten. Ebenso wie der Bulk Transfer verwendet auch der Interrupt Transfer eine Fehlerkorrektur. Geräte welche diese Übertragungsart nutzen sind beispielsweise Tastaturen oder Mäuse.

4.3.7. Isochronous Transfer

Der Isochrone Transfer wird verwendet, wenn eine maximale Übertragungszeit von Paketen garantiert werden muss. Diese Garantieaussage wird über das Wegfallen der Fehlerkorrekturmaßnamen erkauft, was voraussetzt, dass auftretende Übertragungsfehler keine Probleme verursachen. Man kann die Eigenschaften der Datenübertragung mit denen einer UDP-Verbindung vergleichen. In der USB Implementierung spiegelt sich diese Eigenschaft z.B. darin wieder, dass kein Handshake-Paket übertragen wird um den Erhalt der Daten zu quittieren. Mögliche Verwendung sind z.B. Audio oder Video-Streams. Für Interrupt- und Isochrone Transfers werden gemeinsam 90% bis Full Speed (12 Mbps) (respektive 80% ab Hi-Speed(480 Mbps)) von der Bandbreite reserviert.

4.3.8. Aufteilung der Bandbreite

Wie aus der obigen Aufstellung ersichtlich, wird für Control, Interrupt und Isochrone Transfers die komplette Bandbreite reserviert. Für Bulk-Transfers steht an Bandbreite nur zur Verfügung, was nicht anderweitig benötigt wird.

4.4. Android Open Accessory Protocol

Das Android Open Accessory (AOA) Protocol ist seit der Version 3.1 des Betriebssystems Bestandteil von Android. Es ermöglicht Android Geräten, die keinen eigenen USB-Hostcontroller besitzen mit anderen Geräten zu kommunizieren. Hierzu übernimmt das „Accessory“ genannte Gerät das mit dem Android Gerät verbunden wird, die zur USB-Kommunikation notwendige Hostfunktionalität. Daraus ergibt sich der Vorteil, dass das Android Gerät nicht die Energie für das angeschlossene Gerät bereitstellen muss. Für das AOA-Protokoll entwickelte Accessories sind dadurch außerdem mit einer höheren Anzahl an Geräten kompatibel, da kein Hostmodus auf Seiten des Android Gerätes notwendig ist. Ein Gerät welches das AOA Protocol in der Version 1.0 unterstützt, stellt neben einem von der USB Spezifikation geforderten Control-Endpoint auf Adresse 0 weiterhin zwei Bulk-Endpoints zur Verfügung. Für jede Übertragungsrichtung gibt es einen Bulk-Endpoint. Die groben Schritte die ein Accessory ausführen muss um die Kommunikation zu einem Androidgerät herzustellen sind:

1) Warten bis ein Gerät verbunden wurde
2) Herausfinden ob sich das Gerät bereits im AOA-Modus befindet
3) Wenn nötig das Verbundene Gerät auffordern in den AOA-Modus zu wechseln
4) Kommunikation mit dem Gerät starten, wenn es das AOA-Protokoll unterstützt

Dieses Vorgehen wird aus dem folgenden Programmablaufplan in Abbildung 4.3 ersichtlich.

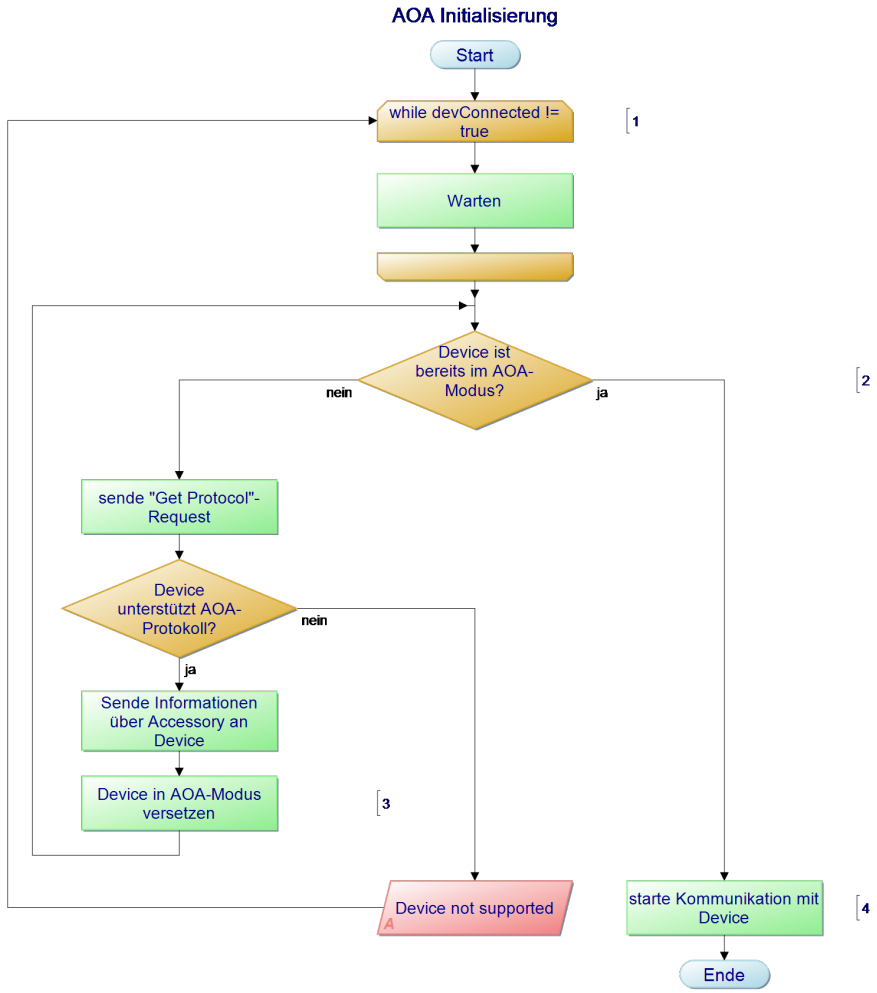


Abbildung 4.3.: Pogrammablaufplan zur AOA Intitialisierung

4.4.1. Kompatibilität prüfen

Um festzustellen ob ein angeschlossenes Gerät den AOA Modus unterstützt müssen die im USB Device Descriptor übertragenen Eigenschaften überprüft werden. Die Vendor-ID (VID) eines unterstützten Gerätes sollte der VID 0x18D1 von Google entsprechen. Gleichzeitig sollte als Product-ID (PID) 0x2D00 oder 0x2D01 angegeben sein, welche Geräte melden wenn sie sich bereits im AOA Modus befinden. In diesem Fall kann direkt die Kommunikation mit dem Gerät beginnen und die nächsten Schritte können übersprungen werden. Sollten PID und/oder VID nicht den erwarteten Werten entsprechen, kann nicht direkt entschieden werden

ob das angeschlossene Gerät den AOA-Modus unterstützt. Es muss zunächst versucht werden das Android Gerät anzuweisen sich in den AOA-Modus zu versetzen. Dazu wird ein Control Request an das Gerät übertragen. Wie aus der Android Dokumentation ersichtlich muss dieser, wie in Listing 4.1 gezeigt, aufgebaut sein (vgl. [13]).

```
requestType:    USB_DIR_IN | USB_TYPE_VENDOR
request:        51
value:          0
index:          0
data:           protocol version number (16 bits little endian sent
                from the device to the accessory)
```

Listing 4.1: Control-Request zum anfordern der der unterstützten AOA-Version (Quelle: [13])

Der Request Typ wird hierbei aus USB_DIR_IN und USB_TYPE_VENDOR zusammengesetzt. Diese Masken repräsentieren Werte welche durch die USB-Spezifikation vorgegeben werden. Durch USB_DIR_IN wird definiert, dass der Host Daten empfangen möchte. USB_TYPE_VENDOR beschreibt, dass es sich um einen Herstellerspezifischen Request handelt. Der eigentliche Request wird durch die Nummer 51 angegeben, dessen Bedeutung vom Hersteller festgelegt wird und bei einem Android-Gerät für die „Get Protocol“-Anweisung steht. Bei der Datenübertragungsphase des Control Transfer wird dann eine 16-Bit Versionsnummer in Little-Endian Bytereihenfolge an den Host übermittelt (vgl. [13]). Sollte das Gerät diese Anfrage nicht unterstützen, z.B. weil es sich um kein Android Gerät handelt, wird das dem Host mit einem STALL-Handshake mitgeteilt (vgl. [21]).

4.4.2. Gerät in AOA-Modus versetzen

Wenn die vom Host erhaltene Protokollversionsnummer gültig ist, muss der Host dem Device Informationen zur Identifikation übertragen. Der Aufbau des zugehörigen Control-Requests:

```
requestType:    USB_DIR_OUT | USB_TYPE_VENDOR
request:        52
value:          0
index:          string ID
data:           zero terminated UTF8 string sent from accessory to
                device
```

Listing 4.2: Control-Request zum senden der Accessory Informationen

Wie aus dem Request Typ ersichtlich, werden vom Host diesmal Daten in Richtung des Gerätes angekündigt. Es handelt sich wieder um einen herstellerspezifischen Request. Als

Index muss eine ID angegeben werden, welche einen der folgenden unterstützten Parameter Identifiziert:

```
manufacturer name:  0
model name:         1
description:        2
version:            3
URI:                4
serial number:      5
```

Listing 4.3: Werte die der Index-Parameter des unter 4.2 gezeigten Requests annehmen kann

Durch diese mit UTF-8 (Universal Character Set Transformation Format, 8-Bit) kodierten String-Parameter ist es dem Android-Gerät möglich, zum Accessory passende Anwendungen zu finden. Wenn keine Anwendung vorhanden ist, kann dem Benutzer durch diese Informationen außerdem erleichtert werden eine passende Anwendung zu installieren. Dazu ist besonders die Möglichkeit einen URI (Uniform Resource Identifier) zu übertragen interessant. Darüber kann z. B. ein Weblink übertragen werden der dem Nutzer vom Betriebssystem direkt präsentiert wird um diesen auf eine Webseite zu leiten wo er das passende Anwendungspaket zum Accessory finden kann. Nach dem Übertragen der verschiedenen Strings zur Identifizierung des Accessorys, wird erneut ein Control-Request an das Android-Gerät gesendet (siehe Listing 4.4).

```
requestType:    USB_DIR_OUT | USB_TYPE_VENDOR
request:        53
value:          0
index:          0
data:           none
```

Listing 4.4: Control-Request zum aktivieren des AOA-Modus

Dieser veranlasst das Gerät dazu, sich in den Accessery-Modus zu versetzen. Das Gerät soll dementsprechend die USB-Verbindung mit dem Host neu aufbauen und sich diesmal als AOA-Kompatibles Gerät zu erkennen geben.

4.4.3. Kommunikation beginnen

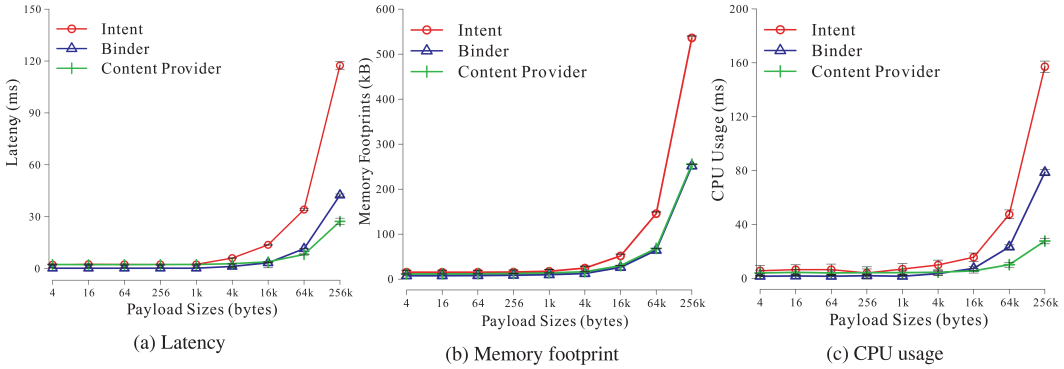
Sobald sichergestellt ist, dass ein kompatibles Gerät verbunden wurde, kann der Host die Informationen aus den Device und Endpoint -Deskriptoren verwenden um die Kommunikation mit den verfügbaren Endpoints zu starten. Wie oben genannt, werden Geräte mit zwei unterschiedlichen PIDs unterstützt. Ein Gerät mit der PID 0x2D00 stellt ein Interface mit zwei Bulk-Endpoints zur Verfügung, jeweils einen pro Übertragungsrichtung. Bei einem Gerät mit

der PID 0x2D01 wird ein zweites Interface bereitgestellt, welches wie bei der ersten Variante über einen Bulk-Endpoint pro Richtung verfügt. Dieses Interface wird allerdings speziell für ADB- (Android Debug Bridge) Kommunikation genutzt und hier nicht benötigt. Um jetzt die Kommunikation zu starten wird ein „Set Configuration"-Request vom Host gesendet, dieser wird nicht durch das AOA-Protokoll sondern im USB-Standard spezifiziert. Bei diesem Request wird als Wert die Nummer der Konfiguration übergeben die das Device einstellen soll. Im Fall des ersten Interfaces zur einfachen Kommunikation wird also eine 1 eingetragen. Nach diesem Request sind die Endpoints eingerichtet, und es können Daten an sie übertragen werden (vgl. [13] und [21]).

4.5. Performanz von IPC-Mechanismen unter Android

Um später ein geeignetes Verfahren für IPC innerhalb des Androidsystems auswählen zu können, müssen die verschiedenen Möglichkeiten zunächst miteinander verglichen werden. In dem Artikel „Performance Evaluation of Android IPC for Continuous Sensing Applications" [24] werden für den Einsatz in Anwendungen zur permanenten Sensordatenerfassung die Eigenschaften der Android eigenen IPC-Verfahren bezüglich auftretender Latenzen, effizienter Nutzung des Arbeitsspeichers und ihrer CPU-Auslastung verglichen. Schwerpunkt des Artikels sind Anwendungen die Datenraten zwischen 32 Byte/Sekunde bis 232 KB/Sekunde erfordern. Verglichen werden Binder, Intent, und Content Provider. Zur Messung wurde ein Benchmark entwickelt der die Daten über die genannten Mechanismen von einem Sende-Prozess zu einem Empfangs-Prozess transferiert. Ausgeführt wurde der Versuch auf einem Samsung SGH-T959 Galaxy S (1GHz ARM Cortex-A8 CPU, 512MB RAM) unter Android 2.2.

Das Ergebnis zeigt, dass die Binder-Variante in Bezug auf Latenz am besten von den Dreien abschneidet. So werden sehr kleine Datenmengen von 16 Bytes mit einer um den Faktor 12 kürzeren Latenz im Vergleich zu einem Intent übertragen (siehe Abbildung 4.4). Ein Effekt der außerdem stark hervortritt ist, dass sich die Lantenz ab einer Größe von 4KB signifikant erhöht. Diesen Effekt erklären die Autoren mit der initialen Größe des Kernel Puffers und der beim Überschreiten dieser Grenze notwendigen Speicherallokation. Im Speicherverbrauch liegen Content Provider und Binder gleichauf, während der Intent mit der doppelten Menge an benötigtem RAM wesentlich schlechter abschneidet. Ein ähnliches Bild zeigt sich auch bei der CPU-Auslastung. Hier ist der Binder bei Datenmengen unter 4KB stets die beste Wahl. So ist bei einer Größe von 64 Bytes die CPU-Auslastung beim Binder um den Faktor 2,5 kleiner als beim Content Provider, im Vergleich zum Intent um den Faktor 4. Über 4KB zeigt sich wie



(a) Latency (b) Memory footprint (c) CPU usage

Abbildung 4.4.: (a)Latenz, (b)Speicherverbrauch, und (c) CPU-Auslastung verschiedener IPC Mechanismen für variabel große Nutzdaten (Quelle: [24])

bereits bei der Latenzmessung der Effekt, dass auch die CPU-Auslastung signifikant ansteigt. So ist die CPU-Auslastung bei 16KB doppelt so hoch wie bei 4KB (vgl. [24]).

Die Autoren weisen darauf hin, dass bei den Messungen des Content Providers die CPU-Zeit für das Freigeben des allokierten Speichers nicht einbezogen werden kann, da dies erst durch die garbage-collection erledigt wird. Beim Vergleich der CPU-Auslastung muss dies beachtet werden, da ein augenscheinlich temporärer Vorteil in Bezug auf die Gesamtleistung des Systems nicht zwangsweise einen Nutzen bringt.

Für weiterführende Informationen zu Binder unter Android sei auf die Quellen [14] und [15] verwiesen. Eine genauere Analyse der vom Binder intern verwendeten Mechanismen wird in der Seminararbeit unter [27] gezeigt.

4.6. Protocol Buffers

Protocol Buffers werden von Google entwickelt und stellen eine effiziente Form der binären Serialisierung von Daten bereit. Offiziell werden Java, C++ und Python als Zielplattformen angegeben. Es stehen aber für nahezu alle aktuellen Programmiersprachen unabhängige Bibliotheken zur Verfügung. Zusätzlich bieten Protocol Buffers grundlegende Funktionen um eigene RPC Verfahren zwischen unterschiedlichen Plattformen zu realisieren. In der Arbeit „RPC for Embedded Systems“ [23] wurde gezeigt, dass sich zur Ansteuerung von Sensorik, ein mit den Protocol Buffers realisiertes RPC verfahren umsetzen lässt. Dazu wurde mit protobuf-c [1] eine C-Implementierung von Protocol Buffers verwendet. Das Ergebnis der Arbeit zeigt,

[1] https://github.com/protobuf-c/protobuf-c

dass sich Protocol Buffers auch auf kleinen eingebetteten Systemen einsetzen lassen. Die Einbindung neu definierter RPC Methoden wurde durch die Verwendung von protobuf-c und den sprachabhängige Eigenschaften von C als recht umständlich beschrieben. Eine direkte Verwendung der C++ Variante würde dieses Problem lösen. Ein Versuch hat allerdings ergeben, dass sich im Gegensatz zu protobuf-c die von Google entwickelte C++ Variante aufgrund ihrer Größe nicht auf dem DISCOVERY-Board einsetzen lässt. Da eine reine Serialisierung für strukturierte Daten zur Interprozesskommunikation zwischen unterschiedlichen Plattformen allerdings häufigen Einsatz finden dürfte, soll mit nanoPB [2] eine besonders kleine Protocol Buffers Implementierung in das Projekt integriert werden. Für die Funktion des Frameworks ist diese nicht notwendig, sondern stellt lediglich eine zusätzliche Komfortfunktion bereit.

4.7. Existierende Projekte

Durch die vielseitigen Einsatzmöglichkeiten von Mobilgeräten mit gekoppelten MCUs existieren bereits einige Projekte in welchen Systeme mit ähnlichen Funktionen realisiert wurden. Diese sollen in diesem Abschnitt vorgestellt werden. Es soll außerdem überprüft werden ob sich existierende Projekte oder Teile davon in dieser Arbeit wiederverwenden lassen.

4.7.1. STM32F4_ADK und HelloADK

Ein Projekt in dem das AOA-Protokoll umgesetzt wurde ist das in zwei Teilen verfügbare STM32F4_ADK [3] und HelloADK [4] von Yuuichi Akagawa. Der Status des Projektes wird auf der GitHub-Seite als experimentell deklariert und es existiert keinerlei Dokumentation. Der Quellcode umfasst dabei eine Proof-of-Concept implementierung des AOA-Protokolls als Android App und ein Programm für das DISCOVERY-Board. Die Android App in Form einer installierbaren .apk-Datei ließ sich mit einer aktuellen Version der Android Developer Tools ohne Probleme erstellen, auf einem Android-Gerät installieren und starten. Zum Kompilieren des Programms für das DISCOVERY-Board verweist der Autor auf die „Atollic TrueSTUDIO/STM32 Lite“-IDE. Da die kostenfreie Lite-Version dieser IDE eine Beschränkung der Programmgröße auf 32KB mit sich bringt[5] und daher für diese Thesis nicht verwendet wird, wurde das C-Programm in ein CooIDE-Projekt eingebunden. Unter Verwendung dieser IDE und einer aktuellen gcc-Version war aufgrund von Kompilierungsfehlern ein Test nicht ohne weiteres durchzuführen. Nach Beseitigung der Fehlerursachen, die vermutlich hauptsächlich auf die

[2] http://koti.kapsi.fi/jpa/nanopb/
[3] https://github.com/YuuichiAkagawa/STM32F4_ADK
[4] https://github.com/YuuichiAkagawa/HelloADK
[5] http://atollic.com/index.php/download/truestudio-for-arm

gewechselte IDE zurückzuführen sind, konnte die Funktion der Software bestätigt werden. Der Umfang der Software ist minimal gehalten und beschränkt sich auf das Schalten einer LED und dem Abfragen des USER-Buttons des Entwicklerboards. Dazu werden einzelne Bytes übertragen die den gewünschten Status der LED oder des Buttons beinhalten. Als Basis für die Kommunikation wird die von STMicroelectronics bereitgestellte USB-Bibliothek verwendet. Zur Implementierung des AOA-Protokolls wird ein Zustandsautomat verwendet welcher die unter Abschnitt 4.4 erläuterten Schritte zur Initialisierung des Android-Gerätes durchführt. Des weiteren stehen Methoden zum Lesen und Schreiben von Daten für die Bulk-Endpoints zur Verfügung. Dieser Teil der Anwendung wird mit einige Anpassungen und Erweiterungen als wiederverwendbar eingestuft. Die Android App dieser Beispielanwendung lässt sich nicht weiterverwenden da diese scheinbar nur zu Demonstrationszwecken und dadurch sehr unflexibel entworfen wurde.

Durch dieses Projekt wird allerdings gezeigt, dass die Kommunikation zwischen MCU und Android mithilfe der USB-Bibliothek des Herstellers realisiert werden kann und der Einsatz des AOA-Protokolls auf dem DISCOVERY-Board möglich ist.

4.7.2. Android ADK application for STM32 - RS232 client for Android

Ein Report-Dokument, das im Rahmen des „master course of Embedded Systems“ an der Politecnico di Milano Universität erarbeitet wurde, zeigt ein erweitertes Projekt auf Basis der oben genannten Grundimplementierung unter Verwendung der selben Hardware. Das Projektziel hierbei war es, ein Accessory zu entwickeln mit dem sich RS232-Schnittstellen des DISCOVERY-Boards von einer Android App aus konfigurieren und verwenden lassen. Da bei diesem Projekt das SSTM32F4_ADKProjekt verwendet wurde, wurde dieses ebenfalls analysiert und bewertet (vergl. [22]). Das Ergebnis ist dabei mit den oben genannten Aspekten konform und bewertet die grundlegende AOA-Implementierung für das DISCOVERY-Board unter Einschränkungen als wiederverwendbar und die Android App als ungeeignet in Bezug auf ihre Architektur.

Der Report lässt erkennen, das zur Umsetzung der Fokus stärker auf die Modularität der resultierenden Anwendung gelegt wurde. So wurde zur Abwicklung des AOA-Protokolls ein Service erstellt mit dem Activities kommunizieren können. Bei der Auswahl des Kommunikationsmechanismus zwischen Service und Activity wurde nur die Messenger-Variante in Betracht gezogen mit der Begründung, dies sei die einzige Möglichkeit zur Bidirektionalen Kommunikation. Diese Aussage kann nicht Bestätigt werden, da auch mit einer AIDL-Schnittstelle, direkter Verwendung der Binder-Funktionalität oder Intents eine solche bidirektionale Kommunikation realisiert werden kann. Eine weitere Schwachstelle ist die vom Entwickler vermerkte Ein-

schränkung der Anwendung, nur lesbare Zeichen übertragen zu können (vergl. [22]). Diverse Probleme mit der SSTM32F4_ADK"-Implementierung wurden hier bereits erkannt und in der Weiterführung berücksichtigt. Leider ist zu diesem Projekt der Quellcode nicht verfügbar und eine E-Mail Anfrage diesbezüglich an den Autor und den Prüfer blieb unbeantwortet.

4.7.3. IOIO

Das IOIO Projekt [6] beschäftigt sich ebenfalls mit der Kombination von Android Geräten und einer MCU. Der Schwerpunkt liegt dabei in der Ansteuerung der Hardwareschnittstellen auf dem eigens entwickelten Board, das ab ca. 30$ erhältlich ist. Damit können Beispielsweise PWM Signale aus einer Android App heraus erzeugt, oder ein Gerät über SPI angesprochen werden. Durch die Spezialisierung auf Hardwareschnittstellen und die Festlegung auf das IOIO-Board wird dieses Projekt nicht weiter betrachtet. Der Erfolg dieses kommerziell vertriebenen Produktes zeigt aber deutlich die Relevanz des Themas, Android-Geräte mit MCUs zu koppeln.

[6] https://github.com/ytai/ioio/wiki

5. Konzept

Um die anfangs genannten Anforderungen und die Ergebnisse der Analyse bestmöglich in der Lösung zu berücksichtigen, wird im Folgenden ein Konzept erarbeitet. Das Konzept wird nach dem Top-Down Prinzip erstellt. Dazu wird als erstes eine sinnvolle Struktur für das gesamte Projekt festgelegt, um zu definieren an welcher Stelle bestimmte Funktionalitäten implementiert werden sollen. Im Anschluss werden die einzelnen Komponenten und deren Funktionen genau beschrieben und in die definierte Struktur eingeordnet. Auf Basis dieses Konzeptes soll daraufhin die Implementierung des Frameworks stattfinden.

5.1. Logische Struktur

Der Aufbau des Systems bezogen auf die Hardware kann vereinfacht durch Abbildung 5.1 veranschaulicht werden. Es sind die beiden kommunizierenden Geräte und die Verbindung beider mittels einer Datenübertragungskomponente erkennbar. Aus diesem physikalisch orientierten Aufbau soll im Folgenden die logische Struktur des Frameworks abgeleitet werden. Das Framework soll auf zwei grundsätzlich andersartigen Systemen eingesetzt werden. Allgemein wäre in diesem Fall eine plattformunabhängige Implementierung von Vorteil, um Großteile des Systems nur einmalig umsetzen zu müssen. Unter Android wird zur Entwicklung hauptsächlich Java verwendet. Eine weitere Möglichkeit bietet das Android NDK, mit dem native C/C++ Anwendungen entwickelt werden können. Auf der Android Developer Referenz wird zum Vergleich beider beschrieben: „Notably, using native code on Android generally does not result in a noticable performance improvement, but it always increases your app complexity.“ [10]. Eine leichte Steigerung der Performanz steht also einer erhöhten Komplexität gegenüber. Durch die Verwendung von Nativem Code auf dem Android-Gerät wird außerdem die Plattformunabhängigkeit zwischen Android-Geräten mit verschiedener Hardwarearchitektur erschwert (vgl. [11]). Der Einsatz von Java auf der MCU wird durch die Vorgabe hierbei C/C++ zu verwenden ausgeschlossen. Hinzu kommt das unterschiedliche Laufzeitverhalten zwischen dem Android-Gerät mit, und der MCU ohne Betriebssystem und

der erhebliche Unterschied an verfügbaren Ressourcen. Durch die genannten Gründe werden zwei einzeln zu implementierende Teilsysteme als geeigneter erachtet.

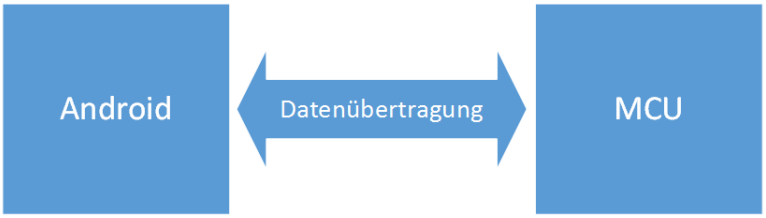


Abbildung 5.1.: Grober Aufbau der Hardware

Um diesen sehr allgemein gefassten Aufbau nun zu einem konkreten Konzept zur Umsetzung zu erweitern sind einige Designentscheidungen abzuwägen und festzuhalten. Unter Kapitel 3 wurde bereits erwähnt dass die Architektur einen modularen Ansatz verfolgen soll. Ein Ziel ist nun die Einschätzung, welche Art von Modulen erstellt werden und welche Funktionalitäten diese jeweils beinhalten sollen. Dazu ist es zunächst sinnvoll den strukturellen Aufbau des Kommunikationssystems detaillierter zu betrachten. Unter Abbildung 5.2 ist ein Kommunikationsmodell zu sehen das die Datenübertragung in einer Richtung darstellt. Das Quell- oder Zielsystem als Einheit, können verglichen mit Abbildung 5.1 die MCU oder das Android-Gerät repräsentieren. Die Quelle und das Ziel selbst, sollen später die Prozesse auf den jeweiligen Systemen repräsentieren.

Der Blick auf das Modell soll dabei speziell auf die Sende- und Empfangseinheit gerichtet werden. In diesem Modell hat der Sender die Aufgabe, die an ihn übergebenen Daten in eine Form zu wandeln, welche mit dem Übertragungssystem zum Empfänger transferiert werden kann. Der Empfänger dekodiert diese und übergibt sie an das Ziel (vgl. [28] S.44 ff.). Für die Struktur des Kommunikationssystems ist nun entscheidend auf welcher Ebene diese Funktionalitäten später umgesetzt und einer Anwendung zur Verfügung gestellt werden sollen. Im wesentlichen sind bei der Wahl der Integrationsebene zwei Varianten denkbar. Die Kommunikationsfunktionalität kann entweder als integrierbarer Teil direkt in die Anwendung eingefasst werden oder über einen eigenständigen Dienst der Anwendung zur Verfügung gestellt werden. Beide Varianten sollen im Folgenden verglichen werden.

5.1.1. Kommunikationsfunktionalität als Teil der Anwendung

Eine Möglichkeit besteht darin, sämtliche Komponenten die zur Kommunikation nötig sind z.B. als Bibliothek in die Anwendung einzubinden um die Prozesse damit direkt mit der Transport-

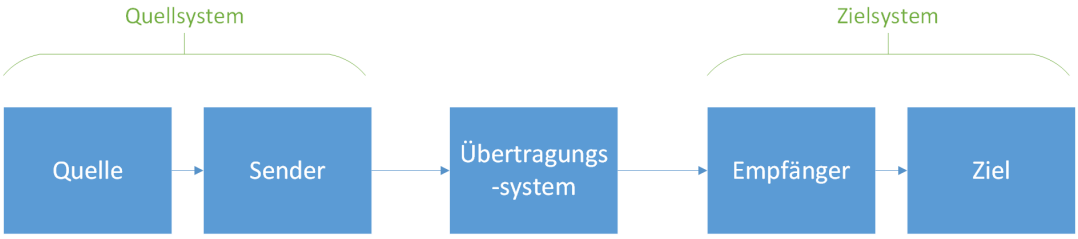


Abbildung 5.2.: Blockdiagramm eines einfachen Kommunikationsmodells (frei nach [28] S.44 Fig.1.4)

schicht zu verbinden. Unter Abbildung 5.3 ist eine solche Struktur dargestellt. Die Funktionen von Sender und Empfänger aus Abbildung 5.2 sind dabei in Form einer Bibliothek in jedem Prozess einmal vorhanden. Dieses Vorgehen bietet den Vorteil, dass die Datenübertragung direkt vom sendenden Prozess aus stattfinden kann. Somit können weitere lokale Kopiervorgänge der Daten z. B. in Form einer Übertragung an einen speziellen Sende-Prozess vermieden werden, was der Anforderung nach Performanz zugute kommt. Des weiteren ist die nicht-funktionale Anforderung nach einer effizienten Einbindung des Frameworks in eine Anwendung damit ohne Umwege zu erreichen, da nur Bibliothek in ein Projekt integriert werden muss.

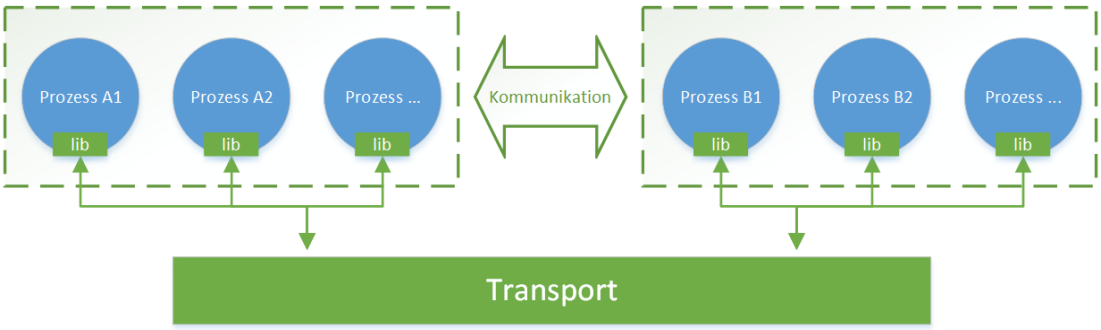


Abbildung 5.3.: Logische Struktur - Kommunikationsfunktion als Teil des Prozesses

Nachteile ergeben sich dabei u. a. im Hinblick auf Wartbarkeit, Kompatibilität, Flexibilität und Synchronisierung. So kann das Übertragungsprotokolls der genannten Bibliothek nicht ohne weiteres aktualisiert bzw. verändert werden, oder Veränderungen müssen dabei Abwärtskompatibilität sicherstellen, da sämtliche Details der Protokollimplementierung direkt auf Ebene der Anwendung zu finden sind. Anwendungen können zur Aktualisierung des Protokolls zwar theoretisch mit der aktuelleren Bibliothek neu kompiliert werden, was sich

aber bereits bei einer geringen Anzahl an Anwendungen durch den entstehenden Aufwand als unpraktikabel erweist.

Das Problem bei der Synchronisierung besteht darin, dass die Prozesse auf einem System ggf. nicht zeitgleich auf die Transportschicht zugreifen können und zur Koordinierung der Zugriffe in diesem Fall zusätzliche Synchronisierungsmechanismen zwischen den Prozessen nötig sind. Dieses Problem existiert hauptsächlich bei der Android Implementierung, da hierbei ein Betriebssystem eingesetzt wird das parallele Anwendungen erlaubt.

5.1.2. Kommunikationsfunktionalität als eigenständiger Dienst

Die andere Möglichkeit ist eine Architektur bei der das Übertragungsprotokoll und der Datentransport in einer von der Anwendung unabhängigen Komponente umgesetzt werden. Da diese Komponente einen Dienst für andere Anwendungen bzw. Prozesse zur Verfügung stellt, wird sie im weiteren auch „Dienst“ genannt. Das Übertragungsprotokoll wird hierbei nur zur Übertragung zwischen den Diensten genutzt und kann dadurch unabhängig von den Anwendungen verändert werden (in Abbildung 5.4 gefüllt, grün dargestellt). Um bei dieser Variante dennoch den Vorteil einer simplen Integration in ein eigenes Projekt zu erreichen, wird eine vom Protokoll unabhängige Komponente verwendet (in Abbildung 5.4 als SAP (Service Access Point) gekennzeichnet), die in die Anwendung eingebunden wird und ihrerseits mit dem Dienst kommuniziert. Die Problematik mit dem gleichzeitigen Zugriff auf die Transportschicht ist zwar auch hierbei zu lösen, lässt sich aber einfacher realisieren da passende Mechanismen direkt im gemeinsam genutzten Dienst platziert werden können, der in der anderen Variante nicht vorhanden ist.

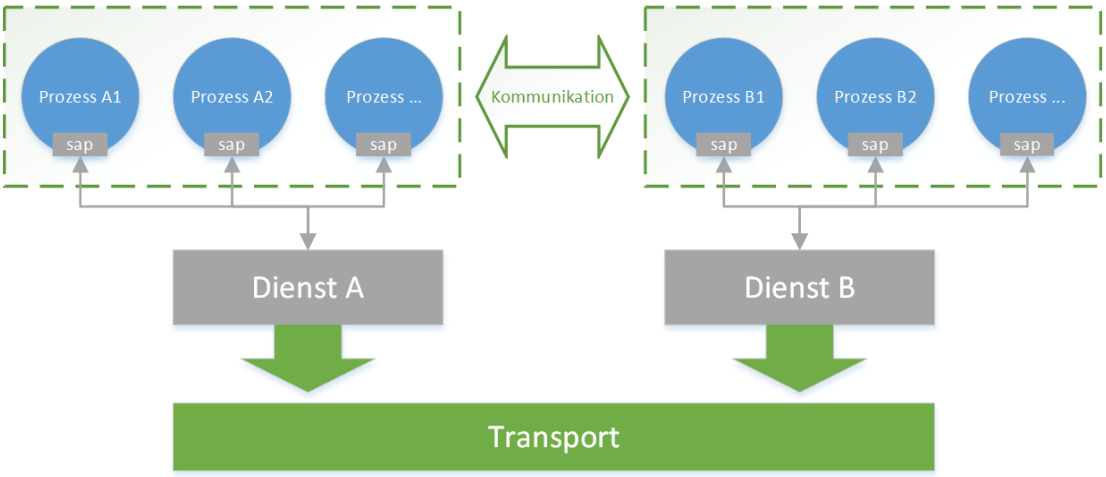


Abbildung 5.4.: Logische Struktur - Kommunikationsfunktion als Dienst

Aufgrund der oben genannten Vor- und Nachteile erscheint die Implementierung als eigenständiger Dienst zweckmäßiger.

5.2. Komponenten

Zur Umsetzung der definierten Struktur werden unabhängig von der Plattform auf beiden Seiten jeweils mehrere Komponenten benötigt. Um diese austauschbar zu halten, müssen allgemeine Schnittstellen zwischen ihnen definiert werden. Die einzelnen Komponenten und deren Schnittstellen werden in diesem Abschnitt festgelegt und deren Zusammenspiel untereinander dargelegt.

5.2.1. Prozesskomponente

Der Prozess ist in dieser Betrachtung immer entweder Quelle einer Nachricht oder deren Ziel. Wie unter Abschnitt 5.1 festgehalten, soll ein Prozess immer über den lokalen Dienst des eigenen Systems mit Prozessen auf dem anderen System kommunizieren. Um Zugriff auf die Funktionen des Frameworks zu erhalten, muss eine Komponente implementiert werden, die in den Prozess eingebunden werden kann. Die Schnittstelle der Prozesskomponente soll Daten und dazugehörige Attribute vom Dienst entgegennehmen können. Des weiteren sollen darüber Status und Konfigurationsparameter für die Prozess-Dienst-Verbindung übertragbar sein.

5.2.2. Dienst

Zusätzlich zu den Aufgaben der Sender- und Empfängerkomponente des Modells, die in Abbildung 5.2 eingeführt wurden, soll der Dienst die Verbindung und die Konfiguration des Frameworks zwischen den Systemen koordinieren. An ihn sollen Daten mithilfe angemessener lokaler Mechanismen übergeben werden, welche der Dienst dann wiederum an den korrespondierenden Dienst auf dem Zielsystem überträgt. Diese Mechanismen werden den Eigenschaften des jeweiligen Systems entsprechend gewählt. Auf der MCU läuft kein Betriebssystem und es gibt keinerlei Einschränkungen beim Zugriff auf den Speicher. Daher können Daten einfach als Zeiger zwischen Dienst und Prozess ausgetauscht werden. Dieses Vorgehen wird aufgrund der Effizienz präferiert, muss aber bei Einsatz eines Betriebssystems bedacht und ggf. angepasst werden. Auf dem Android-Gerät läuft ein Betriebssystem mit eigenem Speichermanagement und Zugriffseinschränkung auf den Prozesseigenen Speicherbereich. Hier gestaltet sich die Datenübertragung zwischen Dienst und Prozess durch die bessere Absicherung etwas aufwändiger. Aus den Verfahren die Android-Intern für Interprozesskommunikation zur Verfügung stehen, wird eine eigene über AIDL definierte Schnittstelle als geeignetes Mittel gewählt.

5.2.3. Transportschicht

Zusätzlich zu den zwei Diensten auf den beiden Plattformen ist eine wichtige Komponente die Datenübertragung. Bevor genauer auf die Aspekte der Transportschicht eingegangen wird, sollte zur Vermeidung von Verwechslungen definiert werden, dass sich der Begriff „Transportschicht“ nicht auf seine Bedeutung im Rahmen des OSI-Schichtenmodells (vgl. [20]) beschränkt. Vielmehr ist die Transportschicht im Kontext dieser Arbeit als eine logische Ebene zu verstehen, die durchaus Teile der Anwendungsschicht verwenden kann.

Eine der unter Kapitel 3 gestellten Anforderung an das System ist es, die Transportschicht flexibel austauschbar zu machen. Um dies zu erreichen ist eine klare Trennung zwischen Anwendungslogik und den Komponenten für den Datentransport nötig. Um eine möglichst einfache Integration einer anderen Transportschicht zu erreichen, sollte sie für die verwendende Anwendung Transparenz bezüglich des Zugriffs sicherstellen. Das bedeutet in diesem Fall, dass Unterschiede in der Art und Weise der Datenübertragung vor der Anwendung verborgen werden (vgl. [20]). Ein einfacher Weg dies einzufordern ist die Datenübertragung nur über eine vorab definierte Schnittstelle zu ermöglichen und die Details der Funktionsweise dahinter zu kapseln. Diese kann dann von jeder konkreten Variante einer Datenübertragungsschicht implementiert und von der Anwendung verwendet werden. Damit die Schnittstelle dem Entwickler dennoch wenig Flexibilität nimmt sollte es einfach gehalten werden und möglichst abstrakt

einen Kanal zur Datenübertragung darstellen um potentiell beliebige Übertragungsverfahren darin implementieren zu können. Da für dieses System der Schwerpunkt darauf liegt eine Kommunikation zwischen Prozessen auf zwei konkreten Geräten zu realisieren, wird davon ausgegangen, dass ein Datentransport immer zwischen zwei Knoten stattfindet auf welchen diverse Prozesse ausgeführt werden (vgl. Abbildung 5.4). Eine implementierte Transportschicht soll also immer ein direkter Tunnel zwischen zwei voneinander entfernten Diensten sein, die ihrerseits die Daten an die entsprechenden Prozesse übertragen. Eine Dienst-Komponente die einen solchen Tunnel verwendet, soll keinerlei Information darüber benötigen, wie dieser einzurichten ist. Vielmehr soll dem Dienst ein fertig konfigurierter Tunnel zur Verfügung gestellt werden, über welchen direkt Daten übertragen werden können. Hierdurch wird ein Dienst bereits implizit durch den verwendeten Tunnel an den gegenüberliegenden Dienst gekoppelt.

Abbildung 5.5 zeigt die unterschiedlichen Ebenen. Die Farben heben dabei die durch eine Schnittstelle voneinander getrennten Ebenen hervor, während die Markierung am Rand die Rolle im oben gezeigten Kommunikationsmodell aus Abbildung 5.2 verdeutlicht. Über die TransportEndpointInterface Schnittstellen wird eine konkrete Transportschicht, die im Bild als OpenAccessoryUSBTransport eingetragen ist angesprochen. Diese verwendet auf beiden Systemen wiederum die jeweiligen Bibliotheken. Unter Android steht dazu die Open Accessory Bibliothek und auf der MCU die ST USB Bibliothek mit weiteren Funktionen aus dem Hello ADK Projekt bereit. Beide kommunizieren über den gemeinsamen physikalischen Link miteinander.

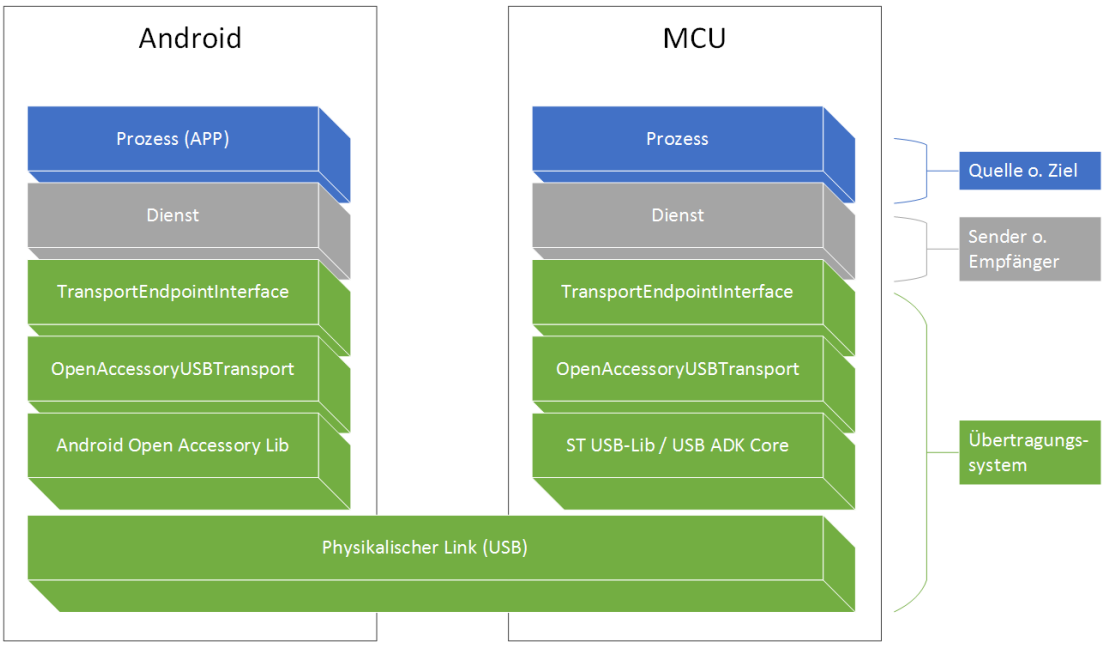


Abbildung 5.5.: Struktur des Frameworks mit Dienstkomponenten

5.3. Funktionen

In diesem Abschnitt wird auf die einzelnen logischen Funktionen des Frameworks eingegangen und deren konzeptionelle Umsetzung definiert.

5.3.1. Adressierung

Um Daten an den richtigen Prozess weiterleiten zu können ist mindestens eine einfache Form von Adressierung notwendig. Eine Adresse soll dabei einen Prozess unabhängig vom System eindeutig identifizieren. Dazu soll zunächst ein Integer-Wert ausreichen. Wie unter Abschnitt 3.1 erläutert, wird davon ausgegangen, dass sich Prozesse die kommunizieren wollen vorab bekannt sind. In der Praxis bedeutet das, dass einer Anwendung vor der Kompilierung eine feste ID zugeteilt wird. Weitere Verfahren zur ID-Vergabe sind zwar durchaus denkbar, sollen in dieser Thesis aber nicht weiter verfolgt werden. Aufgrund der Vergabe von IDs durch den Entwickler ist auch dieser dafür verantwortlich Konflikte durch eine mehrfache Vergabe zu verhindern. Das Framework soll dahingehend keine weiteren Überprüfungen durchführen.

Registrierung

Für bestimmte Einsatzzwecke ist es vorteilhaft, Nachrichten an eine Gruppe von interessierten Teilnehmern zu versenden. Damit in einem solchen Fall eine Alternative zur mehrfachen Übertragung der selben Nachricht an mehrere explizite Teilnehmer zur Verfügung steht, soll eine Broadcast Funktion implementiert werden. Damit soll eine Nachricht an alle interessierten Prozesse des gegenüberliegenden Systems übermittelt werden. Als Beispiel für eine solche Nachricht sei an dieser Stelle eine Statusnachricht mit dem aktuellen Akkustand genannt. Um die genannten interessierten Teilnehmer zu erfassen ist es nötig eine Registrierung zu erstellen bei der sie sich für bestimmte Nachrichten eintragen können. Unterschieden werden soll hierbei also nur durch den Typ der Nachricht.

5.3.2. Synchrone Nachrichten

Für synchron übertragene Nachrichten sind zusätzliche Mechanismen erforderlich. Der Dienst ist bei einer synchronen Nachricht nicht nur dafür zuständig sie an den richtigen Prozess weiterzugeben sondern auch dafür, dem sendenden Dienst die erfolgreiche Nachrichtenübermittlung an den Prozess zu quittieren. Um eine Nachricht eindeutig identifizieren und damit auch eindeutig bestätigen zu können, müssen synchrone Nachrichten mit einer ID versehen werden. Zu diesem Zweck muss ein eigener Nachrichtentyp erstellt werden den der Dienst selbst auswerten kann. Die Quittierung kann dann mit einem Verweis auf die Nachrichten-ID erfolgen. Wie unter Abschnitt 2.2 bereits erläutert, gibt es unterschiedliche Phasen bei der Nachrichtenübermittlung, die als eine erfolgreiche Nachrichtenzustellung aufgefasst werden können. Für die Lösung im Rahmen dieser Arbeit soll das beim RPC-Verfahren gängige Vorgehen genutzt werden, den Erhalt der Daten erst nach vollständiger Abarbeitung auf Seite des Empfängers zu bestätigen. Dieser Aufbau ist einfach anhand einer Receive-Methode vorstellbar bei der erst nach dem Rücksprung aus dem Methodenaufruf der Empfang bestätigt wird. Das bietet den Vorteil, dass je nach Anwendungsfall ohne zusätzlichen Aufwand eine andere Bestätigungssemantik implementiert werden kann. So mag es in einer Anwendung sinnvoll sein nur den reinen Erhalt der Daten zu bestätigen. In einer andern kann wiederum gewünscht sein erst bei erfolgreicher Verarbeitung der Daten die Bestätigung auszulösen. Es ist darauf zu achten, dass bei der Übertragung synchroner Nachrichten zunächst kein Time-out verwendet wird. Diese Funktion wird als optional betrachtet und an dieser Stelle als zukünftige Verbesserung genannt.

5.3.3. Serialisierung

Grundsätzlich wird davon ausgegangen, dass nur Rohdaten übertragen werden. Die Nutzdaten in einer Nachricht werden nicht interpretiert und müssen keiner bestimmten Norm entsprechen. Der Entwickler, der das Framework einsetzt, ist dafür zuständig übertragene Daten geeignet zu kodieren. Da bei der Kommunikation unterschiedlicher Systeme aber meist eine Aufgabe darin besteht die Daten für den Austausch geeignet zu serialisieren soll dahingehend eine gewisse Vorarbeit geleistet werden. Dazu sollen sämtliche Komponenten und das Protokoll dafür geeignet entworfen werden, Nachrichten im „Protocol Buffers“-Format (siehe [18]) zu übertragen. Dies soll erleichtern komplexere Nachrichten zu übertragen ohne für jeden Nachrichten-Typ händisch Funktionen zur Serialisierung zu erstellen. Insbesondere bei verschiedenen Computerarchitekturen und dem Einsatz diverser Programmiersprachen vermindert es den Implementierungsaufwand und stellt somit einen Mehrwert dar. Dies soll außerdem der Anforderung dienen, das Framework möglichst effizient in Projekte einbinden zu können.

5.4. Nachrichtenformat

Um alle geforderten Funktionen umsetzen zu können, müssen zur Übertragung einer Nachricht diverse Informationen übermittelt werden. Dazu gehört die Größe der Nachricht, die ID des Absenders und Empfängers, der Typ der Nachricht und ein Feld für Optionen. Das Optionsfeld soll z.B. dazu dienen ein Attribut für Synchrone Übertragung setzen zu können. Bei synchronen Nachrichten wird zusätzlich eine ID für die Nachricht übertragen, welche nachher zur Bestätigung dieser verwendet wird. Es soll außerdem Raum bieten, das Protokoll für spätere Funktionen erweiterbar zu halten. Damit erhält man im wesentlichen ein zum Ethernet-Frame nach IEEE-802.3 ähnlichen Aufbau, der allerdings kein Pendant zur Frame-Checksumme beinhaltet, da die Sicherstellung der korrekten Datenübertragung von der verwendeten Transportschicht eingefordert wird. Der Nachrichtenaufbau wird in Abbildung 5.6 und Abbildung 5.7 gezeigt. Die breiten Felder sollen mit zwei Byte, die schmalen mit einem Byte und der Payload mit einer maximalen theoretischen Größe von 2^{16} Byte abzüglich der jeweiligen Headergröße von 8 bzw. 10 Bytes dimensioniert werden. Das size Feld gibt immer die gesamte Nachrichtengröße inklusive Payload an.

Abbildung 5.6.: Nachrichtenaufbau - Asynchrone Nachricht

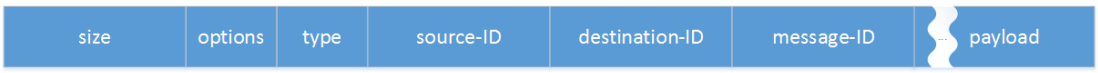


Abbildung 5.7.: Nachrichtenaufbau - Synchrone Nachricht

6. Implementierung

In diesem Kapitel wird die Implementierung der einzelnen Komponenten beschrieben. Dazu werden die beiden Varianten für Android und die MCU getrennt behandelt. Unterschiede beider Varianten werden aufgezeigt.

6.1. Strunktur

Die Anwendungsentwicklung für Android findet vollständig in einer Linux VM statt. Diese beinhaltet eine Eclipse Installation inklusive Android Development Tools (Version 22.6.2-1085508). Unter Eclipse werden zwei Projekte für das Framework angelegt:

AOAPService beinhaltet die Dienstkomponente und die Transportschicht.

AOAPClientLib dient als Bibliothek zum Einbinden des Service in einer Anwendung

Drei weitere Projekte beinhalten die Beispielanwendungen:

AOAPClientExample zur Nutzerinteraktion und Visualisierung.

AOAPClientTimesync zur Zeitsynchronisation.

AOAPClientSyncTX als Benchmark und Evaluierungsanwendung.

Die MCU-Implementierung findet in einem einzelnen Projekt namens STM32F4_ANDROID_IPC statt, da ohnehin nur eine einzelne Binärdatei auf die MCU übertragen wird.

6.2. Android

Im folgenden wird isoliert die Implementierung unter Android beschrieben.

6.2.1. Service

Der AOAPService stellt die zentrale Stelle des Kommunikationsframeworks auf der Seite der Android Plattform dar. Andere Anwendungen sollen sich mit dem Service über Prozessgrenzen hinweg verbinden, um über ihn Daten mit der MCU-Plattform auszutauschen. Der Service soll jederzeit für andere Anwendungen verfügbar sein, sofern eine Verbindung zur MCU-Plattform besteht. Unter Unterabschnitt 5.2.2 wurde definiert, dass die Kommunikation zwischen den lokalen Anwendungen und dem Dienst über eine AIDL Schnittstelle stattfinden soll.

AIDL Umsetzung

Die Dateien AOAPServiceIpcInterface.aidl und AOAPClientIpcInterface.aidl beinhalten die Schnittstellendefinitionen zwischen Service und Client-Anwendungen (siehe Abbildung 6.1).

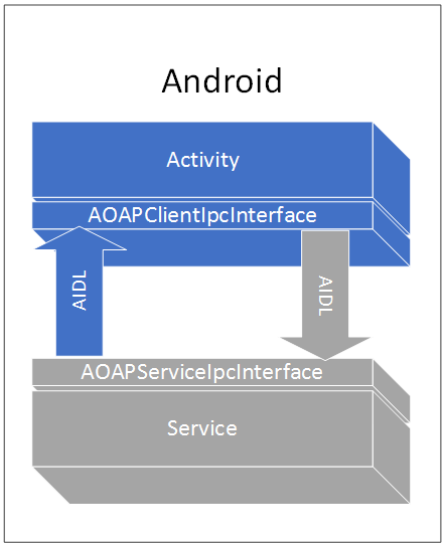


Abbildung 6.1.: Kommunikation über AIDL zwischen Client-Anwendung und Service unter Android

Verwendung des generierten Java Codes

Zur Verwendung des aus den AIDL Dateien generierten Codes, müssen Objekte implementiert werden, auf welchen später die Methodenaufrufe eines entfernten Prozesses ausgeführt werden. Die generierten Java Dateien enthalten dazu neben der reinen Schnittstellendefinition eine abstrakte Klasse namens Stub, welche den Code zum Serialisieren und Übertragen der Über-

gabeparameter beinhaltet. Von dieser wird eine Klasse abgeleitet welche die eingeforderten Methoden implementiert.

AOAPServiceIpcInterface

Das AOAPServiceIpcInterface (siehe Listing 6.1) dient als Schnittstelle zum Service. Über diese werden die Daten einer Nachricht als byte-Array mit den dazugehörigen Parametern wie der Sender-, Empfänger-ID und dem Nachrichtentyp an den Service übergeben. Im Service werden diese dann, nach dem unter Abschnitt 5.4 gezeigten Schema, an den Zielservice übertragen. Außerdem werden Methoden zum Registrieren einer Client-Anwendung und zum an- und abmelden für den Empfang bestimmter Nachrichtentypen deklariert. Beim Registrieren eines Clients über die Methode $registerClient()$ kann dem Service ein IBinder Objekt übergeben werden, mit dem er Zugriff auf die Funktionen der AOAPClientIpcInterface Schnittstelle der Anwendung erhält, diese wird im folgenden Absatz beschrieben. Da unter Android theoretisch zu jedem Zeitpunkt die Zielanwendung angehalten werden kann und das zugehörige Objekt somit nicht mehr erreichbar ist, kann von jeder Methode eine RemoteException geworfen werden.

```
1 public int transmitDataAsync(int src, int dst, int msgType, byte[]
    data) throws android.os.RemoteException;
2
3 public int transmitDataSync(int src, int dst, int msgType, byte[]
    data) throws android.os.RemoteException;
4
5 public int registerClient(android.os.IBinder clientStub) throws
    android.os.RemoteException;
6
7 public int subscribeForMessages(int clientID, int msgType) throws
    android.os.RemoteException;
8
9 public int unsubscribeForMessages(int clientID, int msgType) throws
    android.os.RemoteException;
```

Listing 6.1: AOAPServiceIpcInterface

Im AOAPService wird die Implementierung mit einer anonymen Klasse umgesetzt, die von der Stub-Klasse aus der AOAPServiceIpcInterface Datei abgeleitet ist. Das Objekt serviceIpcImpl stellt die einzige Instanz dieser Klasse dar und erhält alle Aufrufe in Richtung des Services.

AOAPClientIpcInterface

Das AOAPClientIpcInterface (siehe Listing 6.2) dient in umgekehrter Richtung als Schnittstelle der Anwendung, über die der Service empfangene Daten an sie übergibt. Des weiteren wird sie verwendet um die Beendigung des Service an die Anwendung zu melden. Diese kann sich daraufhin ebenfalls beenden oder beliebige andere Schritte ausführen. Nach Empfang dieser Rückmeldung sollten Anwendungen keine Versuche mehr unternehmen, Daten an den Service zu übertragen, da das entfernte Objekt ab diesem Moment in der Regel nicht weiter existiert. Die Methode $onClientRegistered()$ wird beim Verbindungsaufbau mit dem Service aufgerufen, sobald der Service eine Referenz auf das jeweilige AOAPClientIpcInterface-Objekt erhalten und vermerkt hat. Der damit verbundene Ablauf wird im nächsten Abschnitt genauer erläutert.

```
1 public int onReceiveData(int src, int type, byte[] data) throws
    android.os.RemoteException
2
3 public void onClientRegistered(int clientID) throws android.os.
    RemoteException
4
5 public void onServiceStopped() throws android.os.RemoteException
```

Listing 6.2: AOAPClientIpcInterface

6.2.2. ServiceBridge

Die im Unterabschnitt 5.2.1 beschriebene Prozesskomponente wird durch die Klassen ServiceBridge.java (siehe Listing 6.3) und eine im jeweiligen Projekt deklarierte Klasse auf Basis des AOAPClientIpcInterface realisiert. Die ServiceBridge ist dabei als Helfer-Klasse gedacht, welche den Verbindungsaufbau zum Service und den dazu verwendeten Handshake-Ablauf (siehe Abbildung 6.2) implementiert.

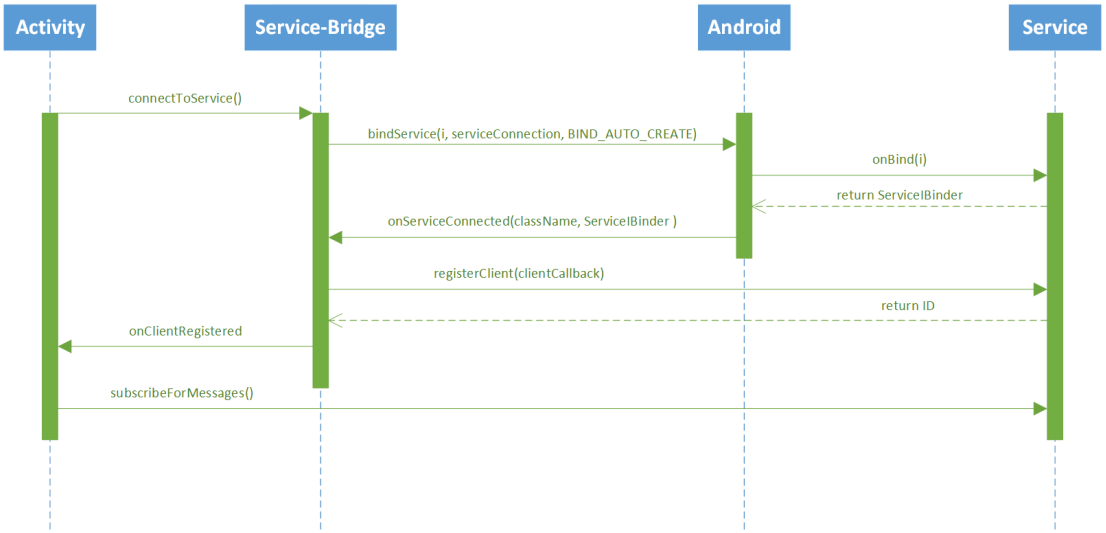


Abbildung 6.2.: Verbindungsaufbau mit dem Service ausgehend von einer Applikation die in einem separatem Anwendungspaket enthalten ist.

```
1 public boolean isBoundToService()
2 public int getClientID()
3 public void connectToService()
4 public void disconnectFromService()
5 public AOAPServiceIpcInterface getServiceStub()
```

Listing 6.3: Schnittstelle der ServiceBridge

Als Anwendungskomponente wird in diesem Fall eine Activity verwendet, die ServiceBridge lässt sich in gleicher Weise aber auch in einem Service verwenden. Von der Activity wird vorab ein ServiceBridge-Objekt erstellt, wobei diesem eine Referenz auf ein AOAPClientIpcInterface Objekt übergeben wird, das alle Aufrufe des Services erhalten soll. Um nun eine Verbindung mit dem Service herzustellen muss nur die Methode $connectToService()$ der ServiceBridge aufgerufen werden. Woraufhin diese die Activity automatisch über die jeweiligen Betriebssystemfunktionen an den Service Bindet. Zusätzlich wird dem Service eine Referenz auf das Zielobjekt übergeben, das die ServiceBridge bei der Initialisierung erhalten hat. Wenn dieser Vorgang abgeschlossen ist, erhält die Activity einen $onClientRegistered()$-Callback. Der Callback wird in diesem Beispiel direkt dazu genutzt, von der Service Bridge über $getServiceStub()$ eine Referenz auf den Service einzuholen und darüber den $subscribeForMessages()$-Aufruf zu tätigen, um fortan Bestimmte Nachrichten zu erhalten.

Für eine bessere Übersicht der zuvor genannten Komponenten zeigt Abbildung 6.3 den gesamten Aufbau anhand eines Klassendiagramms. Es ist anzumerken, dass zwecks der Übersicht nur die relevanten Attribute im Klassendiagramm aufgeführt werden.

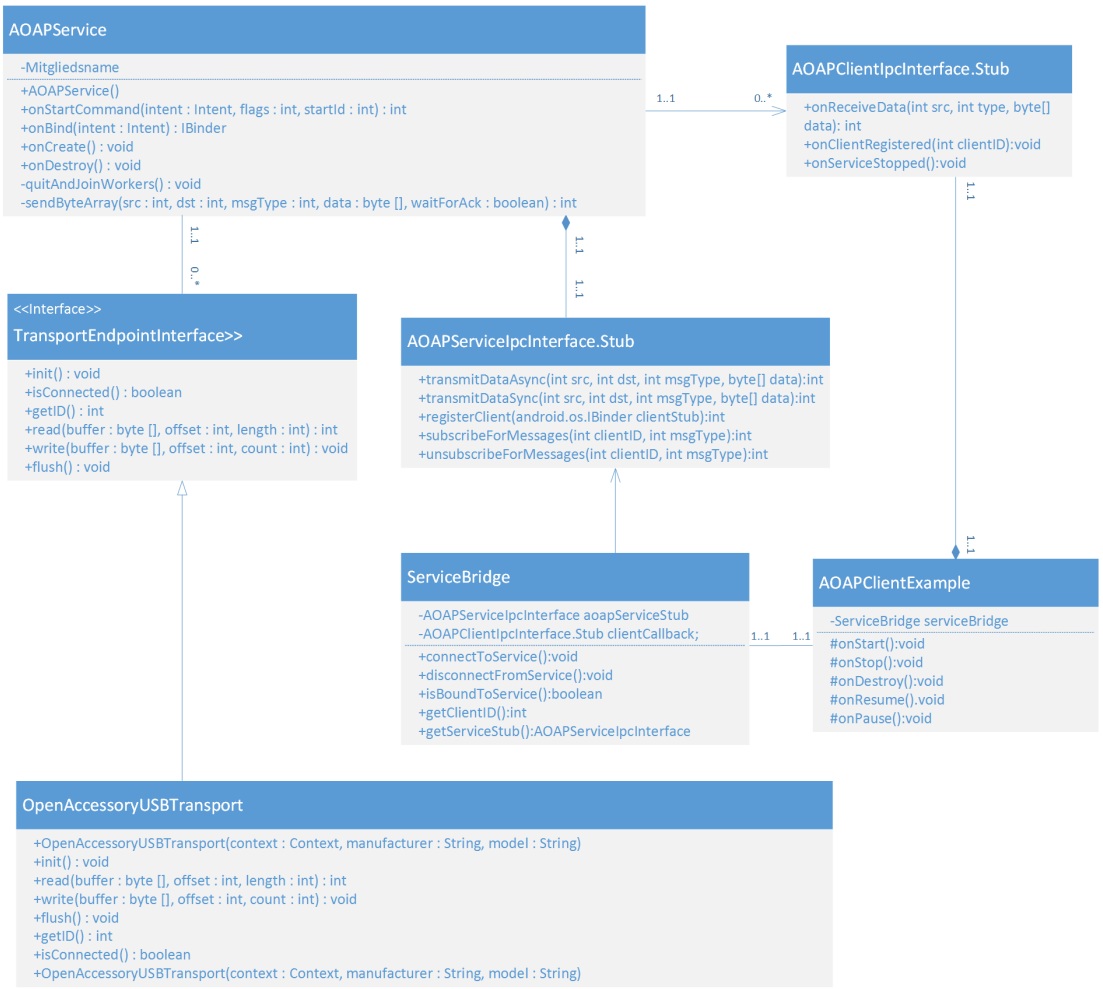


Abbildung 6.3.: Klassendiagramm der Implementierten Komponenten

Eigenschaften der Methodenaufrufe über AIDL

Für AIDL-Aufrufe werden intern die Binder Funktionalitäten verwendet. Dabei werden die Aufrufe an einen Threadpool innerhalb des Zielprozesses delegiert, der auch von diesem verwaltet wird. Trotz der Ausführung durch einen anderen Thread im Zielprozess, werden die Aufrufe synchron durchgeführt. Aufrufe über die AIDL-Schnittstelle können somit gleichzeitig stattfin-

den und die Methoden müssen thread-safe implementiert sein. Dieses Feature wird momentan nur für voneinander unabhängige Aktionen genutzt. So kann gleichzeitig eine Nachricht gesendet und eine Client-Anwendung registriert werden während die gleichzeitige Übertragung mehrerer Nachrichten nicht möglich ist. Wenn zwei Anwendungen gleichzeitig Nachrichten versenden wollen, wird einer der beiden Programmabläufe durch einen synchronized-Block blockiert. Die Entscheidung welcher Thread als nächstes mit dem Senden fortfahren kann, hängt damit von der Reihenfolge ab in der der Scheduler der JVM die Threads ausführt. Die Notwendigkeit der Blockierung liegt der seriellen Datenübertragung zu Grunde.

6.2.3. Service automatisch starten

Activities, Services und andere Komponenten können unter Android mit einem Intent (siehe Abschnitt 2.3.4) gestartet werden, der daraufhin an die jeweilige Komponente übergeben wird. Hierbei ist es wichtig zwischen expliziten und impliziten Intents zu unterscheiden.

Um eine Activity oder einen Service explizit zu starten sind der Klassenname und die Zugriffsrechte auf die Klasse der Komponente nötig. Durch das Berechtigungssystem von Android können diese Komponenten standardmäßig nur vom System oder anderen Komponenten innerhalb derselben Anwendung gestartet werden. Damit der Service aus einem anderen Anwendungspaket verwendet werden kann, muss das exported-Flag in der Manifest-Datei des Services auf true gesetzt werden (siehe Listing 6.4 Z. 28). Zusätzlich wird eine Berechtigung angelegt, welche fremden Anwendungen zur Verwendung des Services eingeräumt werden muss (siehe Listing 6.4 Z. 7).

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="de.haw.aes.aoap.service"
4      android:versionCode="1"
5      android:versionName="1.0" >
6
7      <permission android:name="de.haw.aes.aoap.COMMUNICATE_WITH_SERVICE"
           android:label="Zugriff auf den AOAPService"/>
8
9      <uses-sdk
10         android:minSdkVersion="11"
11         android:targetSdkVersion="19" />
12
13     <application
14         android:allowBackup="true"
15         android:icon="@drawable/ic_launcher"
```

```
16          android:label="@string/app_name"
17          android:theme="@style/AppTheme" >
18          <uses-library android:name="com.android.future.usb.accessory" />
19
20          <activity
21              android:name="de.haw.aes.aoap.service.MainActivity"
22              android:label="@string/app_name" >
23          </activity>
24
25          <service
26              android:permission= "de.haw.aes.aoap.COMMUNICATE_WITH_SERVICE"
27              android:name="de.haw.aes.aoap.service.AOAPService"
28              android:exported="true">
29          </service>
30
31          <activity
32              android:name="de.haw.aes.aoap.service.ServiceStarter"
33              android:label="@string/title_activity_service_starter"
34              android:theme="@android:style/Theme.NoDisplay"
35              android:noHistory="true"
36              android:excludeFromRecents="true">
37
38              <intent-filter>
39                  <action android:name="android.intent.action.MAIN" />
40                  <category android:name="android.intent.category.LAUNCHER" />
41              </intent-filter>
42
43              <intent-filter >
44                  <action android:name="android.hardware.usb.action.
                        USB_ACCESSORY_ATTACHED" />
45                  <category android:name="android.intent.category.DEFAULT" />
46              </intent-filter>
47
48              <meta-data
49                  android:name="android.hardware.usb.action.
    USB_ACCESSORY_ATTACHED"
50                  android:resource="@xml/usb_accessory_filter" />
51          </activity>
52      </application>
53  </manifest>
```

Listing 6.4: AndroidManifest.xml des Service

Um von einer Anwendung die Berechtigung einzufordern muss in der Manifest-Datei lediglich ein Verweis auf die selbe Berechtigung eingetragen werden (siehe Listing 6.5). Bei der Installation wird diese dann vom Benutzer abgefragt und das Betriebssystem stellt sicher, dass keine Anwendung ohne diese Berechtigung den Service starten oder sich daran binden kann.

```
1 <uses-permission android:name="de.haw.aes.aoap.COMMUNICATE_WITH_SERVICE"/>
```

Listing 6.5: AndroidManifest.xml der ClientLib (Auszug)

Implizite Intents werden für Aufgaben verwendet die potentiell von mehreren Anwendungen erledigt werden können, allerdings ohne vorab anzugeben welche konkrete Anwendung dafür verwendet werden soll. So kann ein Bild z. B. sowohl mit einem Betrachtungs-, als auch mit einem Bearbeitungsprogramm oder eine Webseite mit verschiedenen Browsern geöffnet werden.

Um eine Anwendung bei einem bestimmten Ereignis implizit zu starten, bietet Android sogenannte IntentFilter an, die auf Ereignisse im System reagieren können. Wenn ein USB-Accessory angeschlossen wurde, wird ein Intent mit dem Action-String „android.hardware.usb.action.USB_ACCESSORY_ATTACHED“ gesendet. Damit die Anwendung automatisch darauf reagiert, wird ein IntentFilter in die Manifest-Datei der zu startenden Anwendung eingetragen (siehe Listing 6.6).

```
1 <intent-filter>
2         <action android:name="android.hardware.usb.action.
              USB_ACCESSORY_ATTACHED" />
3 </intent-filter>
```

Listing 6.6: Intent Filter für angeschlossenes USB-Accessory

Bei dem Erstellen eines Intents gleicht das Betriebssystem ihn mit den Einträgen des Manifests jeder Anwendung ab und übergibt diesen situationsbedingt der einzig kompatiblen Anwendung, der festgelegten Standardanwendung oder präsentiert dem Nutzer eine Liste von geeigneten Anwendungen.

Da ein Service nicht zur direkten Interaktion mit dem Nutzer konzipiert ist und dieser in der Regel vom Nutzer verborgen arbeiten soll, werden sie im Anwendungsmenü nicht separat aufgeführt. Der Nutzer hat also auch keine Möglichkeit einen Service direkt als Standardanwendung für bestimmte Ereignisse festzulegen. Aus Konsistenz- und vor allem Sicherheitsgründen sollten Services daher keinesfalls über implizite Intents gestartet werden. [1]

[1] Der Benutzer hätte in diesem Fall keinerlei Einfluss darauf ob wirklich der gewünschte Service gestartet wird, oder ein „bösartiger“ der auf denselben impliziten Intent reagiert. (siehe http://developer.android.com/guide/components/intents-filters.html)

ServiceStarter

Um den Service dennoch über den Impliziten Intent zu starten, der ausgelöst wird sobald ein unterstütztes Gerät verbunden wurde, wird der IntentFilter nicht in den Service sondern in die AOAPServiceStarter-Activity eingebunden. Diese hat einzig die Aufgabe, den Service über den vollständigen Klassennamen explizit zu starten. Somit hat der Benutzer, der als Standardvorgang den Service-Starter auswählen kann, volle Kontrolle darüber was geschieht sobald ein Gerät verbunden wird und es ist gleichzeitig möglich den Service automatisch zu starten.

Der ServiceStarter hat außerdem die Aufgabe die Transportschicht zu initialisieren und sie an den Service weiterzureichen (siehe Listing 6.7). Da der ServiceStarter derzeit nur durch das Anschließen eines kompatiblen USB-Gerätes gestartet wird, kann bei dessen Ausführung davon ausgegangen werden, dass zur Kommunikation immer ein OpenAccessoryUSBTransport-Endpoint verwendet wird. Um einen anderen Transport-Endpoint zu verwenden, ist an dieser Stelle später eine Fallunterscheidung nötig. Die Activity ist außer einer kurzen Meldung beim Starten des Services unsichtbar und hat keine Bedienelemente. Sie wird direkt nach dem Starten des Services beendet.

```
1  @Override
2  protected void onCreate(Bundle savedInstanceState) {
3          super.onCreate(savedInstanceState);
4
5          TransportEndpointInterface aoa_usb_transport = new
               OpenAccessoryUSBTransport(getApplicationContext(), "HAW-
               Hamburg", "STM-32F4_FCU-Bridge");
6          aoa_usb_transport.init();
7          AOAPService.endpoints.put(aoa_usb_transport.getID(),
               aoa_usb_transport);
8
9          Toast.makeText(getApplicationContext(), "starting_Service_
               with_USB-Connection...", Toast.LENGTH_SHORT).show();
10         startService(new Intent(getApplicationContext(),AOAPService.
               class));
11         finish();
12 }
```

Listing 6.7: ServiceStarter Activity zum Starten des Service

Damit der Service nur gestartet wird wenn ein kompatibles Gerät verbunden wurde, werden über ein verknüpftes meta-data Objekt (siehe Listing 6.8) Eigenschaften vorgegeben, die

ein angeschlossenes Gerät besitzen muss. Diese Eigenschaften werden in die Datei res/xml/usb_accessory_filter.xml im AOAPService Projekt eingetragen, welche automatisch vom Betriebssystem mit den, von der MCU übertragenen Geräteeigenschaften (siehe Unterabschnitt 4.4.2) verglichen werden.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <usb-accessory manufacturer="HAW-Hamburg" model="STM-32F4_FCU-Bridge"
    version="1.0"/>
</resources>
```

Listing 6.8: usb_accessory_filter.xml

6.2.4. Transportschicht

Für die austauschbare Transportschicht wird das Interface TransportEndpointInterface.java erstellt (siehe Listing 6.9). Es deklariert die unter Java von Input- und OutputStream bekannten Funktionen $read()$, $write()$ und $flush()$, und stellt damit eine Stream-basierte Kommunikationsfunktion bereit. Die Methode $init()$ soll für einen Callback zur Verfügung stehen, der vom Service bei der Initialisierung ausgeführt wird. Darin können in einer konkreten Implementierung eines TransportEndpoints Aufgaben erledigt werden, die noch vor der ersten Kommunikation über die oben genanten Funktionen stattfinden sollen. Über getID() soll die ID des Systems zurückgegeben werden, zu dem der Endpoint eine Verbindung darstellt. Mit isConnected() wird der Verbindungsstatus abgefragt.

```
public interface TransportEndpointInterface{

        public void init();
        public boolean isConnected();
        public int getID();

        public int read(byte[] buffer, int offset, int length)
            throws IOException;
        public void write(byte[] buffer, int offset, int count)
            throws IOException;
        void flush() throws IOException;
}
```

Listing 6.9: TransportEndpointInterface.java

OpenAccessoryUSBTransport

Um die USB Kommunikation über das Android Open Accessory Protokoll zu verwenden, kann die Bibliothek über das Paket android.hardware.usb in ein Android Projekt eingebunden werden. Die USB accessory-APIs sind seit Android 3.1 Bestandteil von Android. Da die Kompatibilität mit möglichst vielen Geräten eine der Anforderungen an das System ist, wird die support-Bibliothek aus dem Paket android.future.usb der Google-APIs verwendet. Hierdurch werden zusätzlich ältere Betriebssystemversionen ab Version 2.3.4 unterstützt.

Die Implementierung der USB-Transportschicht ist in der Klasse OpenAccessoryUSBTransport enthalten. Damit der Verbindungsstatus in einem OpenAccessoryUSBTransport-Objekt beim Entfernen des Gerätes aktualisiert wird, wird in der $init()$ Methode zu beginn

```
1 new IntentFilter(UsbManager.ACTION_USB_ACCESSORY_DETACHED)
```

Listing 6.10: IntentFilter für Entfernen eines USB Gerätes

ein IntentFilter angelegt, der auf das Entfernen des USB-Gerätes reagiert. Mit diesem Filter wird ein BroadcastReceiver als Empfänger für entsprechende Intents registriert. Der BroadcastReceiver aktualisiert bei erhalten des Intents den Verbindungsstatus.

Zum Zugriff auf ein USB-Accessory werden mehrere Schritte benötigt. Mit dem Context-Handle der Anwendung erlangt man über UsbManager.getInstance(context) eine Instanz des UsbManagers, der über die Methode $getAccessoryList()$ eine Liste aller verbundenen Accessories zurückgibt. Ein UsbAccessory Objekt aus dieser Liste bietet getter-Methoden für die String-Attribute wie Hersteller und Modell eines verbundenen Gerätes an.

Über diese Eigenschaften wird überprüft ob ein Angeschlossenes Gerät kompatibel ist. Das erste kompatible Gerät wird verwendet. Nachdem überprüft wurde ob die notwendigen Berechtigungen für den Zugriff auf das Gerät bestehen, wird mit usbManager.openAccessory(usbAccessory) eine ParcelFileDescriptor Instanz eingeholt. Diese wird dazu verwendet einen normalen FileDescriptor zwischen Prozessen zu übertragen. Der eigentliche FileDescriptor kann über ParcelFileDescriptor.getFileDescriptor() erhalten werden. Mit diesem lassen sich abschließend mit new FileInputStream(FileDescriptor) und new FileOutputStream(FileDescriptor) Streams erstellen, welche direkten Lese- und Schreibzugriff auf die USB-Bulkendpoints bereitstellen, die mit der MCU in Verbindung stehen. Die Methoden $read()$, $write()$ und $flush()$ geben Aufrufe direkt an das jeweilige Stream-Objekt weiter.

6.2.5. AOAPClientLib

Um wiederverwendbaren Code direkt in eine Anwendung zu integrieren, stehen in der Android-IDE Library-Projekte zur Verfügung. Sie können wie normale Projekte erstellt und verwendet

werden, bieten aber im Zusammenspiel mit der IDE zusätzliche Funktionen um Berechtigungen und Kompatibilitätsanforderungen in der Zielanwendung einzufordern.

Für das Framework wird ein Library-Projekt namens AOAPClientLib angelegt. Dieses beinhaltet die Funktionalität der ServiceBridge und die Schnittstellendefinitionen AOAPServiceIpcInterface und AOAPClientIpcInterface. Beide werden im src Ordner der Bibliothek platziert, wodurch im gen Verzeichnis der Bibliothek automatisch die resultierenden Klassen für die Schnittstellen Erzeugt werden. Durch die Zentrale Speicherung der Schnittstellendefinitionen muss zur Sicherung der Kompatibilität zwischen Service und Anwendung nur darauf geachtet werden, dass die selbe Version der Bibliothek verwendet wird, und die Fehleranfälligkeit durch manuelles Kopieren entfällt.

Die Bibliothek ClientLib muss in jede Anwendung, die über den Service kommunizieren möchte und das Serviceprojekt selbst eingebunden werden. Dazu muss über die Projekteigenschaften unter $Android$->$Library$ die jeweilige Bibliothek als Referenz hinzugefügt werden [2]. Um auch die Berechtigungen aus dem Manifest der Bibliothek automatisch in das neue Projekt zu übernehmen sollte zusätzlich im Zielprojekt das Flag
manifestmerger.enabled=true
gesetzt werden. Wenn dieser Schritt nicht durchgeführt wird muss die Berechtigung zum verwenden des Services (siehe Listing 6.5) manuell in die Manifest-Datei eingetragen werden, da die Anwendung andernfalls nicht mit dem Service Kommunizieren kann.

6.3. MCU

6.3.1. IPCService MCU

Der Service auf MCU-Seite benötigt zur lokalen Datenübergabe keine zusätzlichen Mechanismen wodurch Client Anwendungen direkt auf einer Referenz des Serviceobjekts arbeiten können (siehe Abbildung 6.4). Die Schnittstelle der C++ Implementierung ist bis auf zusätzliche Parameter für die Länge der Daten, mit der Java Variante identisch (siehe Listing 6.11). Die zusätzlichen Parameter sind erforderlich, da nur ein Zeiger auf die Daten übergeben wird, über den nicht wie aus Java bekannt die Größe des Arrays abgefragt werden kann. Die Schnittstelle, die von einer Client-Anwendung implementiert werden muss, wird in der Datei IPCClientInterface.hpp deklariert. Diese fordert im Gegensatz zu ihrem Java-Pendant eine zusätzliche Methode $process()$ ein. Diese ruft der Service für jeden registrierten Client zyklisch auf und stellt dadurch den laufenden Prozess eines Clients dar. Da kein Betriebssystem verwendet wird,

[2] Details hierzu können in der Android-Referenz zu support-librarys eingesehen werden (siehe http://developer.android.com/tools/support-library/setup.html).

welches die Ausführung der Prozesse unterbrechen kann, muss sich jeder Client kooperativ verhalten und sollte keine lang andauernden Operationen in seiner $process()$ Methode ausführen. Hierdurch wird direkt die Latenz zwischen den Aufrufen einzelner Prozesse beeinflusst.

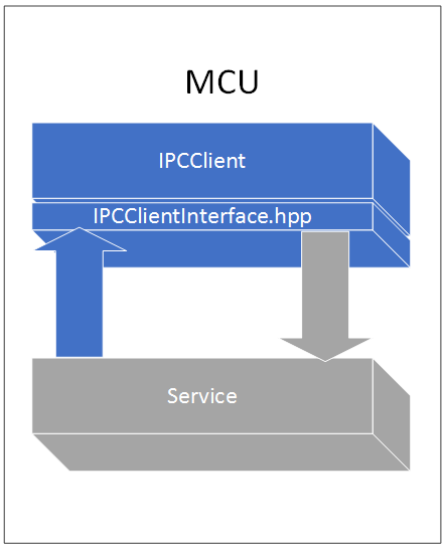


Abbildung 6.4.: Direkte Kommunikation über Methodenaufrufe zwischen Client-Anwendung und Service auf der MCU.

```
1
2 PROCESSING_STATE process();
3 int transmitDataAsync(uint16_t src, uint16_t dst, uint8_t msgType,
     uint8_t* data, uint16_t dataLen);
4 int transmitDataSync(uint16_t src, uint16_t dst, uint8_t msgType,
     uint8_t* data, uint16_t dataLen);
5 int registerClient(IPCClientInterface* clientStub);
6 int subscribeForMessages(IPCClientInterface* client, uint8_t msgType
     );
7 int unsubscribeForMessages(int clientID, int msgType);
```

Listing 6.11: IPCService.cpp (public Methoden)

Innerhalb des Services wird zum Lesen von Daten die $performReadOperations()$-Methode verwendet. Sie ließt Daten aus dem TransportEndpoint, in einen Puffer im Service. Zu beginn eines Lesevorgangs wird immer nur die Größe des Headers angefordert. Anschließend wird der Header gemäß Abbildung 5.6 bzw. Abbildung 5.7 interpretiert, die restlichen Daten der Nachricht gelesen und die Daten mit den entsprechenden Parametern an die $OnReceive()$-

Methode der empfangenden Clients übergeben. Da eine Referenz auf den Puffer des Services übergeben wird, muss der Client die Daten direkt im Aufruf der $OnReceive()$-Methode verarbeiten oder zur späteren Verwendung in einen eigenen Speicherbereich kopieren. Wenn der Client die empfangenen Daten verändern soll, muss er diese vorher ebenfalls kopieren, da bei Broadcast-Nachrichten allen Clients eine Referenz auf den selben Speicher übergeben wird um Kopiervorgänge zu minimieren und den Speicherverbrauch gering zu halten. Bei synchronen Nachrichten wird beim Empfang die Transaktionsnummer in einer Liste vermerkt um anschließend dem Absender mit einer ACK-Nachricht den Empfang zu bestätigen.

6.3.2. Transportschicht

Das C++ Pendant zur Schnittstelle der Transportschicht wird in der Datei TransportEndpointInterface.hpp definiert. Die Schnittstelle ist, wie bereits bei den Methoden zum versenden von Nachrichten, bis auf einen zusätzlichen Parameter für die Größe der zu schreibenden Daten und einer $process()$-Methode gleich aufgebaut. Die $process()$ Methode stellt wie bereits bei den Clients die Prozessausführung dar und kann dazu verwendet werden, zyklisch Aufgaben der Transportschicht auszuführen. Auch sie sollte nicht blockieren, da kein Betriebssystem eingesetzt wird, wodurch alle Abläufe sequenziell ausgeführt werden.

Zur Implementierung der konkreten Variante OpenAccessoryUSBTransport wird die USB-Bibliothek von STMicroelectronics und Teile des "HelloADKProjektes verwendet.

Die Abläufe zur Initialisierung des AOA-Protokolls können vollständig vom "HelloADKProjekt wiederverwendet werden. Diese funktionieren nach dem unter Abschnitt 4.4 beschriebenen Prinzip und sind als Zustandsautomat realisiert, der indirekt über die USB-Bibliothek aufgerufen wird.

USB Bibliothek

Die USB-Bibliothek ist in Version 2.1 in Form von Quelldateien verfügbar, welche direkt ins Projekt integriert werden können. Die Dokumentation dazu ist in der User manual UM1021 auf der Herstellerseite [3] erhältlich.

Unter Abbildung 6.5 ist die logische Struktur der USB-Bibliothek zu erkennen. Dem Entwickler werden mit der Bibliothek das library- und das low level driver-Modul zur Verfügung gestellt. Zusätzlich werden als USB-Klassen Modul eine Implementierung einer HID- (Human Interface Device) und einer MS- (Mass Storage) Klasse als Klassenmodul mitgeliefert. Das library-Modul kommuniziert dabei direkt mit dem Klassenmodul.

[3] http://www.st.com/st-web-ui/static/active/jp/resource/technical/document/user_manual/CD00289278.pdf

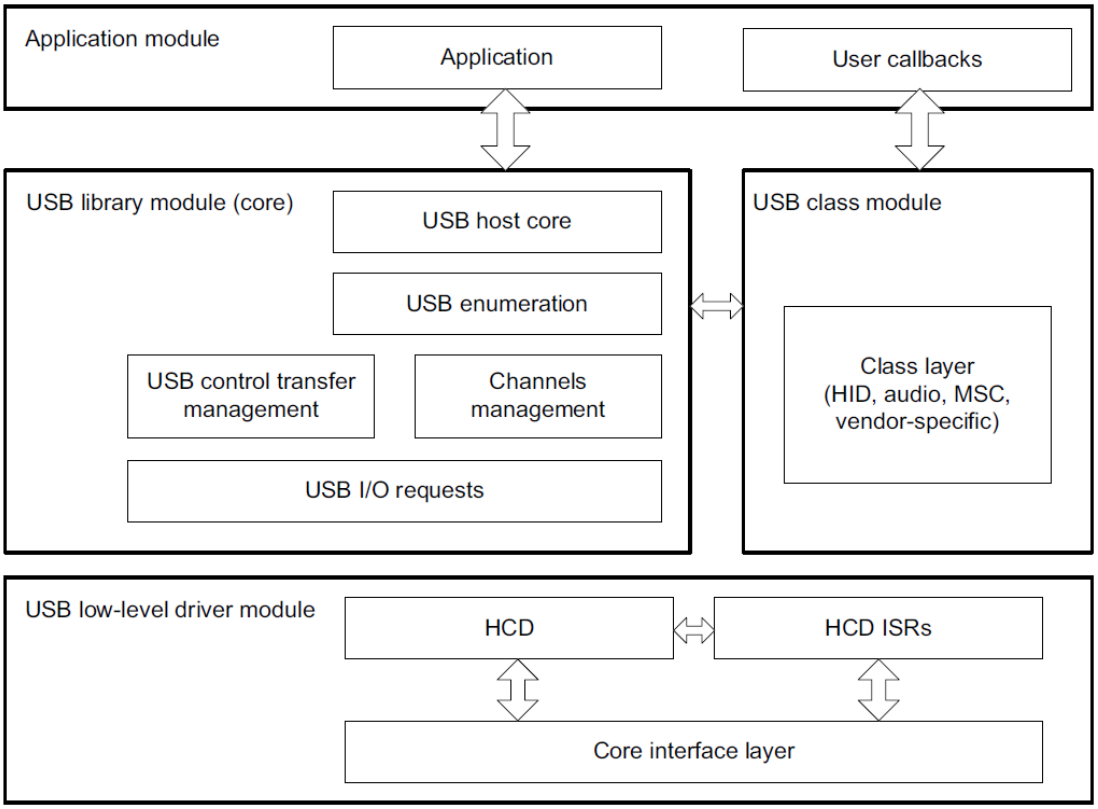


Abbildung 6.5.: USB Hostbibliothek Übersicht (Quelle: Usermanual UM1021 von STMicroelectronics)

Zur Initialisierung der Hostbibliothek wird die Funktion USBH_Init (siehe Listing 6.12) verwendet. Die Parameter haben folgende Bedeutung:

USB_OTG_CORE_HANDLE dient dem Zugriff auf die Konfiguration und Hardwareregister des USB Core.

USB_OTG_CORE_ID_TypeDef legt fest ob die High- oder Fullspeed Konfiguration des OTG-Core verwendet werden soll.

USBH_HOST zeigt auf die Variablen des Host Zustandsautomaten.

USBH_Class_cb_TypeDef beinhaltet Funktionszeiger auf callbacks der verwendeten Geräteklasse.

USBH_Usr_cb_TypeDef beinhaltet Funktionszeiger auf allgemeine callbacks über welche High-level Ereignisse der USB Peripherie signalisiert werden.

```
void USBH_Init(USB_OTG_CORE_HANDLE *pdev,
               USB_OTG_CORE_ID_TypeDef coreID,
               USBH_HOST *phost,
               USBH_Class_cb_TypeDef *class_cb,
               USBH_Usr_cb_TypeDef *usr_cb);
```

Listing 6.12: Initialisierungsfunktion der Hostbibliothek

Der Ablauf sämtlicher Funktionen der Bibliothek wird von einem Zustandsautomaten (siehe Abbildung 6.6) gesteuert. Zu dessen Ausführung muss vom Programm zyklisch USBH_Process(USB_OTG_CORE_HANDLE *pdev , USBH_HOST *phost) aufgerufen werden. Nachdem ein angeschlossenes Gerät erkannt und der enumeration Prozess durchlaufen wurde, wird im Zusatand HOST_USR_INPUT auf eine mögliche Nutzerinteraktion gewartet um mit der Ausführung der Geräteklassenfunktionen fortzufahren. Anschließend werden im HOST_CLASS_REQUEST Zustand typischerweise Aktionen ausgeführt um das Gerät für die Benutzung im Klassenmodus vorzubereiten. Dazu wird der „Requests“-callback des USBH_Class_cb_TypeDef aufgerufen, der an die Initialisierungsfunktion der Bibliothek übergeben wurde. Erst wenn dieser Vorgang abgeschlossen wurde, wird in den HOST_CLASS Zustand gewechselt in dem regelmäßig der „Machine“-callback aufgerufen wird, der einen Zustandsautomaten für die Klassenfunktionen ausführt.

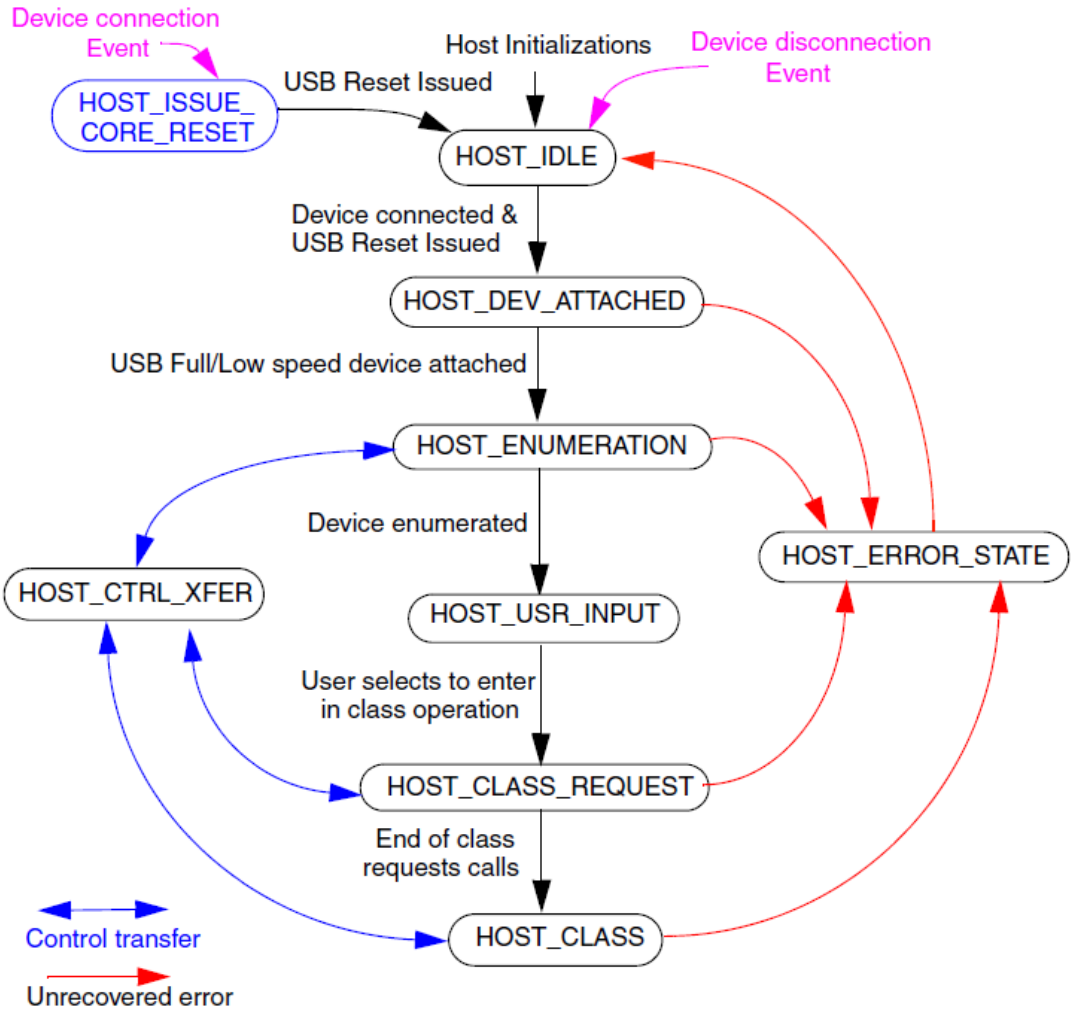


Abbildung 6.6.: FSM für Hostfunktionalität (Quelle: [1])

Die Interaktion zwischen library-Modul und Klassenmodul erfolgt über Funktionszeiger, die der Bibliothek zur Laufzeit in einem struct vom Typ USBH_Class_cb_TypeDef übergeben werden, der in der Datei usbh_core.h definiert ist.

Um mit der Bibliothek zu interagieren stehen diverse Funktionen zur Verfügung. Listing 6.13 Zeigt die beiden Funktionen zum Lesen und Schreiben in einen USB Bulk-Endpoint.

```
1 USBH_Status USBH_BulkSendData ( USB_OTG_CORE_HANDLE *pdev,
2                                 uint8_t *buff,
3                                 uint16_t length,
4                                 uint8_t hc_num)
```

```
5
6 USBH_Status USBH_BulkReceiveData( USB_OTG_CORE_HANDLE *pdev,
7                                   uint8_t *buff,
8                                   uint16_t length,
9                                   uint8_t hc_num)
```

Listing 6.13: Hostfunktionen zum starten eines Bulktransfers

Es ist zu beachten, dass die angeforderte Aktion nach dem Rücksprung in der Regel nicht fertig abgearbeitet ist. Hierzu muss gegebenenfalls mehrmals die Methode $USBH\_Process()$ aufgerufen werden, damit der Zustandsautomat die zur Übertragung notwendigen Control-Transfers ausführen kann. Um den Fortschritt einer angeforderten Übertragung, der innerhalb der USB-Bibliothek mittels Interrupt aktualisiert wird zu prüfen, stehen die in Listing 6.14 gezeigten Funktionen bereit.

```
1 URB_STATE HCD_GetURB_State (USB_OTG_CORE_HANDLE *pdev , uint8_t
      ch_num)
2 HC_STATUS HCD_GetHCState (USB_OTG_CORE_HANDLE *pdev ,  uint8_t
      ch_num)
3 uint32_t HCD_GetXferCnt (USB_OTG_CORE_HANDLE *pdev, uint8_t ch_num)
```

Listing 6.14: Statusfunktionen der USB-Bibliothek

Bei all diesen Funktionen wird eine Referenz auf einen USB_OTG_CORE_HANDLE übergeben in dem sämtliche Variablen des verwendeten Host-Controllers verwaltet werden. Zusätzlich muss angegeben werden, für welchen Endpoint die Daten abgefragt werden sollen. Der Zustand wird zwischen Lese- und Schreiboperationen also getrennt gehalten.

Mit $HCD\_GetHCState()$ können detailliert Informationen über den Status eines Transfers eingeholt werden. Dadurch lässt sich bei einem Fehler z. B. auslesen ob es sich um einen Data-Toggle Fehler, CRC-Fehler etc. handelt,

Bei $HCD\_GetURB\_State()$ wird der Zustand aus einer höheren Ebene betrachtet und es wird beispielsweise zurückgegeben, dass der Vorgang bereits vollständig abgearbeitet wurde, dies noch andauert, oder dass ein Fehler aufgetreten ist. Ein Fehler wird dabei nicht genauer spezifiziert, wobei ein STALL-Zustand einen eigenen Rückgabewert besitzt.

Über $HCD_GetXferCnt()$kann ausgelesen werden wie viele Bytes bisher übertragen wurden.

Zunächst wurde versucht auch die Datenübertragungsfunktionalität des HelloADK-Projektes zu verwenden. Tests haben jedoch gezeigt, dass die rudimentäre Implementierung nur zur Übertragung einzelner Bytes geeignet ist und sie sich bei größeren Datenmengen instabil und

fehleranfällig verhält. Des weiteren führt der zur Übertragung verwendete Zustandsautomat dazu, dass die USB-Bandbreite nicht Symmetrisch auf beide Senderichtungen aufgeteilt wird. Ein weiterer Nachteil ergibt sich dadurch, dass die Datenübertragung wie bei der USB-Bibliothek nicht direkt in den lese- bzw. schreib Methoden ausgeführt wird. Wodurch das Abfragen des Übertragungsfortschritts und das Handling der USB-Bibliothek separat, außerhalb der jeweiligen Methode ausgeführt werden muss.

Statt wie bei der HelloADK-Implementierung im Zustandsautomat, wird die Datenübertragung jeweils direkt in den $read()$- und $write()$-Methoden ausgeführt. Die $read()$-Methode signalisiert über einen negativen Rückgabewert, dass keine Daten gelesen werden können und kehrt in diesem Fall sofort zurück, um die Ausführung nicht unnötig zu blockieren. Das write blockiert bis die übergebenen Daten vollständig übertragen wurden, und kehrt andernfalls nur zurück wenn ein Fehler auftritt.

6.4. Beispielanwendungen

6.4.1. AOAPClientExample

Die AOAPClientExample Anwendung soll Benutzereingaben auf dem Android Gerät eine einfache Visuelle Rückmeldung auf dem MCU-Board erzeugen. Dazu sind zwei Modi auswählbar. In der Startkonfiguration der Anwendung kann der Nutzer durch Tippen und Wischen auf einer farbigen Fläche die vier verschiedenfarbigen LEDs des DISCOVERY-Boards steuern. Dazu werden die Koordinaten, welche im onTouch Callback erhalten werden an die MCU gesendet und dort mittels PWM auf den LEDs ausgegeben. Eine Berührung in der Mitte der Fläche dimmt alle LEDs auf die dunkelste Stufe. Je weiter die Berührung an den Außenkanten liegt desto heller leuchtet die jeweilige Farbe auf. Als alternative kann zur Eingabe auch der integrierte Beschleunigungssensor des Android-Gerätes verwendet werden. Diese Demo dient dem Zweck der Veranschaulichung der Datenübertragung und um Subjektiv eventuelle Latenzen einschätzen zu können. Des weiteren werden die zur Übertragung verwendeten Daten als Anschauungsbeispiel mit dem ProtcolBuffer Format serialisiert.

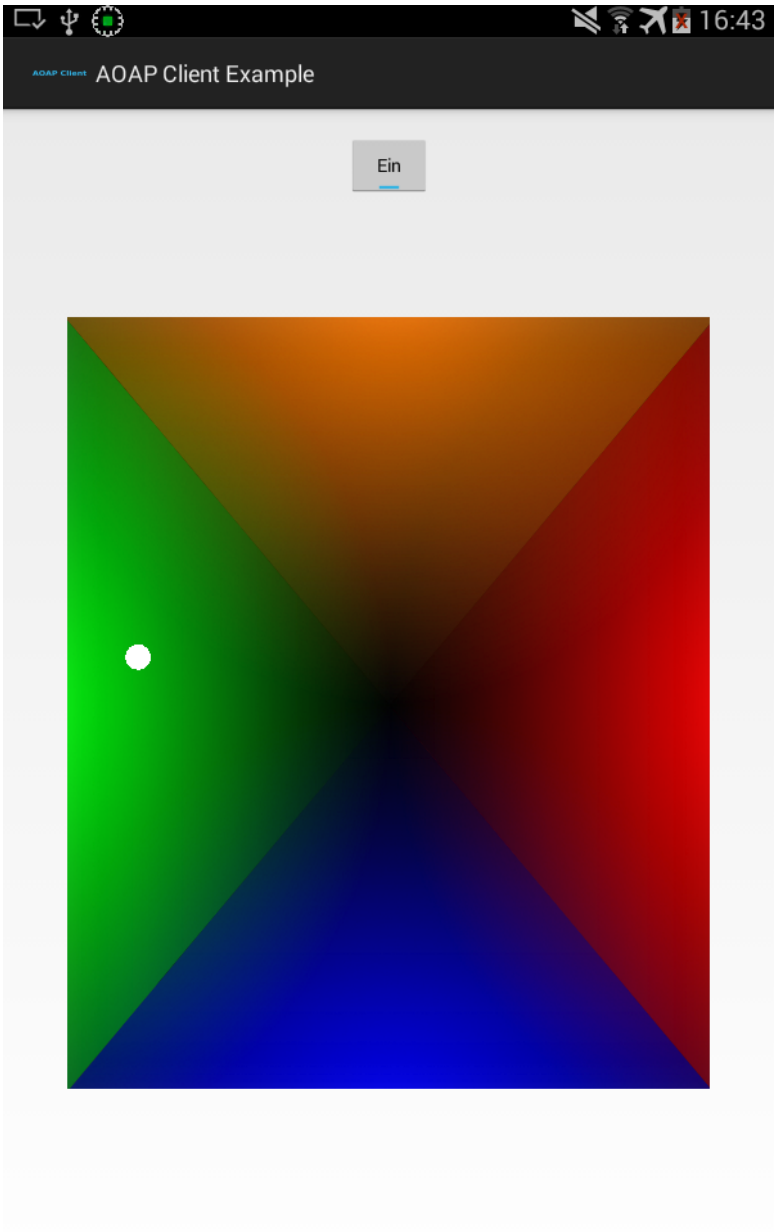


Abbildung 6.7.: AOAPClientExample Anwendung

6.4.2. AOAPClientSyncTX

Die Anwendung AOAPClientSyncTX bietet verschiedene Funktionen die hauptsächlich zur Evaluierung dienen. Es wird an einem einfachen Beispiel die Funktion synchroner Nachrichten gezeigt. Außerdem wird eine Benchmark Funktionalität integriert, die den Datendurchsatz misst. Zusätzlich steht eine Option zur Verfügung, womit übertragene Daten auf Beiden Seiten auf Korrektheit geprüft werden.

Zur Demonstration Synchroner Nachrichten steht in der Android Anwendung ein Button zur Verfügung der über die Methode $transmitDataSync()$ der Dienstschnittstelle, eine Synchrone Nachricht mit dem Nachrichtentyp MSG_TYPE_SYNC_TEST_WAIT_FOR_BUTTON_PRESS an die MCU überträgt. Wenn der Zielprozess eine Nachricht mit diesem Typ empfängt wird eine Schleife betreten die erst beendet wird, sobald der USER-Button auf dem DISCOVERY-Board gedrückt wird. Es ist zu beachten, dass dies ausschließlich der Demonstration dient, um die synchrone Übertragung für den Anwender sichtbar zu machen. In einem Produktivsystem sollte in keinem Fall die $OnReceive()$-Methode einer Anwendung dauerhaft blockiert werden.

Die Benchmarkfunktion wird verwendet um den Datendurchsatz mit unterschiedlichen Paketgrößen in beide Richtungen zu testen. Dazu können jeweils die Paketgrößen und das automatische Senden von Daten, für Beide Senderichtungen unabhängig voneinander eingestellt werden. Die Paketgrößen entsprechen dabei der reinen Nutzlast der Nachrichten und nur diese wird zur Berechnung der Übertragungsraten verwendet. Die Konfiguration findet vollständig über die Android Anwendung statt, welche Änderungen an das Anwendungsgegenstück der MCU übermittelt.

Um zu verifizieren, dass die Daten ohne Fehler übertragen werden, könnte ein Prüfsummenverfahren verwendet werden. Dadurch kann aber nur sichergestellt werden, dass eine einzelne Nachricht korrekt übertragen wurden. Eine Nachricht die fälschlicherweise zweimal übermittelt wurde, wird damit jedoch nicht als Fehler erkannt.

Eine Möglichkeit zu verifizieren, dass die Datenübertragung über mehrere Nachrichten hinweg ordnungsgemäß funktioniert, ist, dem Empfänger vorab auf sicherem Weg mitzuteilen welche Daten er empfangen wird. Um nicht auf beiden Seiten große Testdatensätze speichern zu müssen, werden die Daten stattdessen erst zur Laufzeit erzeugt.

zu diesem Zweck, wird in der AOAPClientSyncTX Anwendung ein Pseudo Zufallszahlengenerator auf Basis eines linearen Kongruenzgenerators verwendet.

Ein linearer Kongruenzgenerator lässt sich mit folgender Gleichung Beschreiben (vgl. [25]):

$$X_{n-1} = (aX_n + c) \text{mod} m, n \geq 0$$

Mit einer geeigneten Parameterwahl für a, c und m lassen sich damit reproduzierbar Zufallszahlen generieren. Wie unter [26] gezeigt lässt sich die Modulo-Operation auf Systemen mit 32-Bit Integer Repräsentation einsparen indem für m der Wert 2^{32} gewählt wird. Bei der Multiplikation zweier long-variablen werden nur die unteren 32 Bit des eigentlich 64 Bit breiten Ergebnisses gespeichert, was exakt einer Modulo-Operation mit 2^{32} entspricht.

Durch $m = 2^{32}$ ergeben sich $a = 1664525$ und $c = 1013904223$ als geeignete Parameter um eine ausreichend lange Periode von Pseudo-Zufallszahlen zu erhalten (vgl. [26]). Diese werden in der Methode $IPCClientTransmitSync :: getPseudoRandom()$ (siehe Listing 6.15) verwendet, welche als Parameter immer den zuletzt zurückgegebenen Wert erhält, als startseed wird 0 verwendet. Der aktuelle Zustand der Zufallszahlenkolonne wird, in Form des letzten zurückgegebenen Wertes, auf beiden Seiten jeweils für Sende- und Empfangsvorgänge separat gespeichert. Dieser wird nur zur Initialisierung des Systems zurückgesetzt und mit jeder erhaltenen Nachricht weiter entwickelt.

```
1 uint32_t IPCClientTransmitSync::getPseudoRandom(uint32_t seed){
2         return (uint32_t)(1664525L*seed + 1013904223L);
3 }
```

Listing 6.15: Funktion zum generieren von Pseudo-Zufallszahlen

Da Java 8 unter Android bisher nicht unterstützt wird, steht kein Datentyp zur Verfügung, mit dem direkt das Verhalten einer unsigned long Multiplikation abgebildet werden kann. Um die Funktion dennoch analog zur oben gezeigten Variante zu implementieren wird zur Berechnung statt eines einfachen Datentypen die BigInteger Klasse verwendet. Die Modulo-Operation wird realisiert indem nur die Wertigkeit der unteren 8-Bits zurückgegeben wird.

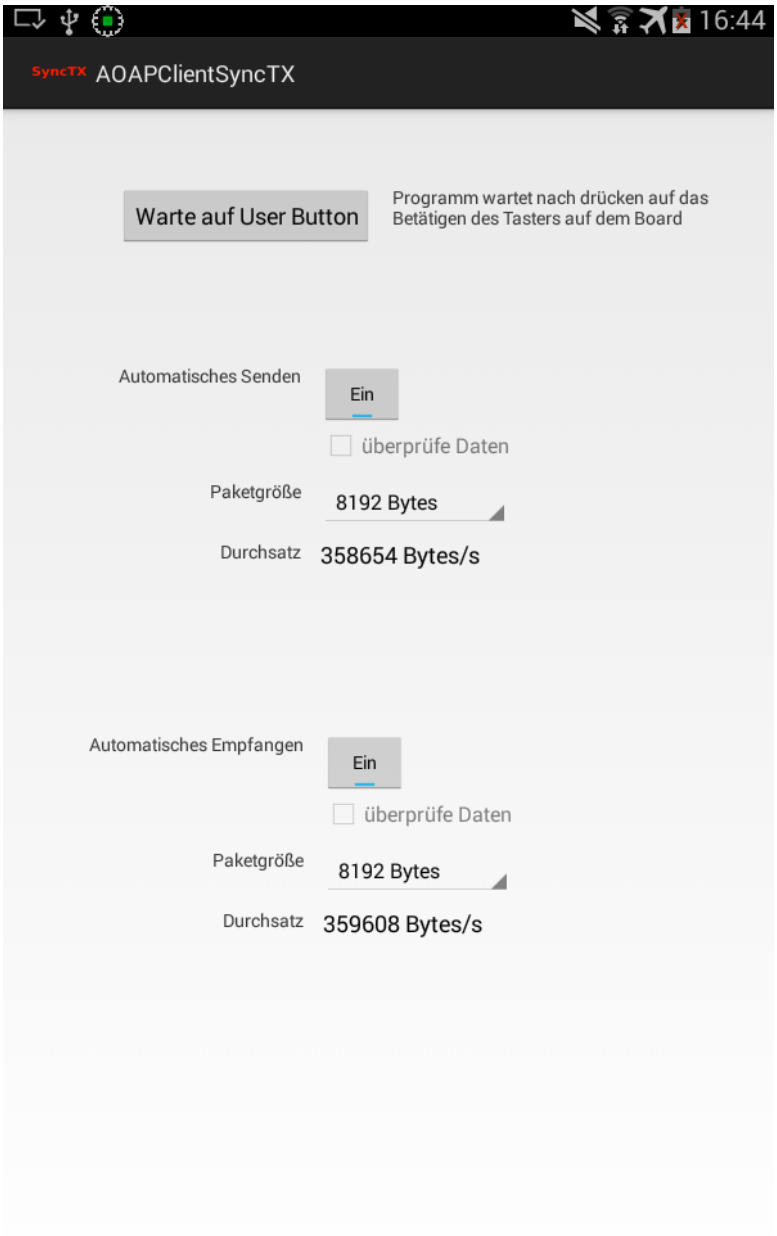


Abbildung 6.8.: AOAPClientSyncTX Anwendung

6.4.3. AOAPClientTimesync

Zur Zeitsynchronisation zwischen dem Android-Gerät und der MCU wird eine leicht abgewandelte Variante des Berkeley-Algorithmus verwendet. Den Zeitserver stellt das Android-Gerät dar. Somit soll in diesem Fall stets die Zeit der MCU an die des Androidsystems angeglichen werden.

Der Ablauf gestaltet sich wie folgt:

1. APP fordert Zeitstempel von MCU zum Zeitpunkt t_0 an
2. MCU sendet nach Empfang aktuellen Zeitstempel
3. Die Antwort $T_{MCU}(t_1)$ wird von APP zum Zeitpunkt t_1 empfangen
4. Round Trip Time wird berechnet mit $RTT = (t_1 - t_0)$
5. Die Momentane Zeit der MCU wird auf $T_{MCU}(t_1) + \frac{RTT}{2}$ geschätzt
6. Die Differenz $diff_n = T_{AND} - T_{MCU}$ zwischen beiden Zeiten wird ermittelt
7. $diff_n$ wird zwischengespeichert, wenn keine ungewöhnliche Schwankung vorliegt. 1-7 mehrmals mehrmals wiederholen
8. Mittelwert aller berechneten $diff$ an MCU übermitteln
9. Zeit auf MCU um $diff$ anpassen

Timer 3, welcher zum Darstellen der Millisekunden verwendet wird, wird bei dem eingestellten Systemtakt von 168 MHz mit einer Frequenz von 84 MHz betrieben. Diese wird durch den Prescaler von 42000-1 um den Faktor 42000 geteilt. Der Timer zählt 2000 Schritte, was einer Auflösung von 0,5 Millisekunden entspricht. Timer 3 taktet über die „Timer Synchronisation“-Funktion direkt Timer 5, welcher somit die Sekunden darstellt.

Die Steuerung der Zeitsynchronisation wir über einen kleinen Zustandsautomaten (siehe Abbildung 6.9) abgewickelt, dessen Ausführung direkt in der $onReceiveData()$-Methode stattfindet. Der Automat kennt drei verschiedene Zustände. Der COLLECT Zustand dient nur dazu, ausreichend Messungen der RTT zu sammeln bevor diese in die nachfolgenden Berechnungen einfließen. Nachdem initial r Werte gesammelt wurden, wird nicht wieder in diesen Zustand zurück gewechselt. Im OBSERVE Zustand wird aus den zuvor gesammelten Daten der Mittelwert und die Standardabweichung berechnet. Wie unter Schritt 7 angegeben wird der zuletzt errechnete Wert verworfen, wenn er eine ungewöhnlich hohe Schwankung aufweist. Diese Grenze für die Schwankung wird durch die doppelte Standardabweichung der gemessenen RTTs markiert. Mit dem Mittelwert der gültigen RTT-Werte wird anschließend die Zeitdifferenz zwischen beiden Uhren ermittelt. Diese werden um eine Höhere Robustheit gegen kurzzeitige Schwankungen zu erreichen, über die letzten m Berechnungen gemittelt.

Der Automat behält den OBSERVE Zustand bei, bis die Abweichung beider Uhren einen vorgegebenen Schwellenwert erreicht. Daraufhin wechselt der Zustand direkt zu SYNC. In diesem wird ein eventueller Drift der MCU-Uhr berechnet und eine Nachricht mit der Abweichung und dem Drift an sie übermittelt. Die Drift-Berechnung kann dabei nur stattfinden, wenn bereits vorher eine Synchronisation der Uhr stattgefunden hat.

Die Werte r und m wurden experimentell bestimmt und jeweils auf 100 festgelegt.

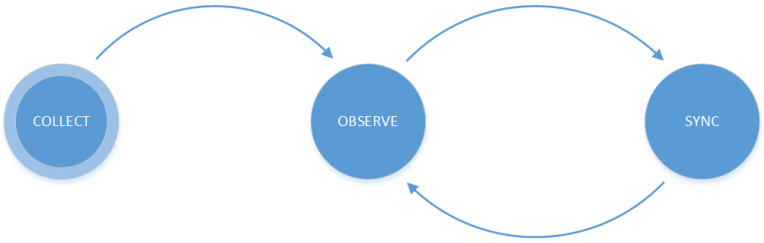


Abbildung 6.9.: Zustandsautomat zur Zeitsynchronisation

Die Anwendung (siehe Abbildung 6.10) stellt zur Interaktion zwei Buttons bereit. $SendeZeitstempel$ ist für Debugging Zwecke gedacht und sendet einen einzelnen Zeitstempel an die MCU. $Auto-Sync$ aktiviert oder deaktiviert das automatische Senden einer Zeitkorrekturnachricht. Die Anwendung erfragt in jedem Fall zyklisch die Zeit der MCU und aktualisiert regelmäßig die Anzeige der oben beschriebenen Werte.

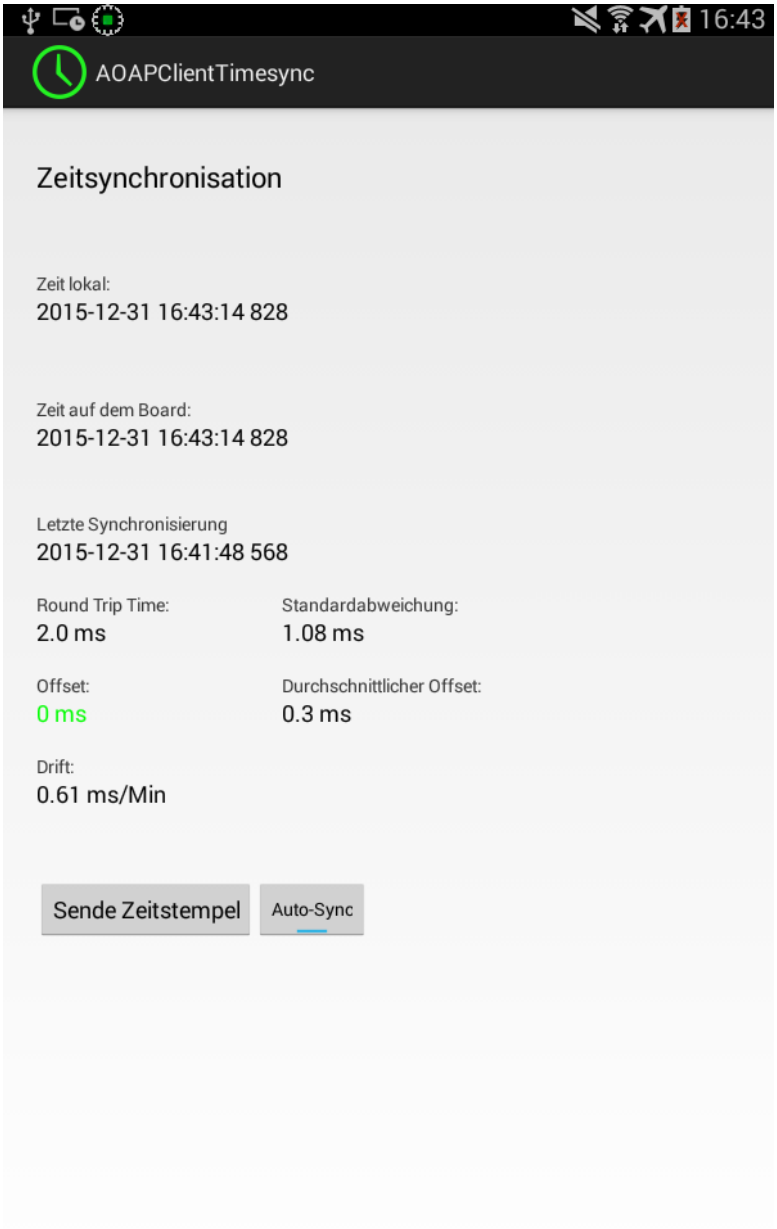


Abbildung 6.10.: AOAPTimesync Anwendung

7. Evaluierung

Im folgenden Kapitel wird zunächst das Vorgehen zur Evaluierung des Systems beschrieben. Im Anschluss wird überprüft ob die von der Zeitsynchronisationsanwendung durchgeführten Berechnungen korrekt sind. Zur Durchführung der Evaluierung wird auf die anderen zuvor vorgestellten Beispielanwendungen zurückgegriffen. Es sollen Messungen zum Durchsatz, zum Latenzverhalten vereinzelter Übertragungen und zu den Auswirkungen bidirektionaler Kommunikation durchgeführt werden. Die Ergebnisse sollen jeweils Auswirkungen bei der Verwendung unterschiedlich großer Pakete herausstellen und eine Aussage zur Leistungsfähigkeit erlauben. Die erhaltenen Ergebnisse werden im Nachfolgenden Abschnitt ausgewertet.

7.1. Allgemeine Vorbereitungen

Um unverfälschte Ergebnisse zu erhalten wird vor Ausführung der Messungen der Flugmodus aktiviert um Funkschnittstellen wie W-LAN und Bluetooth zu deaktivieren und damit Seiteneffekte durch andere Anwendungen zu minimieren. Außerdem wird der Energiesparmodus deaktiviert und der Bildschirm-Timeout auf 30 Minuten gesetzt. Anschließend wird das Tablet sauber herunter gefahren und neu gestartet. Nach dieser Konfiguration wird 15 Minuten gewartet damit das Tablet sich im IDLE-Zustand befindet. Für die MCU genügt ein Druck auf den Reset-Button. Erst im Anschluss an diese Prozedur wird per USB die Verbindung mit der MCU hergestellt.

7.2. Durchsatz

Zur Messung der Übertragungsraten wird beim Senden bzw. Empfangen in der AOAPClient-SyncTX Anwendung die Datenmenge summiert und in Relation zur Zeit als Übertragungsrate auf dem Bildschirm ausgegeben.

7.2.1. Unidirektional

Die Messung der Übertragungsrate erfolgt mit verschiedenen Paketgrößen. Bei der ersten Messreihe wird stets nur in eine Richtung übertragen. Die Übertragung findet für jede Paketgröße 20 Sekunden lang in einer Dauerschleife statt. Am ende eines Übertragungszyklus wird die gemessene Übertragungsrate gespeichert. Bevor ein neuer Zyklus mit einer anderen Paketgröße gestartet wird, ruht das System 5 Sekunden. Zur Vereinfachung der Nomenklatur wird die Übertragungsrichtung fest aus Sicht des Tablets betrachtet, da dieses die Oberfläche für die Nutzerinteraktion bereitstellt. Die Übertragung vom Tablet zur MCU wird im folgenden als Senden, die umgekehrte Richtung als Empfangen betitelt.

Abbildung 7.1 zeigt den Durchsatz beim Senden. Die Übertragungsrate bewegt sich im Bereich Zwischen 4411 Bytes/s und 722.904 Bytes/s und steigt von der kleinsten Nachrichtengröße mit einem Byte, bis zur größten mit 8192 Bytes monoton an. Ab 512 Bytes Payload steigt die Übertragungsrate nur noch geringfügig.

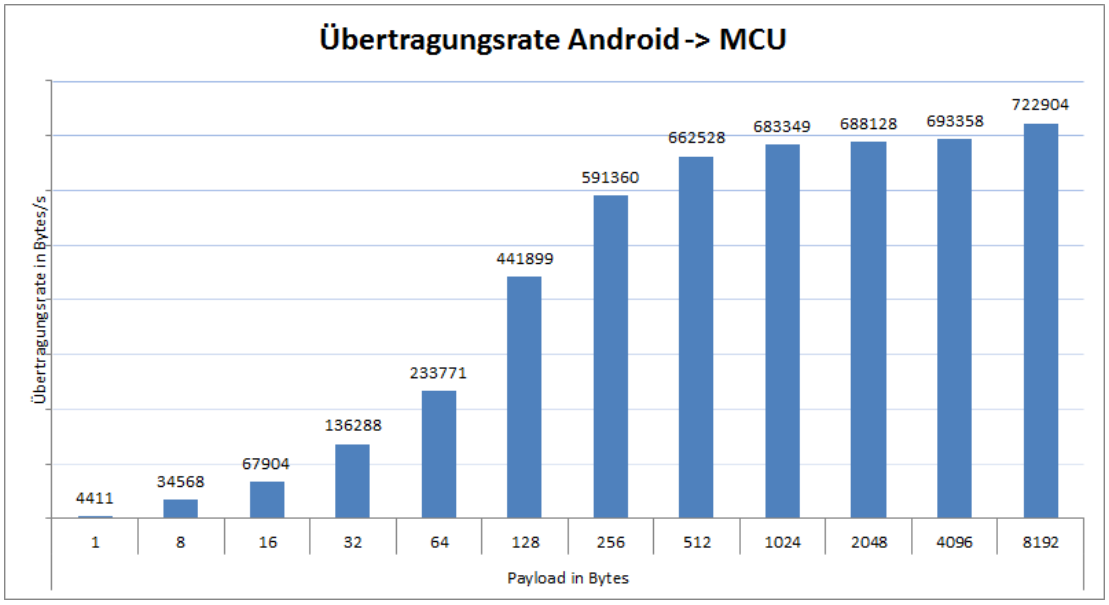


Abbildung 7.1.: Durchsatz Beim Senden in Richtung der MCU

Die unter Abbildung 7.2 gezeigte Übertragungsgeschwindigkeit in Empfangsrichtung fängt bei einem niedrigeren Wert von 2342 an, steigt mit den größer werdenden Paketen nicht so stark an wie wie beim Senden und hat Ihren Höchstwert bei 601.273 Byte/s. Die Übertragungsrate steigt auch hier monoton an, zeigt aber einen eher linearen verlauf, der im oberen Bereich, über 512 Byte, nicht so stark abflacht.

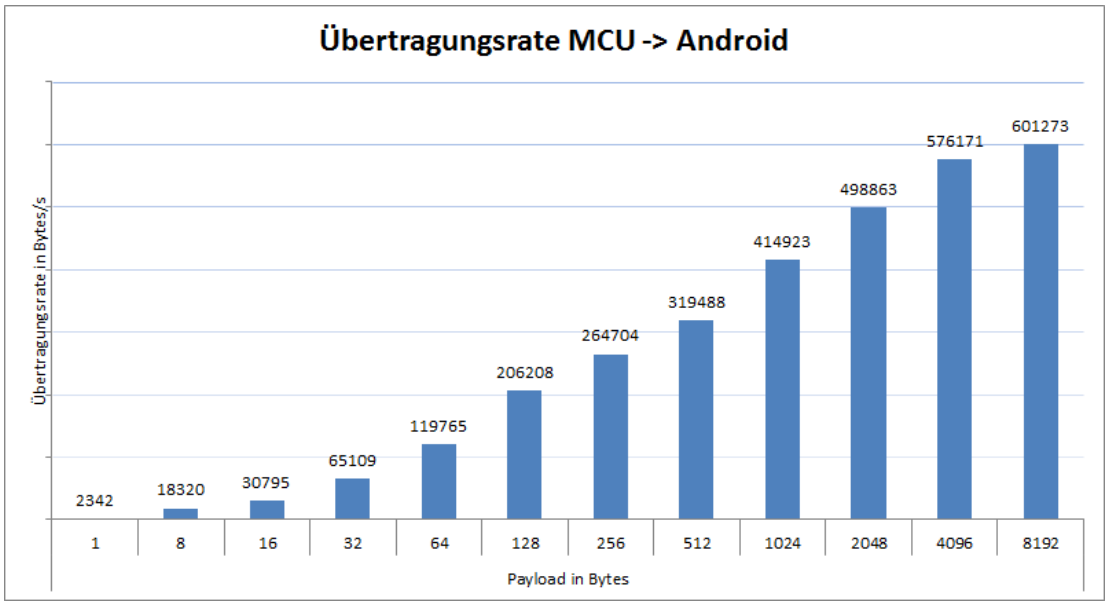


Abbildung 7.2.: Durchsatz beim Empfangen von Daten auf dem Tablet

7.2.2. Bidirektional

Eine weitere Messreihe wird erstellt um die Auswirkungen auf das System bei bidirektionaler Kommunikation zu untersuchen. Dabei wird wie bereits bei der unidirektionalen Übertragung vorgegangen, mit dem Unterschied, dass beide Übertragungsrichtungen gleichzeitig genutzt werden.

Abbildung 7.3 stellt den Sendedurchsatz bei gleichzeitigem Empfangen in Abhängigkeit beider Paketgrößen dar. Auf der horizontalen Achse werden die Paketgrößen zum Senden angegeben, während jede der farbigen Linien einer Paketgröße in Empfangsrichtung entspricht. Für sämtliche Empfangspaketgrößen ergibt sich ein Verlauf, der dem der unidirektionalen Übertragung in gleicher Richtung entspricht. Die Bandbreite wird auf beide Richtungen aufgeteilt. Bei gleicher Sende- und Empfangspaketgröße liefert weiterhin das größte Paket mit 356 kB/s den höchsten Durchsatz.

Auch Abbildung 7.4 welche der selben Darstellung in entgegengesetzter Richtung entspricht, liefert ebenfalls einen, der unidirektionalen Übertragung entsprechenden Verlauf. Der Verlauf ist hier bis zu einer Sendepaketgröße von 512 Byte annähernd identisch und weist kleinere Abweichungen auf, als in Abbildung 7.3 zu sehen sind.
Bei einer gleichen Paketgröße von 512 Byte werden in Beide Richtungen 253 kB/s erreicht.

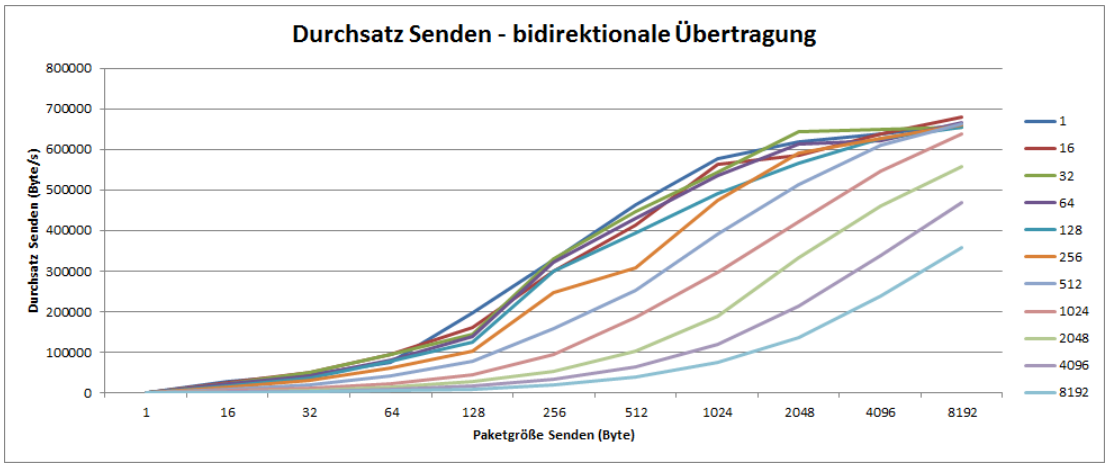


Abbildung 7.3.: Sendedurchsatz bei gleichzeitigem Senden und Empfangen von Daten

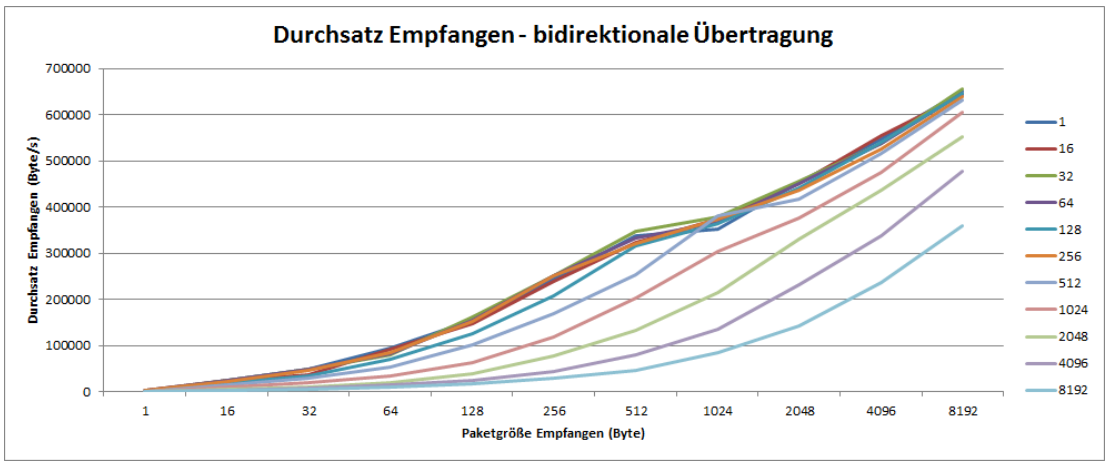


Abbildung 7.4.: Empfangsdurchsatz bei gleichzeitigem Senden und Empfangen von Daten

7.3. Schwankungen der Übertragungszeit

Bereits beim Testen der Anwendung zur Zeitsynchronisation fielen Schwankungen in der Übertragungszeit auf. Um diese Schwankungen und somit auftretende Latenzen zu messen, wird wie zuvor die Anwendung AOAPClientSyncTX genutzt. Zusätzlich kommt ein Oszilloskop des Modells „Tektonix DPO 4054“ zum Einsatz. Dieses bietet die Möglichkeit automatisch Pulslängen zu messen, die anfallenden Daten statistisch auszuwerten und anzuzeigen. In einem

ersten Versuch hierzu wird die Einfache Statistik-Funktion des Oszilloskops dazu verwendet einen Eindruck von den auftretenden Latenzen zu bekommen.

Wenn für die Messung der Pulslänge die Statistikfunktion aktiviert wird, werden außer dem aktuellen Messwert die über die Zeit ermittelten Werte, für den Mittelwert, das Minimum/Maximum und die Standardabweichung der laufenden Messung angezeigt. In diesem Modus können der Mittelwert und die Standardabweichung allerdings nur über maximal 1000 Messungen errechnet werden. Bei einer Übertragungszeit von Beispielsweise 4 Millisekunden kann damit maximal ein Zeitbereich von 4 Sekunden betrachtet werden. Da die beobachteten Schwankungen in der Regel seltener auftreten, wird diese Funktion nur zum Einschätzen des Messbereichs verwendet.

Um die Messungen über eine beliebige Zeitspanne abzudecken, bietet das Oszilloskop eine Histogrammfunktion, mit der gleichzeitig die Verteilung der unterschiedlichen Werte analysiert werden kann. Um genauere Aussagen zur maximalen Übertragungszeit treffen zu können, wird eine Messreihe mit einer Histogrammbreite von 180 ms erstellt. Die Zeitspanne wird zur Messung aller Paketgrößen beibehalten. Da hierzu die Abtastrate auf 5 MS/s reduziert wird, geht der größere Bereich mit einer höheren Messungenauigkeit einher.

Um bei der Erstellung des Histogramms durch große Unterschiede in der Übertragungszeit keine Überschneidungen zwei aufeinanderfolgender Nachrichten zu erhalten, werden von beiden Geräten immer nur zwei Nachrichten gesendet und anschließend wird 200 ms gewartet. Auch bei dieser Messreihe werden stets nur in eine Richtung Daten übertragen. Durch dieses Vorgehen entsprechen die Messwerte einem Anwendungsfall bei dem in einer recht niedrigen Frequenz Nachrichten übermittelt werden. Die langen Pausen zwischen den Sendevorgängen führen außerdem dazu dass sich die Daten nicht direkt mit denen einer permanenten Übertragung vergleichen lassen.

Nach den zu Anfang des Kapitels genannten, allgemeinen Vorbereitungen, wird die AOAPClientSyncTX Anwendung aufgerufen. Der erste Schritt jeder Einzelmessung ist das Einstellen der Parameter über die Benutzeroberfläche. Hierzu ist zunächst die Größe der Pakete auszuwählen und über die GUI das automatische Senden, entweder auf dem Tablet oder der MCU zu aktiviert. Nach dem aktivieren wird mindestens 20 Sekunden gewartet bevor die Datenerhebung gestartet wird. Diese erfolgte für jede Paketgröße über einen Zeitraum von 5 Minuten um auch seltene Ereignisse ausreichend oft aufzuzeichnen und möglichst realistische Maximalwerte zu erhalten.

Abbildung 7.5 zeigt die durchschnittliche Übertragungszeit in Senderichtung. Bis einschließlich der Größe von 32 Byte schwankt die Übertragungszeit um 0,78 ms. Erst ab 64 Bytes zeigt sich ein eindeutiger Aufwärtstrend. Die Werte zeigen ab hier, ebenso wie die Paketgröße, einen exponentiellen Verlauf, der jedoch schwächer ausgeprägt ist. Bei 8192 Bytes wird der Höchstwert von 12,47 ms markiert.

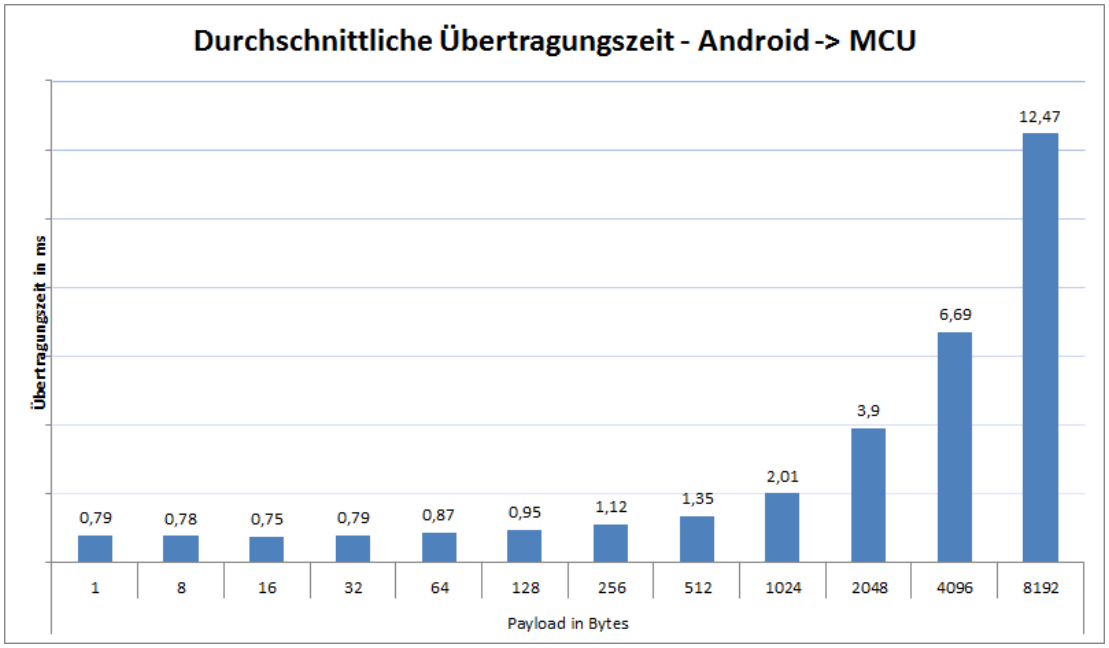


Abbildung 7.5.: Durchschnittliche Übertragungszeiten beim Senden in Richtung der MCU

Abbildung 7.6 stellt die durchschnittliche Übertragungszeit in Empfangsrichtung dar. Die gemessenen Werte sind bei allen Paketgrößen höher als bei der umgekehrten Übertragungsrichtung. Die maximal gemessene Latenz von der MCU zu Android steigt dadurch auf 13,91 ms, für eine Übertragung von 8192 Bytes.

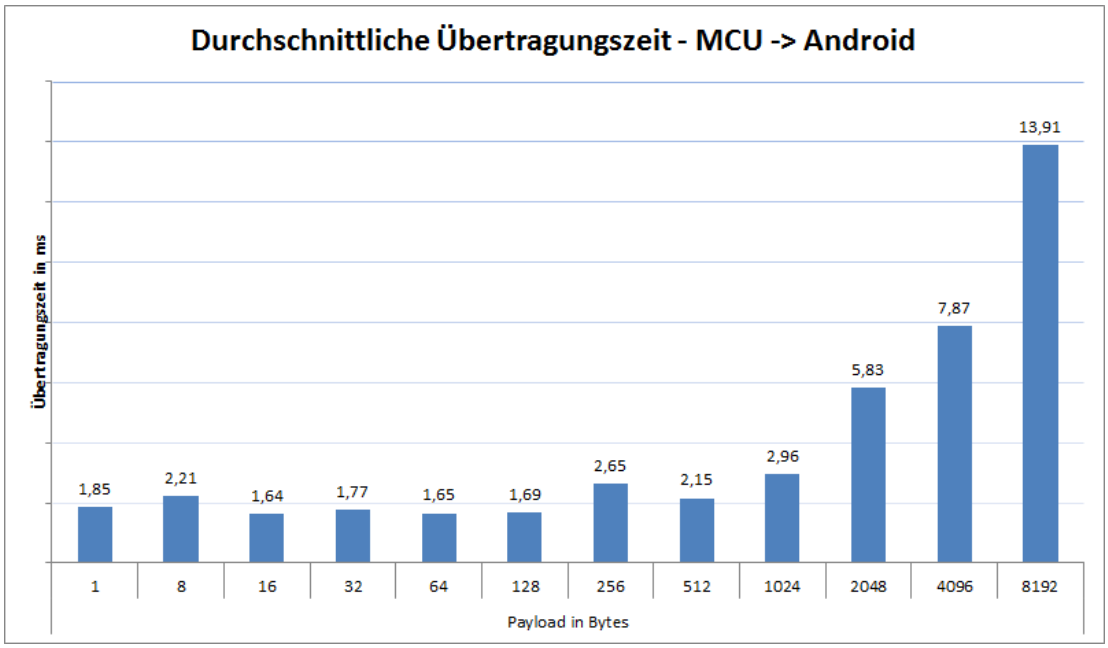


Abbildung 7.6.: Durchschnittliche Übertragungszeiten beim Empfangen von Daten auf dem Tablet

Abbildung 7.7 zeigt die maximal gemessenen Übertragungszeiten beim Senden. Die gemessenen Maxima korrespondieren im unteren Bereich nicht eindeutig mit den jeweiligen Paketgrößen. Der mit 4,7 ms kleinste Maximalwert wurde bei einer Paketgröße von 1024 Bytes gemessen. Das absolute Maximum bildet mit 32,9 ms die Nutzlast von 4096 Bytes, während dieser Wert bei dem größten Paket wieder auf 25,7 ms absinkt. Für alle Paketgrößen unterhalb von 4096 Bytes bleiben die Werte stets unter 13 ms.

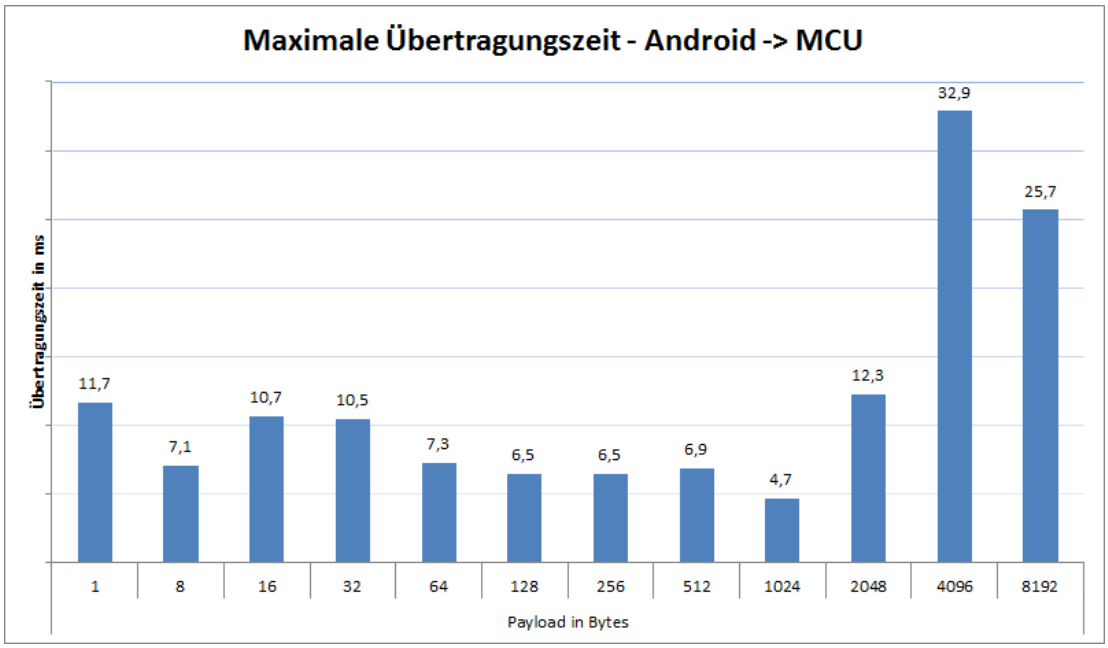


Abbildung 7.7.: Maximale Übertragungszeiten beim Senden in Richtung der MCU

Ein sprunghafteres Ergebnis zeigt Abbildung 7.8. In diesem wird die Maximale Übertragungszeit in Empfangsrichtung dargestellt. Bereits bei einer Paketgröße von 8 Bytes wurde eine Dauer von 28 ms gemessen, welche damit nur um 0,8 ms unterhalb des Maximalwertes bei 2048 Bytes liegt.

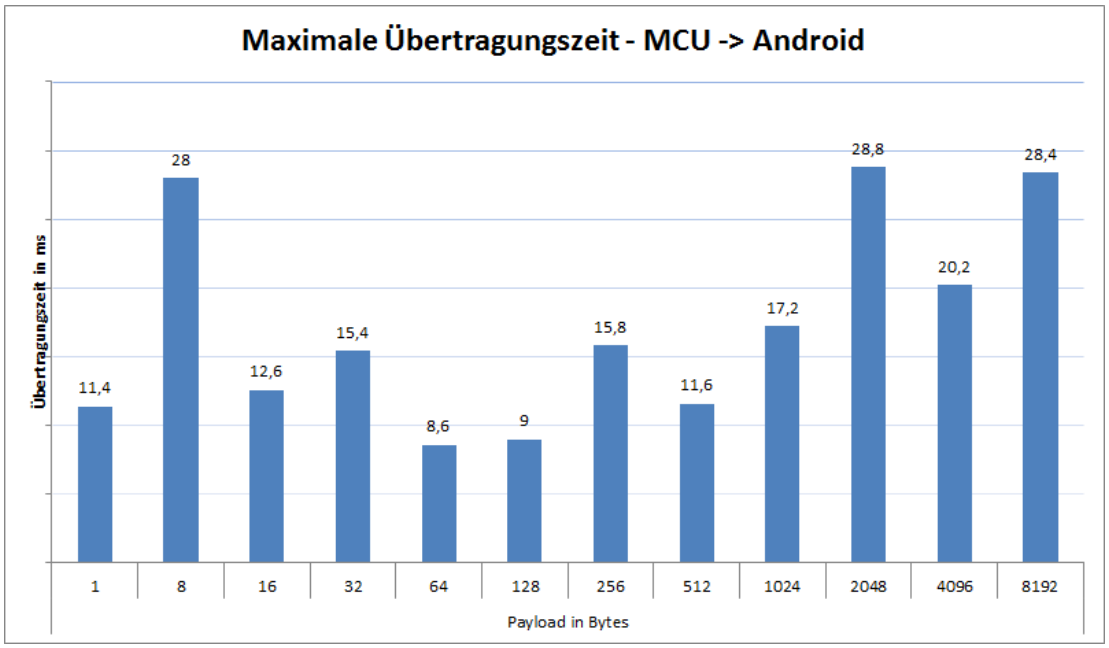


Abbildung 7.8.: Maximale Übertragungszeiten beim Empfangen von Daten auf dem Tablet

7.4. Driftberechnung

Die Anwendung zur Zeitsynchronisation wurde auf unterschiedlichen Geräten getestet[1]. Die angezeigten RTT-Werte bewegen sich in einem Bereich von 1-4 ms, wobei am häufigsten ein Wert von ca. 2 ms angezeigt wird. Beim Testen der Anwendung fiel auf, das vereinzelt überdurchschnittlich hohe Paketlaufzeiten von 10 ms und mehr gemessen wurden. Da diese separat behandelt werden, wird in diesem Teil hauptsächlich die Drift Angabe der AOAPClient-Timesync Anwendung überprüft. Mehrere Versuche haben gezeigt, dass sich der berechnete Drift mit der verwendeten Hardware innerhalb weniger Synchronisationsvorgänge auf ein Niveau von ~0,5 $\frac{ms}{min}$ einpendelt. Zunächst wurde dieser Wert mit den Datenblattangaben verglichen. Aus der Teileliste des DISCOVERY-Boards [2] geht hervor, dass als Oszillator ein Kristall (Modellbezeichnung: HC-49S-C20QSA-8.000MHZ) vom Hersteller jfvny verbaut ist. Im entsprechenden Datenblatt [3] wird für dieses Modell bei 25 °C eine Frequenztoleranz von

[1] Getestet wurde mit einem Samsung Galaxy Tab 3 7.0, dem Asus Nexus 7 (2013 Generation), und dem Lenovo Yoga Tablet 2 Pro 13,3)
[2] http://www.st.com/st-web-ui/static/active/en/resource/technical/document/bill_of_materials/stm32f4discovery_bom.zip
[3] http://search.alkon.net/cgi-bin/pdf.pl?pdfname=05616.pdf

$\pm 20 \times 10^{-6}$ also 20 PPM angegeben. Das ergibt bei einer Nominalen Frequenz von 8 MHz einen Takt zwischen 7999840 Hz und 8000160 Hz. Daraus lässt sich schließen, dass der Maximale Drift des Timers maximal $\frac{20Hz}{1000000Hz} \times 1000ms \times 60s = \pm 1,2 \frac{ms}{min}$ betragen sollte.

Zum Vergleich mit den genannten theoretischen Werten, wird eine Referenzmessung des Systemtaktes durchgeführt. Für diesen Zweck wird ein Pin auf dem Board so konfiguriert, dass er mit einer vom Systemtakt abgeleiteten Frequenz von 1 MHz schwingt. Dazu wird der Timer 2 mit dem Ausgangspin 2 (GPIOA2) verwendet. Dieser wird wie auch Timer 3 und 5 mit 84 MHz getaktet (siehe Abbildung 7.10). Durch einen auf 0 gesetzten Prescaler und der Output Compare Funktion im im Toggle-Modus mit dem Referenzzählwert von 41, wird die theoretische Frequenz am Ausgang auf 1 MHz eingestellt.

An dem Ausgangspin wird daraufhin mit einem Oszilloskop die tatsächliche Frequenz gemessen (siehe Abbildung 7.9). Der tatsächliche Wert liegt mit 8 Hz Abweichung innerhalb der angegebenen Toleranz. Die Abweichung entspricht damit einem Drift von $0,48 \frac{ms}{min}$.

Während dieser Messung wird in der AOAPClientTimesync Anwendung ein Drift von 0,50 ms/Min angezeigt.

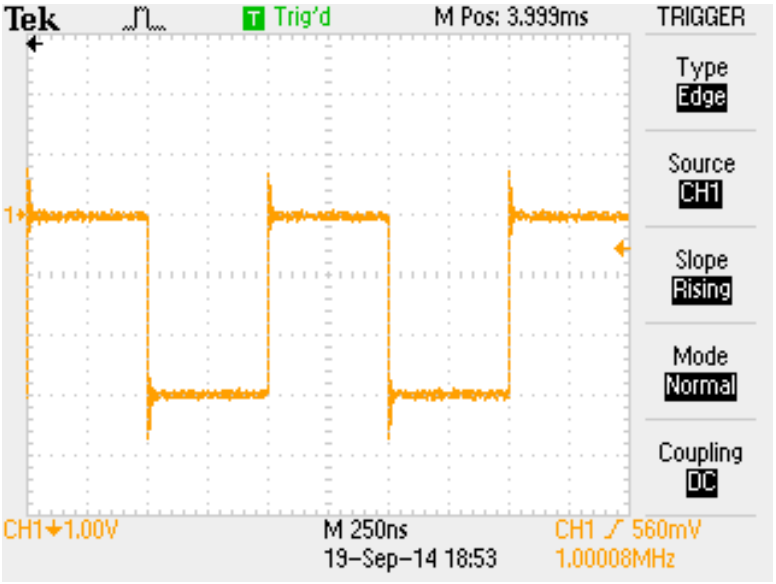


Abbildung 7.9.: Referenzmessung der Timerfrequenz

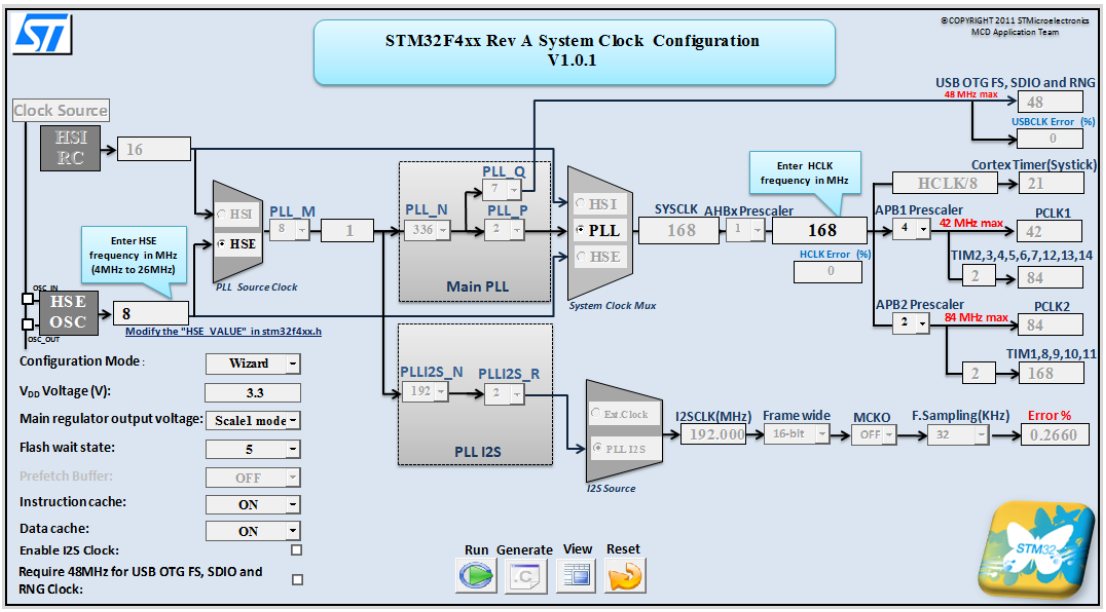


Abbildung 7.10.: Konfigurationstool mit verwendeten Takteinstellungen auf dem DISCOVERY-Board.

8. Diskussion

In diesem Kapitel werden die Ergebnisse der zuvor durchgeführten Evaluierung analysiert und erläutert. Im Anschluss werden einige Verbesserungen am Implementierten System vorgeschlagen.

8.1. Auswertung

Durchsatz

Zunächst wird nur die unidirektionale Übertragung betrachtet.

Die Messungen zum Durchsatz zeigen Unterschiede zwischen Sende- und Empfangsrichtung auf. Als erstes fällt der unterschiedliche Verlauf der Werte auf. An beiden Darstellungen ist jedoch gleichermaßen gut zu erkennen, dass die USB-Bulk Übertragung und das implementierte Protokoll erst bei größeren Datenmengen ihr Potential ausreichend nutzen. Im Vergleich zwischen den Paketgrößen von 1 und 16 Byte zeigt sich das mit einem bis zu Faktor 15 höheren Durchsatz, der hierbei hauptsächlich auf den Overhead durch den Nachrichten-Header zurückzuführen ist. Dieser nimmt mit 8 Bytes und einer Nutzlast von 1 Byte fast 90% der übertragenen Daten ein. Die Skalierung der Bulk-Übertragung nimmt mit zunehmender Paketgröße aber eine wichtigere Rolle ein als der verhältnismäßig schrumpfende Protokoll Overhead, der bereits bei 128 Bytes weniger als 6% der Nachricht darstellt.

Bei genauerem Vergleich beider Diagramme ist erkennbar, dass das Senden stets schneller abläuft. Bis einschließlich der Paketgröße von 512 Bytes ist der Durchsatz näherungsweise um Faktor zwei höher. Die Ursache hierfür liegt vermutlich darin, dass das USB Subsystem der MCU regelmäßig die Verfügbarkeit von Daten an der Gegenstelle abfragt, eine Überprüfung mit dem USB-Analyzer konnte diese Vermutung belegen. Durch das Master-Slave Prinzip von USB ist das Regelmäßige Anfragen durch den Host jedoch erforderlich. Hinzu kommt, dass die MCU durch das Daten-Polling im IDLE-Zustand vermutlich schneller auf verfügbare Daten reagieren kann als das Tablet. Dieser Effekt kann durch das Scheduling unter Android weiter verstärkt werden, da eine verzögerte Zuteilung der CPU an den Lese-Thread sich direkt auf die Latenz auswirken kann. Um diesen Effekt genauer zu untersuchen, kann eine detaillierte Analyse der

USB-Treiber Implementierung von Android, dessen Zeitverhalten und den Auswirkungen des Schedulings auf dessen Performanz zielführend sein. Dies soll aber im Rahmen dieser Arbeit nicht weiter verfolgt werden.

Mit den Messungen zur bidirektionalen Übertragung lässt sich zeigen, dass die Bandbreite bei gleicher Paketgröße gleichmäßig auf beide Richtungen aufgeteilt wird. Sind die Paketgrößen ungleich, ist das Lesen im Schnitt wieder etwas schneller. Die Ergebnisse führen zu dem Schluss, dass sich durch die gleichzeitige Übertragung der Gesamtdurchsatz nicht erhöht.

Eine der Anforderungen war es, über das System regelmäßig, innerhalb von 20 ms einen Datenblock von 512 Byte zu übertragen. Der Durchsatz des Systems beträgt bei einer Nutzlast von 512 Byte in beide Richtungen jeweils 253 kB/s, was eine Übertragung von 5060 Byte innerhalb von 20 ms in Beide Richtungen erlaubt. Diese Anforderung ist damit ausreichend erfüllt.

Latenz

Die Auswertung der durchschnittlichen Übertragungszeit liefert ein zum Durchsatz passendes Ergebnis. In Senderichtung ist die Übertragungszeit ebenfalls kürzer als beim Empfangen. Eine wesentliche Erkenntnis aus diesen Daten stellt das Verhältnis zwischen dem Durchsatz bei dauerhaftem Senden und dem Übertragen von Daten in verhältnismäßig niedriger Frequenz dar. Besonders im unteren Bereich bis 256 Bytes ist das kontinuierliche Senden stets um mehr als Faktor drei schneller.

Bei Betrachtung der Messungen zu den maximalen Übertragungszeiten fällt auf, dass sich im Bereich zwischen 1 und 2048 Byte kein eindeutiger Zusammenhang zwischen der Größe der Nutzlast und den maximal auftretenden Latenzen erkennen lässt, da bereits bei einer Größe von einem Byte eine maximale Latenz von fast 12 ms auftreten konnte. Im Gegenteil zeigen sich bei größeren Paketen von 1024 Bytes sogar niedrigere Übertragungszeiten. Die Schwankungen bis einschließlich 2048 Bytes können durch die relativ selten auftretenden Verzögerungen in dieser Größenordnung und die Messdauer erklärt werden. Die Vermutung liegt nahe, dass sich das Maximum bei allen Paketen bis 2048 Bytes um deren Maximum untereinander, im Bereich von ca. 12,5 ms bewegt. Ab 4096 steigt die maximale Latenz signifikant an, fällt aber bei 8192 wieder ab. Dieses Verhalten entspricht dem Ergebnis der im Abschnitt 4.5 vorgestellten Analyse. Das legt die Vermutung nahe, dass auch hier die Ursache in der initialen Größe des Kernel-Puffers zur Datenübertragung zwischen zwei verschiedenen Prozessen liegt (vgl [24]).

Ein zusätzlicher Versuch hat gezeigt, dass sich durch mehrere gleichzeitig aktive Programme (TimeSync/CLientExample/SyncTX) welche das Framework nutzen, die maximal auftretende

Übertragungszeit nicht zwingend erhöhen lässt. Es sollte aber davon ausgegangen werden, dass sich die gezeigten Übertragungszeiten in seltenen Fällen addieren können.

Die Ergebnisse zur Latenz bieten zwar Anreize für zukünftige Untersuchungen, stellen aber durch den Vergleich mit Ergebnissen der Analyse unter Abschnitt 4.5 keine Überraschung dar.

Zeitsynchronisation

Die durchschnittlichen Übertragungszeiten zeigen, dass eine Nachricht mit bis zu 32 Bytes Nutzlast ca. 0,78 ms benötigt um die MCU zu erreichen. Eine Antwort mit 8 Bytes benötigt allerdings schon 2,21 ms, was eine RTT von ~3 ms ergibt. Von der Anwendung werden meist ca. 2 ms als RTT angegeben was eine Differenz von 1 ms ergibt. Diese lässt sich damit erklären, dass bei der Durchschnittsmessung mit dem Oszilloskop auch die gezeigten Maximalwerte einfließen, diese von der Anwendung aber herausgefiltert werden. Eine zusätzliche Messung mit einer Histogrammbreite von nur 9 ms konnte diese Vermutung untermauern, da der Durchschnitt damit bereits bei 2,12 ms lag.

Die Analyse des berechneten Drifts führt, durch eine Gegenüberstellung der theoretischen Werte, keine Zweifel an der Plausibilität der angezeigten Werte herbei. Die Ergebnisse der Referenzmessung zeigen, dass der von der Anwendung gemeldete Wert von $0,50\frac{ms}{min}$ um 0,2 von dem mit $0,48\frac{ms}{min}$ tatsächlich gemessenen Wert abweicht. Durch die beschriebene Konfiguration des Timers führt die kleinste mögliche Erhöhung des Prescalers bereits zu einer Abweichung von $\frac{1Hz}{84000000Hz} \times 2000$ was gerundet $1,4286\frac{ms}{min}$ Drift entspricht. Die Genauigkeit der Drift-Berechnung kann also, durch die im Verhältnis relativ grobe Justiermöglichkeit, als ausreichend betrachtet werden.

Um eine bessere Justiermöglichkeit zu erreichen könnte die Konfiguration des Timers variiert werden. Dazu wäre die Verwendung von zwei 32 Bit Timern denkbar. Außerdem könnte für eine genauere Driftberechnung innerhalb der Android App über einen längeren Zeitraum der Verlauf beobachtet werden.

8.2. Verbesserungen

Nachfolgend sollen einige Verbesserungen genannt werden die in zukünftigen Versionen sinnvoll erscheinen, im Rahmen dieser Arbeit aber nicht behandelt werden konnten.

8.2.1. Transportschichten

Bei der Entwicklung wurde Wert darauf gelegt die Transportschicht austauschbar zu gestalten. Für einige Anwendungen wäre die Verwendung von kabellosen Kommunikationsschnittstellen besser geeignet, was WLAN und Bluetooth als mögliche Kandidaten herausstellt.

8.2.2. Kommunikationsverfahren

Einige mögliche Erweiterungen stellen zusätzliche Kommunikationsverfahren dar. Nützlich könnten beispielsweise RPC-Verfahren, ein virtueller Shared-Memory oder Streaming sein. All diese lassen sich zwar auch indirekt mit dem Framework realisieren, eine native Implementierung sollte aber effizienter und einfacher in der Benutzung sein. Als mögliche Verbesserung in zukünftigen Versionen seien außerdem Timeouts für die synchronen Aufrufe und eine Priorisierung gleichzeitig anfallender Nachrichten genannt, welche direkt im blockierenden Teil der AIDL-Aufrufe umgesetzt werden könnte (siehe Abschnitt 6.2.2).

8.2.3. Energieverbrauch

Das implementierte System verwendet die MCU als USB-Hostcontroller. Hierdurch stellt diese die Energie für den Bus zur Verfügung. Für manche Einsatzzwecke, besonders in Kombination mit Batteriebetrieb, könnte eine Stromversorgung über das Android-Gerät Vorteile bieten. Weiterführend wären Funktionen für einen autarken Langzeiteinsatz interessant. Dazu wäre denkbar die USB-Schnittstelle vollständig zu deaktivieren. Um dennoch von beiden Seiten, zu jeder Zeit eine Aktion auslösen zu können, ist es vorstellbar die MCU vom Android-Gerät aus, über einen Interrupt durch die Kopfhörerbuchse aufzuwecken.

8.2.4. Performanz

Wie im Kapitel 7 ersichtlich, wird die Bandbreite der verwendeten USB-Schnittstelle nicht vollständig ausgenutzt. Für die gestellten Anforderungen ist dies bei weitem ausreichend, ein höherer Durchsatz macht das System aber potentiell für weitere Einsatzmöglichkeiten Interessant. Als Flaschenhals werden das USB-Handling unter Android und die generelle Skalierung von USB vermutet, welche bei einer Bulk-Übertragung erst mit großen, an einem Stück übertragenen Datenmengen ihr volles Potential nutzbar machen. Für diese Arbeit wurde aufgrund der Anforderung zu möglichst vielen Geräten Kompatibilität herzustellen und der weniger umfangreichen API, darauf verzichtet eine Native C++ Implementierung mithilfe des NDKs zu realisieren. Damit könnte im Gegensatz zur verwendeten Accessory-Bibliothek der Overhead der Android VM umgangen und die Kommunikationsfunktionen auf einer

niedrigeren Abstraktionsebene realisiert werden, was eine höhere Geschwindigkeit in Aussicht stellt.

9. Fazit

Zum Abschluss der Thesis werden die markanten Punkte aus allen Kapiteln resümiert. Diese Thesis zeigt, dass das Implementierte Framework zur Interprozesskomunikation zwischen Android-Geräten und Mikrocontrollern in der Praxis eingesetzt werden kann. Dazu wird ein Nachrichtenbasiertes Kommunikationsverfahren verwendet. Die Architektur hat sich trotz einschränkender Eigenschaften von Android, als stabiles Gerüst zur Implementierung von Kommunikationsfunktionen zwischen eng gekoppelten Systemen erwiesen. Der modulare Aufbau gestaltet das System dynamisch und flexibel ohne einen großen Aufwand zum Einbinden in eigene Projekte einzufordern. Die gestellten Anforderungen an das System werden zufriedenstellend erfüllt. Dennoch gibt es an einigen Stellen Verbesserungspotential. So wird beispielsweise nicht die volle Bandbreite der USB-Schnittstelle genutzt. Des weiteren wären andere Mechanismen zur Kommunikation, wie ein RPC-Verfahren oder ein, durch das Framework verwalteter, virtueller Shared-Memory interessant. Die aufgezeigten Verbesserungsmöglichkeiten sind gut für weitere Projektarbeiten geeignet und bilden ein interessantes Forschungsfeld.

A. Inhalt der DVD

Die beiliegende DVD beinhaltet neben dieser Thesis den erstellten Quellcode und weitere Dateien, deren Struktur nachfolgend gezeigt ist.

Code

Android

- **android_stm32f4_ipc**
  Vollständiger workspace der MCU Implementierung
- **APK**
  Installierbare Anwendungspakete für Android

MCU

- **CoIDEworkspace**
  Vollständiger workspace der MCU Implementierung

Images

Bilder die in dieser Thesis verwendet wurden

B. Abkürzungen

ADK Accessory Development Kit

AIDL Android Interface Definition Language

AOA Android Open Accessory

CAN Controller Area Network

FSM finite state machine

GPIO general purpose input/output

GPS global positioning system

GSM global gystem for mobile communications (Mobilfunkstandard 2. Generation)

GUI graphical user interface

IDE integrated development environment

IPC inter-process communication

LTE long term evolution (Mobilfunkstandard 4. Generation)

MCU micro controller unit

NDK native development kit (Android Entwicklungsumbgebung welche nativen Code unterstützt)

NFC near field communication

PID product ID

PWM Pulsweitenmodulation

RFID radio-frequency identification

RPC remote procedure call

RS232 Eine Unterart von UART, definiert Timing und Spannungspegel

RTT round trip time

SPI serial peripheral interface

UART universal asynchronous receiver transmitter

UDP user datagram protocol

UMTS universal mobile telecommunications (Mobilfunkstandard 3. Generation)

URI uniform resource identifier

USB universal serial bus

UTF8 8-Bit UCS (universal character set) transformation format

VID vendor ID

VM virtuelle Maschine

WLAN wireless local area network

Literaturverzeichnis

[1] *STM32F105xx, STM32F107xx, STM32F2xx and STM32F4xx USB On-The-Go host and device library*, 2012. – URL http://www.st.com/st-web-ui/static/active/jp/resource/technical/document/user_manual/CD00289278.pdf. – Zugriff am 2014-09-08

[2] *Android Developers - About*. 2014. – URL http://developer.android.com/about/index.html. – Zugriff am 2014-10-14

[3] *Android Developers - API Guide, BroadcastReceiver*. http://developer.android.com/guide/topics/providers/content-provider-basics.html, 2014. – Zugriff am 2014-11-04

[4] *Android Developers - API Guide, Connectivity*. http://developer.android.com/guide/topics/connectivity/index.html, 2014. – Zugriff am 2014-10-18

[5] *Android Developers - API Guide, Processes and Application Life Cycle*. http://developer.android.com/guide/topics/processes/process-lifecycle.html, 2014. – Zugriff am 2014-11-04

[6] *Android Developers - API Guide, Processes and Threads*. http://developer.android.com/guide/components/processes-and-threads.html, 2014. – Zugriff am 2014-11-05

[7] *Android Developers - BroadcastReceiver*. http://developer.android.com/reference/android/content/BroadcastReceiver.html, 2014. – Zugriff am 2014-11-02

[8] *Android Developers - Parcel*. http://developer.android.com/reference/android/os/Parcel.html, 2014. – Zugriff am 2014-11-02

[9] *Android Developers - The Developer's Guide, Application Fundamentals*. developer.android.com/guide/components/fundamentals.html, 2014. – Zugriff am 2014-10-15

[10] *Android Developers - Tools, Android NDK*, 2014. – URL http://developer.android.com/tools/sdk/ndk/index.html. – Zugriff am 2014-09-10

[11] *Android Developers - Tools, System and Software Requirements*, 2014. – URL https://developer.android.com/tools/sdk/ndk/index.html#Reqs. – Zugriff am 2014-12-07

[12] *Android Developers - Training, Starting an Activity*, 2014. – URL http://developer.android.com/training/basics/activity-lifecycle/starting.html. – Zugriff am 2014-09-10

[13] *Android Open Accessory Protocol 1.0.* https://source.android.com/accessories/aoa.html, 2014. – Zugriff am 2014-11-02

[14] *Android Reference - Binder*, 2014. – URL http://developer.android.com/reference/android/os/Binder.html. – Zugriff am 2014-12-12

[15] *Android Reference - IBinder*, 2014. – URL http://developer.android.com/reference/android/os/IBinder.html. – Zugriff am 2014-12-12

[16] *Android Reference - Intent*, 2014. – URL http://developer.android.com/reference/android/content/Intent.html. – Zugriff am 2014-12-09

[17] *Android Reference - Service*, 2014. – URL http://developer.android.com/reference/android/app/Service.html. – Zugriff am 2014-12-09

[18] *Protocol Buffers - Developer Guide*, 2014. – URL https://developers.google.com/protocol-buffers/docs/overview. – Zugriff am 2014-10-14

[19] *USB Specification*, 2014. – URL http://www.usb.org/developers/docs/. – Zugriff am 2014-10-27

[20] Andrew S. Tanenbaum, Marten van S.: *Verteilte Systeme Prinzipien und Paradigmen.* Pearson Studium, München, 2008 (2. Auflage)

[21] Axelson, Jan: *USB Complete Fourth Edition: The Developer's Guide.* Lakeview Research, Madison (Wisconsin), 2009 (4. Auflage)

[22] Beretta, Michele: *Android ADK application for STM32 RS232 - client for Android.* September 2013. – URL http://home.deib.polimi.it/bellasi/lib/exe/fetch.php?media=students:beretta_finalreport.pdf. – Zugriff am 2014-09-04

[23] EDZIEJEWSKI, Michal S.: *RPC for Embedded Systems*. Warschau, Master Thesis, Oktober 2010. – URL http://robotyka.ia.pw.edu.pl/twiki/pub/Publications/AllTheses/docent-msc-2010.pdf

[24] HSIEH, Cheng-Kang ; FALAKI, Hossein ; RAMANATHAN, Nithya ; TANGMUNARUNKIT, Hongsuda ; ESTRIN, Deborah: Performance Evaluation of Android IPC for Continuous Sensing Applications. In: *SIGMOBILE Mob. Comput. Commun. Rev.* 16 (2013), Februar, Nr. 4, S. 6–7. – URL http://doi.acm.org/10.1145/2436196.2436200. – ISSN 1559-1662

[25] KNUTH, Donald E.: *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997. – ISBN 0-201-89684-2

[26] PRESS, William H. ; TEUKOLSKY, Saul A. ; VETTERLING, William T. ; FLANNERY, Brian P.: *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. New York, NY, USA : Cambridge University Press, 1992. – ISBN 0-521-43108-5

[27] SCHREIBER, Thorsten: *Android Binder*. 2011

[28] STALLINGS, William: *Data and Computer Communications*. Pearson Education, New Jersey, 2010 (Ninth Edition). – ISBN 978-0-13-217217-2

[29] WOLF, Jürgen: *Linux-UNIX-Programmierung Das umfassende Handbuch*. Galileo Press, Bonn, 2005. – URL http://openbook.galileo-press.de/linux_unix_programmierung/index.htm

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 9. Januar 2015 Michel Rottleuthner