Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Wassilij Beaucamp

Framework für Android/MCU-basierte Smart Systems

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Wassilij Beaucamp

Framework für Android/MCU-basierte Smart Systems

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.-Ing Andreas Meisel

Eingereicht am: 27. Oktober 2017

Wassilij Beaucamp

Thema der Arbeit
Framework für Android/MCU-basierte Smart Systems

Stichworte
Android, Mikrocontroller, Smart-System, Framework, MCU, Kommunikation

Kurzzusammenfassung
Android bietet eine gute Plattform für autonome Systeme wie Smartphones und Tablets. Sie sind autonom, weil sie lange Batterielaufzeiten und Sensorkomponenten bieten. Mit diesen Eigenschaften ist es möglich, intelligente Applikationen zu entwickeln. Das Ziel dieser Thesis ist es, diese Eigenschaften als Plattform für das zentrale Nervensystem eines mobilen Smart-Systems zu nutzen, indem ein Mikrocontroller mit einem Smartphone verbunden wird. Der Mikrocontroller wird für die Kontrolle der Peripherie benutzt. Der Kamerasensor wird für die Erkennung einer Linie, der gefolgt werden soll, genutzt. Die Kommunikation zwischen Mikrocontroller und Android sowie die Ansteuerung der Motoren wird als Framework implementiert.

Wassilij Beaucamp

Title of the paper
Framework for Android/MCU-based Smart Systems

Keywords
Android, microcontroller, Smart-System, framework, MCU, communication

Abstract
Android offers a great platform for autonomous systems in the way of smartphones or tablets. They are autonomous in the sense of offering great battery usage and sensors to base smart apps on. The goal of this thesis was, to use that platform as the brain of a mobile, smart system by connecting a smartphone to a microcontroller. The microcontroller is used to control the periphery needed to make the system mobile. The camera is used as a sensor to detect a line that needs to be followed. The communication with the microcontroller and the actuation of the motors is fitted to be a framework and offer some configurability.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Der Begriff Framework wie folgt definiert: „A framework is a semi-complete application. A framework provides a reusable, common structure to share among applications. Developers incorporate the framework into their own application and extend it to meet their specific needs“ ([22], Seite 4). Auf der Grundlage dieser Definition soll, in dieser Arbeit, ein Framework für MCU/Android basierte mobile Smart-Systems entwickelt werden. Ein intelligentes, mobiles System ist ein System, das auf durch Sensoren erfasste Daten eine Reaktion in Form einer Bewegung folgen lässt.
Die Mikrocontollereinheit (MCU) soll hierbei die Kontrolle über den mobilen Teil des Systems haben. Die Mobilität soll durch einen Model-Truck geliefert werden, der durch seine Motoren Einfluss auf die Bewegung des Gesamtsystems nimmt.
Der smarte Teil des Systems wird abgedeckt von einem Android-Smartphone. Viele Smartphones liefern Hardware, die sich für den Einsatz in intelligenten Systemen eignet. Das Android-Betriebssystem bietet eine weit verbreitete und somit wohlbekannte Umgebung, um die Hardware zu benutzen. Sensoren wie Kamera, GPS-Module oder Gyroskop-Sensoren eignen sich hervorragend für die Orientierung im Raum und sind auf fast allen aktuellen Smartphones zu finden. Auch sind viele Geräte dazu in der Lage, untereinander oder mit anderen Geräten über Wege wie WLAN, USB, UMTS oder Bluetooth zu kommunizieren und eignen sich somit für den Einsatz in einem Netzwerk. Deshalb soll hier getestet werden, ob ein Android-Gerät als Grundlage für ein Smart-System dienen kann.
In der Bachelor-Ausbildung an der Hochschule für angewandte Wissenschaften Hamburg ist Java die erste Programmiersprache, die gelehrt wird. Im Fachbereich der Technischen Informatik ist es besonders schwierig, schnell einen Bezug zu den eigentlichen Aufgabenbereichen eines technischen Informatikers herzustellen. In dieser Bachelorarbeit soll versucht werden, einen schnelleren und einfacheren Einstieg in die Robotik, einen Teilbereich der Technischen Informatik, zu finden. Das Android-Betriebssystem erlaubt es, in Java zu programmieren. Es könnte somit eventuell für einen Programmieranfänger nützlich sein, mit einem geeigneten Framework erste Erfahrungen an Robotern zu sammeln.

2 Grundlagen

2.1 Android

Android ist eine Software-Plattform für mobile Endgeräte wie zum Beispiel Fahrzeug-Multimedia-Systeme, Wearable-Computer, Fernseher und Smartphones. Entwickelt wird das Betriebssystem von der Open Handset Alliance, die von Google im Jahr 2007 mit 33 Partnern gegründet wurde [1]. Das Konsortium setzt sich für das Erstellen eines offenen Standards für Mobilgeräte ein. Android ist, bis auf den Linux-Kernel, vollkommen quelloffen und steht zur freien Nutzung zur Verfügung. Dieses Konzept hat dazu geführt, dass Android Anfang 2017 am Markt für Mobiltelefone einen Marktanteil von 85% gehalten hat [18].

2.1.1 Architektur

Die Grundlage der Android Software-Plattform ist der Linux-Kernel. Die Android Runtime (ART) zum Beispiel verlässt sich auf den Kernel in zugrundeliegenden Funktionalitäten wie Threads und Speicher-Verwaltung.
Die Hardware Abstraction Layer (HAL) stellt Schnittstellen zu der Hardware des genutzten Geräts zur Verfügung, mit Hilfe dieser können Applikationen aus dem Java API Framework die Hardware ansteuern. Die HAL besteht aus einzelnen Bibliotheksmodulen, die jeweils eine Hardware-Komponente wie Kamera, Bluetooth oder Audio widerspiegeln.
Auf Android-Geräten, die die Version 5.0 (API Level 21) oder höher nutzen, läuft jede Applikation in einem eigenen Prozess und mit einer eigenen Instanz der Android Runtime. Der größte Unterschied zur vorhergegangenen Dalvik Laufzeitumgebung ist, dass die Dalvik Executable (DEX) Dateien nun Ahead of Time (AOT) zur Installationszeit anstatt Just in Time (JIT), also zur Laufzeit, kompiliert werden.
Viele der Kernkomponenten von Android, wie zu Beispiel ART und HAL, erfordern die Nutzung von nativen Bibliotheken, die in C oder C++ programmiert sind. Die Android-Plattform bietet Java Programmierschnittstellen, die die Funktionalität einiger der nativen Bibliotheken bereitstellen. Falls die entwickelte Applikation direkt C- oder C++-Code ausführen muss, kann das Android Native Development Kit benutzt werden.

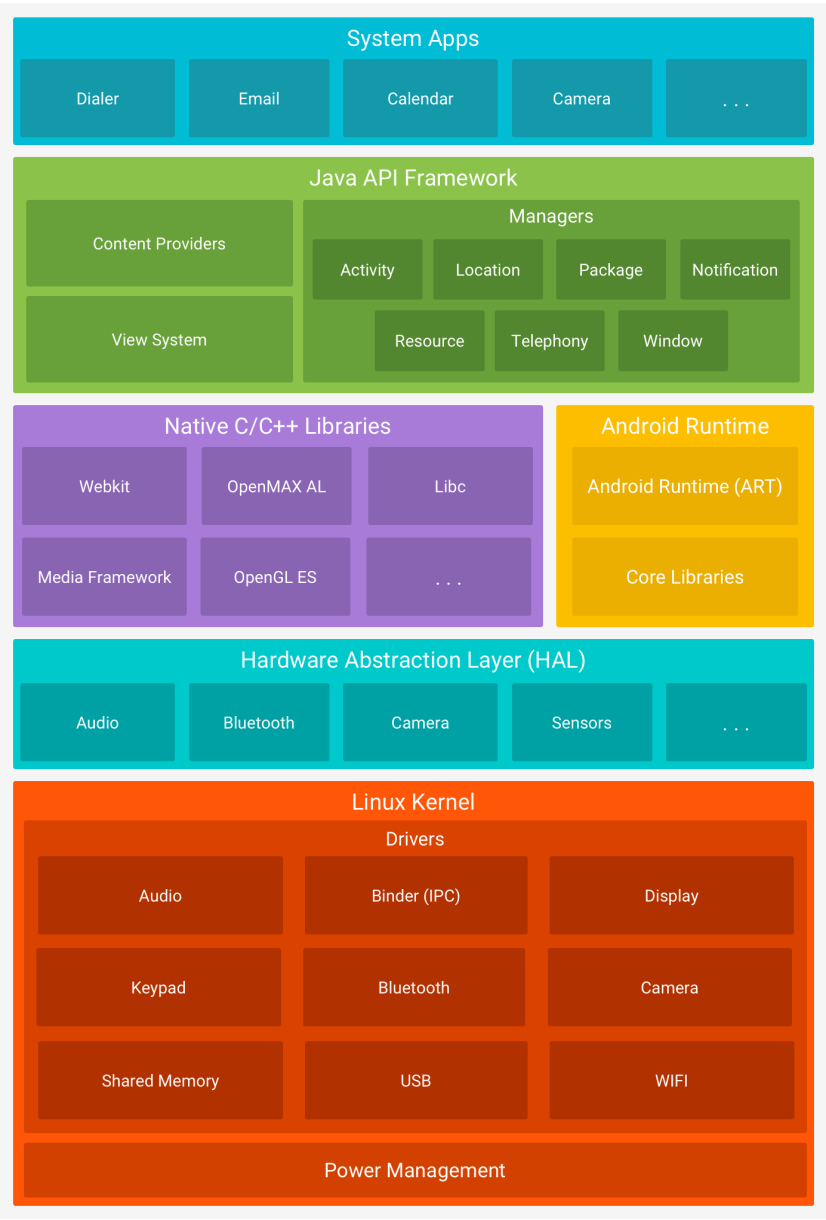


Abbildung 2.1: Der Android Software-Stack

Alle Eigenschaften des Android Betriebssystems sind durch in Java geschriebene Programmierschnittstellen erreichbar. Diese Schnittstellen bilden die Bausteine, die benötigt werden, um eine Android-Applikation zu programmieren, sie vereinfachen das Wiederverwenden von modularen Kernkomponenten des Systems und Systemdiensten. Im Folgenden sollen einige Systemelemente genauer beschrieben werden.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
        "http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical" >
        <TextView android:id="@+id/text"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="I_am_a_TextView" />
        <Button android:id="@+id/button"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="I_am_a_Button" />
</LinearLayout>
```

Listing 2.1: Beipiel einer XML Layout Beschreibung

Das **View System** wird benutzt, um die Benutzeroberfläche einer App zu erstellen. Alle Elemente der Oberfläche sind *View*- oder *ViewGroup*-Objekte. Ein *View* ist ein Element, das etwas auf dem Bildschirm abbildet, mit dem der Benutzer interagieren kann (vgl. [24], Seite 192). Eine *ViewGroup* fasst andere ViewGroups und Views zu einem Objekt zusammen (vgl. [20]). Der übliche Weg, das Layout einer Applikation zu definieren, ist, es in einer XML-Datei zu beschreiben (Listing 2.1).

Content Provider dienen dazu, Daten zwischen verschiedenen Applikation auszutauschen, Dateien aus einer Anwendung heraus zu speichern oder Daten von anderen Applikationen zu laden (vgl. [24], Seite 92). Die Content Provider kapseln die Daten ein und stellen Mechanismen zur Verfügung, um Datensicherheit zu gewährleisten (vgl. [14]).

Zusätzlich zu den schon genannten Elementen des Java API Frameworks existieren noch eine Reihe von Managern, die Funktionen wie Notification- und Locationmanagement anbieten. Zwei dieser Manager sollen nun näher vorgestellt werden.

Der **Resource Manager** stellt Elemente zu Verfügung, die kein Code sind. Hierzu gehören Abbildungen, Layout-Dateien oder Texte. Es ist wichtig, diese Ressourcen vom Applikationsco-

de zu separieren, da vor allem Layouts auf den vielen Geräten, auf denen Android eingesetzt wird, sich den Umgebungsvariablen wie zum Beispiel Displaygröße oder Displayausrichtung anpassen müssen. Für alle Ressourcen sollte eine Standard- und mehrere Alternativkonfigurationen spezifiziert sein. Die Standardkonfiguration wird immer dann eingesetzt, wenn die Geräteumgebung keinen Einfluss auf die Ressource nimmt oder wenn keine Alternativkonfiguration vorhanden ist (vgl. [17]).
Der **Activity Manager** bestimmt über den Lebenszyklus einer Activity. Activities sind der grundlegende Baustein jeder App. Im Gegensatz zu dem üblichen Programmierparadigma, dass jedes Programm über eine *main()*-Methode gestartet wird, werden in Android Instanzen einer Activity-Komponente mit spezifischen Callback-Methoden aufgerufen. Diese Callbacks korrespondieren mit den Stadien des Lebenszyklus einer Activity. Die Activity-Klasse dient als Einstiegspunkt für das Interagieren mit dem Benutzer, sie liefert ein Fenster in welchem die Applikation die Benutzeroberfläche zeichnen kann. Üblicherweise füllt das Fenster den gesamten Bildschirm, allerdings gibt es auch Activities, die andere Fenster nur teilweise überdecken. Eine App besteht typischerweise aus mehreren Activities, so kann es zum Beispiel eine Activity zum Verändern von Einstellungen geben oder eine andere Activity bietet die Möglichkeit an, eine Email zu verfassen. Generell sollte es immer eine Activity geben, die als erstes auf dem Bildschirm angezeigt wird, diese wird als *main activity* bezeichnet. Um Activities innerhalb einer Applikation benutzen zu können, müssen sie im Manifest deklariert werden (vgl. [15]). Der Kernsatz an Callbacks, der von der Activity-Klasse implementiert werden muss, sind sechs Methoden: *onCreate()*, *onStart()*, *onResume()*, *onPause()*, *onStop()* und *onDestroy* (vgl. [19]). Diese werden immer dann vom System aufgerufen, wenn die Activity einen neuen Zustand annimmt. Dieses Verhalten wird in Abbildung 2.2 dargestellt.

2.1.2 Rechteverwaltung

Jede Applikation auf einem Android-Gerät läuft wird im zugrundeliegenden Linux-Kernel als ein Benutzer angesehen. Durch diese Eigenschaft können die in Linux verwendeten Nutzerzugriffsrechte angewendet werden, um jeder App eigene Berechtigungen zuzuteilen. Berechtigungen beinhalten neben reinen Dateizugriffsrechten auch weiter eingreifende Rechte wie Zugriff auf Komponenten wie Bluetooth, Kontakte, Nachrichten oder Kamera. Für Applikationen die Android Version 5.1.1 (API Level 21) oder niedriger als Zielversion benutzen, werden Rechte bei der Installation eingeholt. App mit der Android-Zielversion 6.0 (API Level 23) oder höher holen die benötigten Berechtigungen während der Laufzeit ein. Das erlaubt dem Nutzer die vergebenen Berechtigungen zu jeder Zeit wieder zurückzuziehen. Dies war vorher nur mit der Deinstallation der Applikation möglich (vgl. [16]). Die Berechtigungen sind nicht

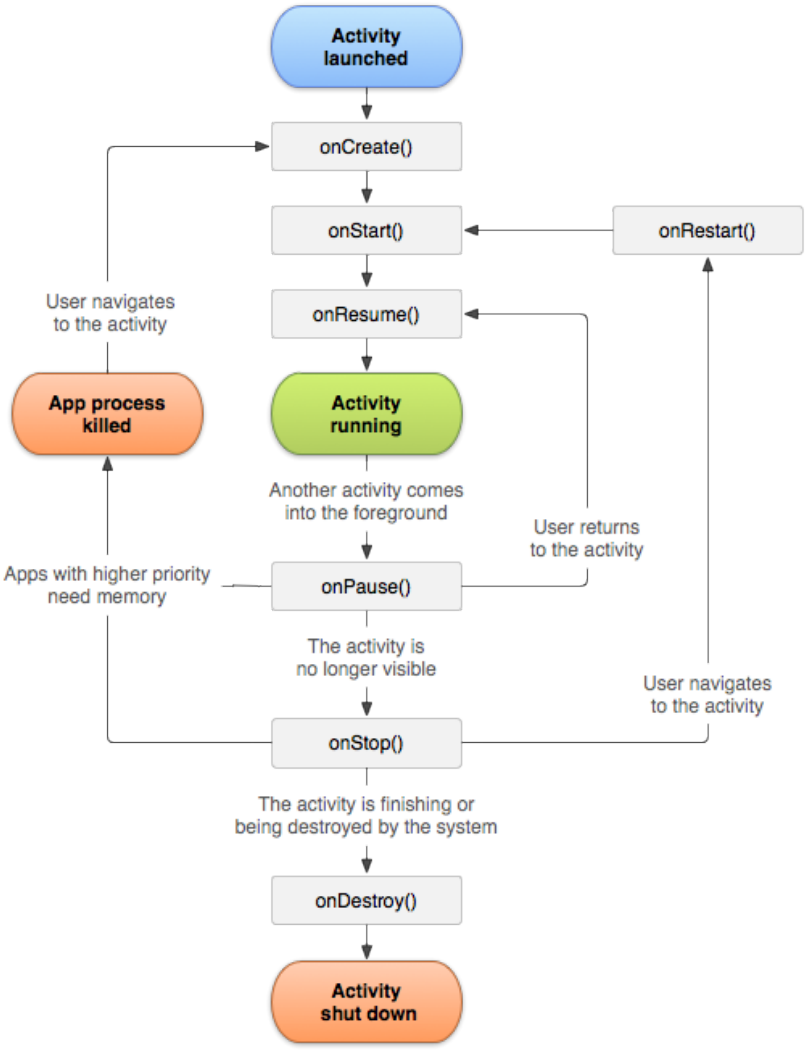


Abbildung 2.2: Activity Lebenszyklus

beschränkt auf jene des Betriebssystems, sondern es können auch eigene Berechtigungen formuliert werden.

2.2 Hardware

Bei dem Android-Gerät handelt es sich um ein Nexus 7 Tablet mit Android Version 6.0.1 (API Level 23). Jedoch spielt das ausgewählte Android-Gerät keine große Rolle, da sich Abhängigkeit und Kompatibilität fast ausschließlich durch das Betriebssystem ergeben. Für dieses Projekt ist mindestens API Level 21 zur Nutzung der Camera2 API notwendig.
Bei der Umsetzung der Thesis von Michel Rottleuthner wurde ein STM32F4DISCOVERY Entwicklerboard benutzt. Das Board basiert auf einer ARM Cortex-M4 Architektur und „bietet 1 MB Flash, 192KB RAM und ein integriertes ST-LINK/V2-Modul zum Flashen und Debuggen “ (siehe [25], Kapitel 2.4). Es wurde sich für dieses Board entschieden, da es in vielen Projekten und Praktika an der HAW Hamburg eingesetzt wird. In dieser Arbeit wurde das gleiche Board (STM32F407) benutzt, um möglichen Problemen mit unterschiedlichen Versionen vorzubeugen.

2.3 Werkzeuge

In der Bachelorarbeit von Michel Rottleuthner wurden zwei Entwicklungsumgebungen benutzt. Für die Programmierung des Mikrocontrollers wurde CoIDE von CooCox ausgewählt und wird auch in diesem Projekt benutzt, um eventuelle Komplikationen mit dem Wechsel zu einer anderen Entwicklungsumgebung zu vermeiden. Für die Programmierung des Android-Gerätes wurde in der vorhergegangenen Thesis Android Developer Tools benutzt, da Android Studio zum Zeitpunkt der Werkzeugwahl nur in einer Betaversion zu Verfügung stand. Für diese Arbeit ist Android Studio benutzt worden, da es nun die offizielle IDE (Integrated Development Enviroment) für Androidentwicklung ist. Die Umgebung bietet einfachen Zugriff auf aktuelle und alte Androidversionen. Zusätzlich können über den Android Virtual Device Manager einfach virtuelle Geräte erstellt und zu Debuggingzwecken genutzt werden (vgl. [24], Seite 31).

3 Anforderungen

In diesem Kapitel werden die Anforderungen an das System aus Mikrocontroller und Android-Gerät spezifiziert, die das zu entwickelnde Framework erfüllen soll. Um dies tun zu können muss zuerst der Einsatzbereich eingegrenzt und beschrieben werden. Aus dieser Erläuterung werden dann funktionale und nicht funktionale Anforderungen abgeleitet.

3.1 Einsatz

Das System auf dem das Framework eingesetzt wird ist bekannt. Es handelt sich um eine Kombination aus einem ferngesteuerten Modell-Lastwagen und einem Android-Smartphone. Der Modell-Lastwagen wird gesteuert von einer Mikrocontroller-Unit (MCU), die MCU ist auch dafür zuständig Nachrichten vom Android-Gerät zu empfangen und in Steuersignale umzuwandeln. Das System ist somit wenig dynamisch, da alle Komponenten schon im Voraus bekannt sind und Funktionen schon zur Zeit des Kompilierens festgelegt sind. Die Ansteuerung des Modell-Trucks kann im Prinzip auf zwei Servomotoren minimiert werden. Ein Servomotor dient zur Bestimmung der Geschwindigkeit, der andere zur Einstellung des Lenkwinkels.

3.2 Funktionale Anforderungen

3.2.1 Dynamik

Zwar ist das System zur Zeit der Kompilierung bekannt, allerdings soll es die Möglichkeit geben eventuelle Systemänderungen zu unterstützen. Somit muss es einfach sein, die Software an Änderungen in der Hardware anzupassen. Für diesen Zweck soll das System dazu in der Lage, sein dynamisch auf Änderungen der Motorenanzahl oder -funktion zu reagieren.

3.2.2 Sicherheit

Da es sich bei dem System um ein mobiles System handelt, muss dafür gesorgt sein, dass keine Personen oder Gegenstände zu Schaden kommen. Das heißt es muss eine Möglichkeit geben, das System zu jeder Zeit in einen sicheren Zustand zu versetzen.

3.2.3 Kommunikation

Beide Einzelsysteme (Android-Gerät und Mikrocontroller-Unit) müssen dazu in der Lage sein, bidirektional miteinander zu kommunizieren. Hierfür soll die im Rahmen einer früheren Bachelorarbeit [25] entwickelte Interprozesskommunikation zwischen Android- und Mikrocontrollersystemen genutzt werden.

3.3 Nicht Funktionale Anforderungen

3.3.1 Kompatibilität

Besonders im Rahmen des Android-Betriebssystems muss darauf geachtet werden, dass das Framework mit den vielen Geräten und Versionen, die das Betriebssystem verwenden, kompatibel ist. Die Seite des Mikrocontrollers ist weniger stark von diesem Problem betroffen, allerdings muss auch hier darauf geachtet werden, dass weit genug abstrahiert wird, sodass die Portierung auf unterschiedliche Hardware möglich bleibt.

3.3.2 Benutzbarkeit

Das Framework soll dazu dienen Programmieranfänger mit Hilfe von Java an den Bereich der Robotik heranzuführen. Um die Basisfunktionalität anwenden zu können darf nur Programmierung in Java vonnöten sein. Das bedeutet auch, dass kein Wissen über das Android-Betriebssystem notwendig sein muss, um mit dem Framework umgehen zu können. Weiterhin soll die Einarbeitung in die vorhandene Materie einfach vonstatten gehen.

3.3.3 Modularität

Um Wiederverwendbarkeit und Erweiterungen zu ermöglichen, sollte darauf geachtet werden, dass Aufgabenbereiche einzelner Module klar definiert sind.

3.3.4 Performanz

Besonders an die Kommunikationskomponente zwischen Android und MCU ist die Anforderung gestellt, schnell und zuverlässig Daten zu übertragen. Sie muss nicht dazu in der Lage sein große Datenpakete zu versenden, aber es muss möglich sein viele kleine Pakete schnell genug austauschen zu können. Dies stellt besonders hohe Anforderungen an die Effizienz des genutzten Nachrichtenprotokolls.
Außerdem muss die Bildverarbeitung in der Beispielapplikation schnell genug Lenkdaten

errechnen, um eine sichere Steuerung des Modell-Fahrzeugs zu gewährleisten. Zusätzlich darf die Anwendung nicht zu viel Rechenzeit benötigen, sodass den anderen Android-Apps genügend Ressourcen zur Verfügung stehen.

4 Analyse

In diesem Abschnitt soll ein präziser Blick auf die gestellten Anforderungen geworfen werden. Es werden Verfahren und Techniken vorgestellt die dazu beitragen können, das gestellte Problem zu lösen. Als erster Schritt wird die vorgegebene Interprozesskommunikation analysiert und bewertet, um festlegen zu können, ob diese sich für den gegebenen Anwendungsfall eignet. Daraufhin wird auf die Komponenten des Mikrocontrollers und Androids eingegangen, die benötigt werden, um die gewünschte Funktionalität zu erreichen.

4.1 Interprozesskommunikation

Der hardwaretechnische Aufbau der früheren Bachelorarbeit ähnelt sehr stark der Hardwareauslegung, die bei diesem Projekt umgesetzt werden könnte. Es handelt sich um eine MCU die über USB mit einem Android-Smartphone verbunden ist. Technisch gesehen besteht die Möglichkeit, auch andere Übertragungswege wie zum Beispiel Bluetooth zu verwenden. Allerdings wurde USB mit guter Begründung ausgewählt. Vor allem die hohe Verbreitung und vielseitige Anwendbarkeit wurden in der vorhergegangenen Arbeit als die entscheidenden Faktoren genannt. Aber auch die Zuverlässigkeit und die geringen Kosten gegenüber drahtlosen Verbindungen wurden in Erwägung gezogen(vgl. [25], Kapitel 4.3).

4.1.1 Performanz

Die verwendete Hardware (STM32F4-DISCOVERY) unterstützt nur die sogenannte Full-Speed Geschwindigkeit (12Mbps) in vollem Maße. In der Arbeit wird auf die Geschwindigkeit der Übertragung eingegangen und es muss evaluiert werden, ob die zur Steuerung notwendigen Übertragungsgeschwindigkeiten erreichbar sind. Aus der Bachelorarbeit von Michel Rottleuthner geht hervor, dass eine unidirektionale Nachricht vom Android-Gerät zum μC in etwa 0,78 ms übertragen wird ([25], Kapitel 8.1). Dieser vergleichsweise geringe Wert legt die Vermutung nahe, dass es an der Stelle der Kommunikationskomponente nicht zu Zeitproblemen kommen sollte, da im Rahmen dieses Projektes eine zeitaufwendige Bilderkennung durchgeführt werden

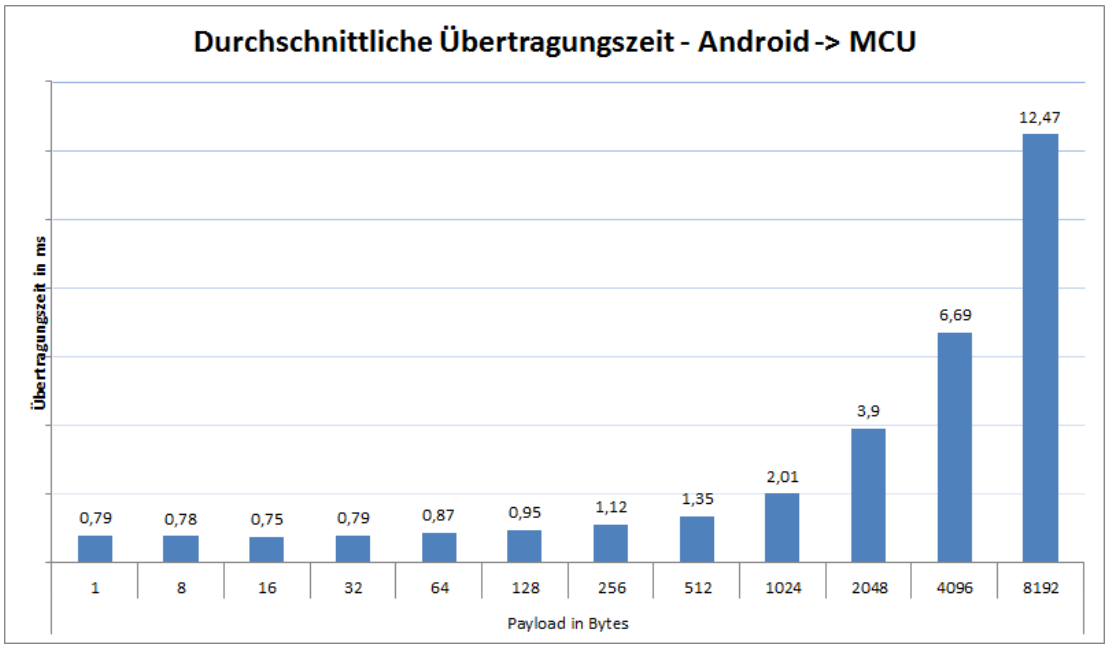


Abbildung 4.1: Durchschnittliche Übertragungszeit von Android zu MCU

soll. Allerdings gilt dieser Wert nur für Nachrichten die eine Payload von weniger als 32 Bytes besitzen. Für Nachrichten bis zu einer Größe von 1024 Bytes erscheint die durchschnittliche Übertragungszeit noch geeignet zu sein für die angestrebte Verwendung. Demgegenüber ist in Abbildung 4.1 zu erkennen, dass es zu erheblichen Performanzverlusten bei Datenpaketen kommt, die die Größe von 1024 Bytes überschreiten. Der Verlust wird vor allem mit Android internen Prozessen begründet, da die Speicherallokation von ursprünglich 4KB erneuert werden muss.

Ein Problem das beim Lesen der Bachelorarbeit von Michel Rottleuthner auffällt sind die maximalen Übertragungsraten besonders im Bereich von Nachrichten kleiner als 1024 Bytes (siehe Abbildung 4.2). Die Sprunghaftigkeit der Werte in diesem Bereich kann auf ungenügend ausführliche Testreihen zurückgeführt werden. Zusätzlich wird gesagt, dass diese starken Schwankungen nur selten auftreten(vgl. [25], Kapitel 8.1). Aufgrund des seltenen Auftretens werden die Latenzen nicht als Problem erachtet.

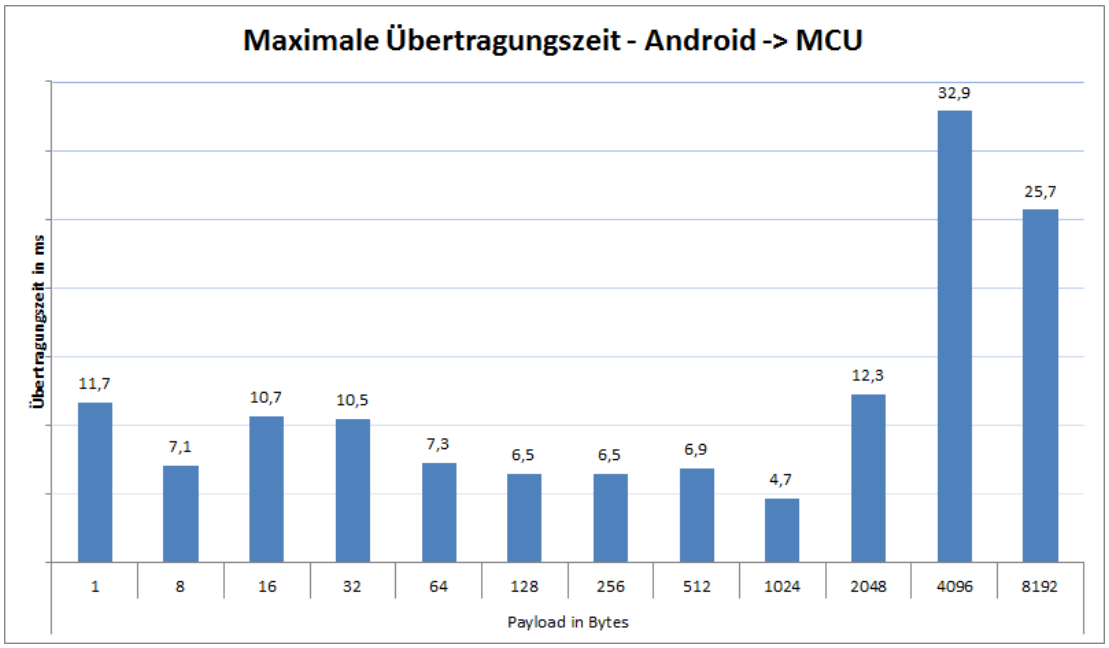


Abbildung 4.2: Maximale Übertragungszeit - Android -> MCU

4.1.2 Verwendung

Die logische Struktur des Rottleuthner Projektes leitet sich aus dem Aufbau der Hardware ab. Die beiden Systeme, die kommunizieren, sind grundlegend verschieden. Als erster grundlegender Unterschied werden die verschiedenen Programmiersprachen aufgezählt. Auf dem Mikrocontroller kommen zur Implementierung C und C++ in Frage. Auf dem Android-Gerät wird hauptsächlich in Java programmiert. Es gibt zwar die Möglichkeit mit dem Android NDK (Native Development Kit) in C oder C++ zu programmieren, allerdings erhöht eine Implementierung mit Hilfe des Toolsets die Komplexität der App. Der erhöhten Komplexität steht nur eine geringe Leistungssteigerung gegenüber (vgl. [25], Kapitel 5.1). Außerdem geht mit der Verwendung des Android NDK die Hardwareunabhängigkeit, die Android verspricht verloren. Als zweiter Grund für die Entwicklung von zwei Teilsystemen wird das unterschiedliche Laufzeitverhalten zwischen Mikrocontroller-Unit und Android genannt, das daher rührt, dass der μC ohne und das Android-Gerät, mit Betriebssystem arbeitet. Zusätzlich vergrößert der erhebliche Unterschied von bereitstehenden Ressourcen die Differenz in Verhalten. Bei der Kommunikationskomponente aus der vorgegebenen Bachelorarbeit handelt es sich um einen eigenständigen Dienst, der von den verschiedenen Prozessen, die kommunizieren wollen, parallel genutzt wird (siehe Abbildung 4.3). Über einen Service Access Point (SAP) können

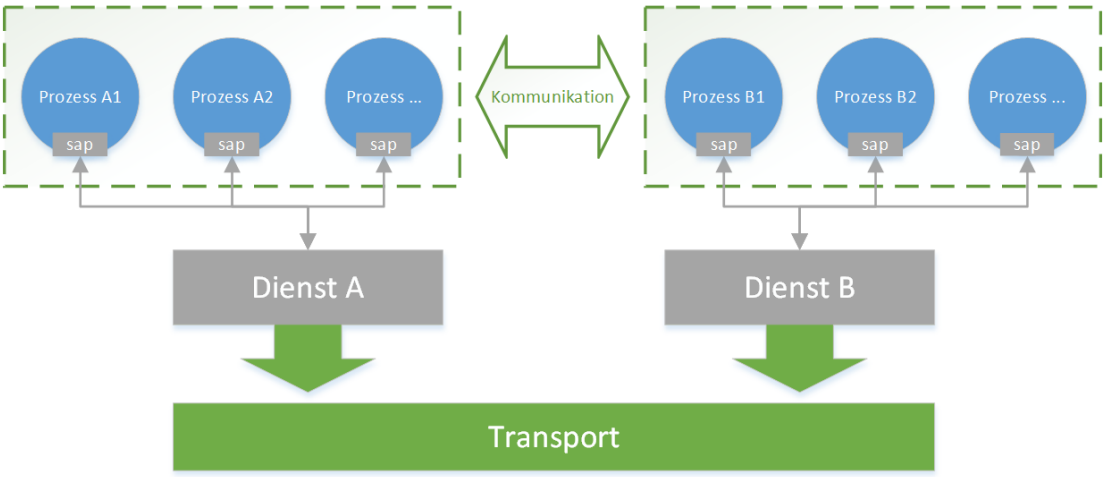
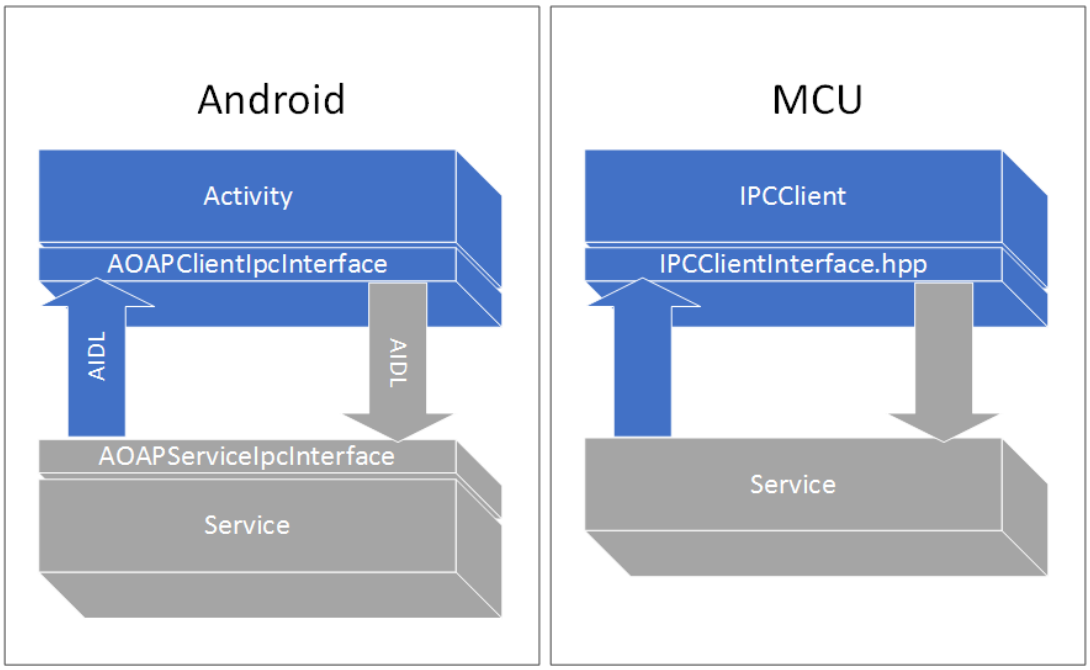


Abbildung 4.3: Überblick des angebotenen Service

die einzelnen Prozesse auf beiden Seiten auf den Kommunikationsdienst zugreifen und somit miteinander Daten austauschen. Auf Seiten des Mikrocontrollers wird die ST USB Bibliothek, erweitert mit einigen Funktionen aus dem Hello ADK Projekt, genutzt. Auf der Gegenseite wird die Android Open Accessory Bibliothek verwendet, um die USB Verbindung anwendbar zu machen. Zur effizienten Serialisierung der Daten werden die von Google entwickelten Protocol Buffers genutzt. Auf Mikrocontrollerebene kommt nanoPB, eine besonders speicherunintensive Variante, zum Einsatz.

Um die Transportschicht nutzen zu können, muss das TransportEndpointInterface angesprochen werden, für die Kommunikation über USB existiert die Klasse OpenAccessoryUSBTransport, die das Interface implementiert. Im Kern der Kommunikationsframeworks auf der Android-Seite steht der AOAPService. Über eine AIDL Schnittstelle findet die Kommunikation zwischen dem Service und den lokalen Prozessen statt. Für Anwendungen auf dem Android-Gerät dient die Klasse ServiceBridge als Kontaktpunkt zur Kommunikationskomponente. Die ServiceBridge wird als Helfer-Klasse eingesetzt um den Handshake mit dem Service zu unterstützen. Damit die ServiceBridge mit der Anwendung kommunizieren kann und Nachrichten weitergeleitet werden können, wird hier eine Referenz, die das AOAPClientInterface implementiert, übergeben. Um eine Verbindung mit dem Service herzustellen muss nur die Methode *connectToService()* aufgerufen werden.

Auf Seite des Mikrocontrollers wird kein zusätzlicher Mechanismus zur lokalen Datenübertragung benötigt, da die Clients direkt mit einer Referenz des Services arbeiten können. Objekte die mit dem Service kommunizieren, müssen das IPCClientInterface implementieren. Um Nachrichten senden zu können, muss die Methode *process()* vorhanden sein. Die Funktion



(a) Teilsystem: Android (b) Teilsystem: MCU

Abbildung 4.4: Kommunikationskomponenten der beiden Teilsysteme

wird zyklisch aufgerufen. Da kein Betriebssystem existiert, muss sich jeder Client kooperativ verhalten. Das heißt die *process()* sollte keine lang andauernden Operationen beinhalten.

4.2 Mikrocontroller

Das Framework, das in der Arbeit von Michel Rottleuthner erarbeitet wurde, verwendet eine Mikrocontrollerunit der Firma STMicroelectronics. Es ist nicht Teil dieser Arbeit, das Kommunikationsframework in allzu großem Maße zu verändern. Deshalb wird auch in diesem Projekt eine MCU der genannten Firma verwendet werden. Die meisten Geräte des Herstellers bieten genügend Ressourcen, um die gestellten Anforderungen zu erfüllen. So werden zum Beispiel Zeitgeberbausteine gebraucht, um die Motoren mit PWM-Signalen versorgen zu können. Zusätzlich können über verschiedene Wege Sensoren ausgelesen werden, einige Möglichkeiten wären SPI oder I2C Bussysteme. Es muss außerdem eine Methode zum Anschließen der geforderten Fernsteuerung geben. Hierfür könnte sich USART als geeignet erweisen. Auch die erforderliche USB-Klinke wird von dem Großteil der Geräte unterstützt. Einen Vorteil, den die zu verwendenden Mikrocontroller mitbringen, sind die Standard Peripheral Libraries für ST

Geräte. Sie ermöglichen die Portierung des Programms auf verschiedenen MCUs.
Auf Ebene der Software müssen Schnittstellen zu Motoren, Fernbedienung und der Kommunikationskomponente geschaffen werden. Für die Fernsteuerung wird ein Gerät der Firma Spektrum verwendet. Hauptsächlicher Grund für die Verwendung ist das vom Hersteller offengelegte und dokumentierte Protokoll (DSMX) der Kommunikation zwischen einem Remote Receiver und der Fernbedienung. Es handelt sich um eine Fernbedienung der Reihe Spektrum DX8 und einen Remote Receiver DSMX. Mit dieser Kombination aus Geräten gelingt es, die Anzahl von genutzten Pins am Mikrocontroller auf einen Einzelnen zu reduzieren. Beide Geräte sind zu weit mehr in der Lage als für dieses Projekt notwendig ist, allerdings standen beide zur sofortigen Verwendung zu Verfügung und somit kommen sie hier zum Einsatz. Die Fernbedienung ist dazu in der Lage bis zu acht Kanäle (vgl. [21]) gleichzeitig zu bedienen, jedoch sind für dieses Projekt nur zwei Kanäle (Geschwindigkeit und Lenkung) für die Ansteuerung des Model-Trucks notwendig.
Der Remote Receiver operiert bei einer Geschwindigkeit von 125000 Bits pro Sekunde. Ein Datenpaket ist 16 Byte groß und wird je nach dem ausgewählten Protokoll alle 11ms oder 22ms gesendet. Die Daten werden über USART mit einem Stopbit und keinem Start- oder Paritätsbit übertragen (vgl. [23]). Um den Receiver in den Binden-Modus zu bringen, müssen innerhalb von 200ms nach dem Anschalten eine Reihe von fallenden Flanken empfangen werden. Die Anzahl der benötigten Impulse kann aus Tabelle 4.1 entnommen werden. Für dieses Projekt

Pulse	Modus	Protokolltyp
7	Internal	DSMx 22ms
8	External	DSMx 22ms
9	Internal	DSMx 11ms
10	External	DSMx 11ms

Tabelle 4.1: Mögliche Modi der DSMX - Bindung

wird der externe Empfänger in den Modus Internal DSMx 11ms versetzt. Internal bedeutet in diesem Fall, dass dieser Empfänger alleinstehend arbeitet. In diesem Modus werden die Daten wie in Listing 4.1 übertragen. Das Feld *system* zeigt das ausgewählte Protokoll an. *fades* zählt die Anzahl der verpassten Datenpakete, diese Zahl wird erst erneuert, nachdem ein neues Paket empfangen werden konnte. Die restlichen 14 Byte enthalten die Steuerinformationen. Jeweils 2 Byte repräsentieren einen Kanal, wobei die Bits Null bis Zehn die Servoposition zeigen und die Bits 11 bis 14 die Kanalidentifikationsnummer darstellen (Tabelle 4.2). In einem Datenpaket, das vom Remote Receiver an den Mikrocontroller übertragen wird, ist nicht genügend Platz, um alle Kanäle zu bedienen, somit muss von der Software für jedes Paket zuerst

ID	Name
0	Throttle
1	Aileron
2	Elevator
3	Rudder
4	Gear
5	Aux 1
6	Aux 2
7	Aux 3
8	Aux 4
9	Aux 5
10	Aux 6
11	Aux 7

Tabelle 4.2: Kanalidentifikationsnummern

die Kanalidentifikationsnummer überprüft werden. Es kann nicht davon ausgegangen werden, dass sich die Kanäle immer an der selben Position im Array befinden.

```
typedef struct
{
uint8_t system;
uint8_t fades;
unsigned short int servo[7];
} INT_REMOTE_STR_t;
```

Listing 4.1: Internal DMSx 11ms - Daten Format

4.3 Kamera unter Android

Um die Funktionsfähigkeit des entwickelten Frameworks zu zeigen, sollen Steuerinformationen aus Aufnahmen mit der Kamera generiert werden. Das Rottleuthner-Projekt benutzt Android 4.2.2 (API Level 17), mit API Level 21 ist eine neue Camera API (android.hardware.camera2) hinzugefügt worden. Mit dem Hinzufügen der neuen, gilt die vorhergegangene Schnittstelle (android.hardware.camera) als veraltet. Die Camera2-API soll verwendet werden, um Bildinformationen vom Kamerasensor zu erhalten und darauf eine Spur zu erkennen, um dieser zu folgen. Gleichzeitig heißt das, dass die Android Version für die Rottleuthner Arbeit auf mindestens Version 5.0 aktualisiert werden muss.
Um mit der Kamera-Programmierschnittstelle ein Bild aufzunehmen, muss eine Anzahl von Schritten durchlaufen werden.

Als erstes müssen Informationen über alle vorhandenen und aktiven Kamerasensoren des Geräts eingeholt werden (siehe [8]). Hierfür ist die Klasse *CameraManager* zuständig. Die Auskunft über die einzelnen Kameras werden in einem *CameraCharacteristics* geliefert. In diesem Objekt sind Informationen wie zum Beispiel Kameraausrichtung (vorne/hinten) in Relation zu Bildschirm, das Vorhandensein eines Kamerablitzes oder die unterstützten Auflösungen für Aufnahmen enthalten (siehe [7]).
Für Schritt zwei muss ein Objekt geschaffen werden, das die Kameraaufnahmen entgegennimmt sobald sie zur Verfügung stehen. Hierfür eignen sich eine *SurfaceTexture*, ein *ImageReader* oder ein *MediaRecorder*. Die *SurfaceView* zeigt für Vorschaubilder an. Der *ImageReader* kann für einzelne Bilder benutzt werden, die von der Applikation weiterverarbeitet werden können. Der *MediaRecorder* dient zur Aufnahme von Videos. Alle diese Klassen haben ein gemeinsames Element im Hintergrund und zwar eine *Surface*. Diese Oberfläche muss als Konsument für Bilddaten weitergereicht werden.
Im dritten Schritt muss das gewünschte *CameraDevice* aus der Liste des *CameraManagers* ausgewählt werden. Um das Kameraelement nutzen zu können muss die Methode *CameraManager.open(cameraId)* aufgerufen werden. Es handelt sich hierbei um einen asynchronen Aufruf und das Kameraelement wird über die Callback-Methode *onOpened()* ausgeliefert.
Im nächsten Schritt muss ein *CaptureRequest* für das *CameraDevice* generiert werden. Ein *CaptureRequest* ist eine unveränderliche Ansammlung von Einstellungen und Ausgaben, die für eine Bildaufnahme gebraucht wird (vgl. [9]). Allerdings sollte davon abgesehen werden, einen *CaptureRequest* direkt zu erstellen, da die Vielzahl von Einstellungsmöglichkeiten zu einer hohen Fehleranfälligkeit führt. Anstatt dessen sollte ein *CaptureRequest.Builder* verwendet werden, dieser Builder kann vom *CameraDevice* mit voreingestellten Vorlagen wie zum Beispiel *CameraDevice.TEMPLATE_PREVIEW* eingeholt werden. Zum Builder müssen dann nur noch die Konsumenten der Bilddaten zugefügt werden.
Im folgenden Schritt muss eine *CaptureRequestSession* erstellt werden. Diese dient als Kontext für die *CaptureRequest*. Um die *CaptureRequestSession* zu initialisieren, muss eine oder mehrere *Surface* bereitgestellt werden. Das erstellen einer Session kann mehrere hundert Millisekunden in Anspruch nehmen, da die dem Kameraelement internen Pipelines konfiguriert und Speicher allokiert werden müssen. Diese Operation wird somit asynchron ausgeführt und über den *onConfigured*-Callback wird die nutzbare Session ausgeliefert. Im Falle des Fehlschlagens der Konfiguration wird die Callback-Methode *onConfiguredFailed* aufgerufen, die angeforderte *CaptureRequestSession* wird nicht aktiviert und kann nicht benutzt werden (siehe [10]).
In Schritt sechs wird über die *CaptureRequestSession* ein *CaptureRequest* and das *CameraDevice* gestellt. Es gibt verschiedene Varianten eine Anfrage vorzulegen, um einzelne Bilder zu

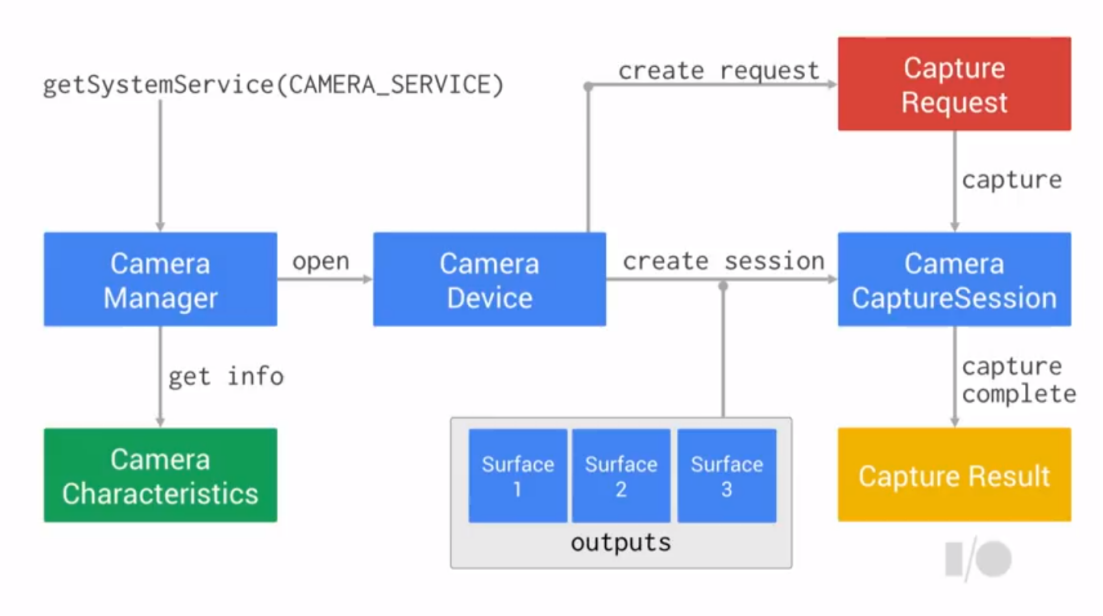


Abbildung 4.5: Zusammenspiel der Camera2 API Klassen

erhalten wird eine *capture()*-Anfrage gestellt. Wenn eine Vorschau mit den Bilddaten versorgt werden soll, muss eine *setRepeatingRequest()*-Anfrage gestellt werden. Das Aufsetzen einer neuen *setRepeatingRequest()* beendet alle vorhergehenden wiederholenden Anfragen. Soll eine sich wiederholende Aufnahmeanfrage beendet werden, muss die Methode *stopRepeating()* aufgerufen werden. Einzelne Aufnahmeanfragen haben Vorrang gegenüber sich wiederholenden Anfragen (vgl. [10]).
Im letzten Schritt ist es an der Zeit das *CaptureResult* zu verarbeiten. Das Resultat wird über den, beim Aufruf der *capture()*-Anfrage mitgegebenen, *CaptureRequestSession.CaptureCallback* zurückgegeben. Allerdings handelt es sich beim *CaptureResult* nicht um die Bilddaten, sondern nur um die bei der *CaptureRequest* mitgegebenen Konfigurationseinstellung und zusätzlich einige Metadaten über den Zustand der Kamera während der Aufnahme (siehe [11]). Die Bilddaten werden direkt an die oben genannten Konsumenten (*SurfaceView, MediaRecorder, ImageReader*) ausgeliefert.

4.4 Sicherer Zustand

Bei am Boden bleibenden mobilen Systemen kann absoluter Stillstand als sicherer Zustand dienen. Das heißt, dass die Motoren in eine voreingestellte Ruheposition oder neutrale Stellung gebracht werden müssen. Für den Motor, der als Antrieb dient, ist als Ruheposition diejenige zu wählen, die keine Rotation der Räder bewirkt. Für den Servo, der die Lenkung beeinflusst, ist die Ruheposition nicht wichtig, wenn Stillstand angenommen wird.
Es gibt verschiedene Gründe, warum in den sicheren Zustand übergegangen werden muss. Der am wahrscheinlichsten auftretende Grund liegt hierbei allerdings nicht auf Seite des Mikrocontrollers, sondern bei dem Android-Gerät. Dieser Grund besteht darin, dass die Fahrbahn nicht erkannt werden kann oder nicht gefunden wird. Als nächster Grund, um in den sicheren Zustand zu wechseln kommt der Verlust der Kommunikation zwischen Android-Gerät und Mikrocontroller in Frage. In diesem Fall sind zwei Möglichkeiten in Betracht zu ziehen: Als erstes kann eventuell der Nutzer informiert und dazu aufgefordert werden, Kontrolle mit Hilfe der Fernbedienung zu übernehmen. Als zweites ist der sofortige Übergang in den sicheren Zustand zu erwägen. Als dritter Grund wird der Verlust der Verbindung zur Fernbedienung angenommen, in diesem Fall muss sofort in den sicheren Zustand übergegangen werden, auch wenn die Kommunikation zwischen Android und Mikrocontroller noch aktiv ist, da es keine Möglichkeit mehr gibt im Notfall einzugreifen. Es ist allerdings sehr unwahrscheinlich, dass der dritte Grund auftritt, da das System aufgrund der sehr strikt definierten Nutzungsumgebung die Verbindung zur Fernbedienung nicht verlieren sollte. Vorstellbare Situationen für den Kommunikationsverlust zur Fernbedienung sind das Verlassen der Reichweite oder das Blockieren des Funksignals durch massive Gegenstände im direkten Weg. Diese beiden Situationen sollten im Bereich des Testfelds, ein einzelner Raum, nicht auftreten. Nicht ausschließbar ist es, dass es zu einem Ausfall der Hardware kommt, jedoch ist es je nach Schwere des Ausfalls nicht immer möglich korrekt oder überhaupt zu reagieren.

4.5 Hardware Aufbau

Ziel der Arbeit ist es, durch eine Linienerkennung einen Model-Truck bestimmten Linien folgen zu lassen. Als Modul für die Bilderkennung soll ein Smartphone genutzt werden. Das heißt, es muss eine Möglichkeit gefunden werden, das Gerät auf dem Lastwagen anzubringen, sodass die Kamera zumindest Teile des Untergrundes und der Fahrbahn aufnehmen kann. Zusätzlich muss für den Mikrocontroller, der die Signalgebung für die Motoren des Models erledigt, Platz gefunden werden. Nun muss noch dafür gesorgt werden, dass der Mikrocontroller, die Moto-

ren und der Funkempfänger mit Strom versorgt sind, also muss Raum für ein oder mehrere Batterien vorhanden sein. An den Model-Truck kann ein Anhänger gekoppelt werden, deshalb sollte es kein Problem sein, genügend Stauraum für alle verwendete Hardware zu finden.

4.6 Ähnliche Projekte

IOIO ist eine Baureihe von Mikrocontrollern, die es Android-Applikationen erlauben mit externen elektronischen Geräten zu interagieren. Die Verbindung mit dem Android-Gerät funktioniert über USB oder Bluetooth. Das Board erlaubt digitale Aus- und Eingänge und die mitgelieferten quelloffenen Bibliotheken ermöglichen es, Funktionen wie PWM, I2C, SPI oder USART zu verwenden. Diese Vorteile sprechen zwar dafür IOIO in diesem Projekt zu verwenden. Allerdings wurde von der Verwendung abgesehen, da es schon eine hausinterne Lösung der HAW für die Kommunikation mit Mikrocontrollern gibt. Außerdem erscheint das Produkt nicht mehr im aktiven Bestand des Herausgebers SparkFun zu sein, da alle Geräte im Shop als retired gekennzeichnet sind.

5 Konzept

In diesem Kapitel wird ein konzeptueller Lösungsansatz für die gestellten Probleme vorgestellt. Das System kann in zwei Teilsysteme separiert werden. Die Seite des Mikrocontrollers ist der erste Teil, der betrachtet werden soll. Darauffolgend wird die Seite des Android-Smartphones begutachtet. Die lose Kopplung der Systeme über den Rottleuthner Kommunikationsservice ermöglicht diese separate Betrachtungsweise. Allerdings muss ein Protokoll entwickelt werden, mit dem sich die beiden Teilsysteme verständigen können. Dies soll als letzter Schritt dieses Kapitels vorgestellt werden.

5.1 Mikrocontroller

Der Ablauf in diesem Abschnitt soll davon ausgehen, dass wir zuerst nur einen Motor mit dem Kommunikationsservice verbinden wollen. Im Folgenden wird erarbeitet, welche Komponenten zur Umsetzung dieses Verfahrens benötigt werden.

5.1.1 Motor

Um einen Motor ansteuern zu können muss, zuerst eine Timerbaustein als PWM-Signalgeber ausgewählt werden. Der Mikrocontroller STM32F4-Discovery bietet je nach Variante bis zu vierzehn Timerbausteine an. Nachdem der Timerbaustein konfiguriert ist, ist es möglich einen Servomotor mit den notwendigen Signalen zu bedienen. An dieser Stelle muss dafür gesorgt werden, dass der Motor Nachrichten von dem Kommunikationsservice erhalten kann. Der Kommunikationsservice arbeitet mit einem Subscriber-Model. Also muss die Motorkomponente das Interface eines Subscribers implementieren, um Nachrichten vom Service zu erhalten. An dieser Stelle ist anzumerken, dass die Motoren auch eine vitale Rolle in der Sicherheit des Systems spielen und somit auch von der Komponente, die die Fernsteuerung repräsentiert, Nachrichten erhalten müssen.

5.1.2 Fernbedienung

Weiterhin wird eine Komponente für den Umgang mit den vom Remote Receiver ausgelösten USART-Interrupts benötigt. An dieser Stelle kommt es zu dem Problem, dass die Mikrocontroller Umgebung und vor allem die Standard Bibliotheken in C geschrieben sind und somit nicht unbedingt mit C++ Objekten kommunizieren können. Es muss also ein C-Wrapper für das C++-Objekt geschaffen werden, der es erlaubt die Interruptinformationen weiterzuleiten. Die Remote Komponente muss außerdem dazu in der Lage sein, mehrere USART-Interrupts zu einem 16-Byte Datenpaket (siehe Kapitel 4.2) zusammenzufassen. Nachdem ein volles Datenpaket empfangen wurde, muss kontrolliert werden, ob die Fernbedienung die Kontrolle übernimmt. Es ist vorstellbar die Kanäle auf Veränderungen zu überprüfen, die die Steuerinformationen enthalten. Eine andere Vorgehensweise die denkbar ist, ist es einen der vielen anderen Kanäle der Fernbedienung zu nutzen um die Kontrollübernahme zu signalisieren. Am geeignetsten erscheint der Gear Kanal an der Spektrum DX8 Fernbedienung, da dieser nur zwei Werte (0 oder 1) annehmen kann. Die Variante, die die Steuerinformationen überprüft bietet den Vorteil, dass kein dritter Kanal notwendig ist. Allerdings muss der vorhergegangene Wert eines jeden Datenpakets gespeichert werden. Die in diesem Projekt verwendete Fernbedienung stellt genügend Kanäle zur Auswahl um einen dritten Kanal problemlos zu nutzen.

5.1.3 Verteiler

Zu diesem Zeitpunkt bekommt die Motorkomponente von zwei Quellen Nachrichten zugestellt und muss entscheiden, welche verarbeitet werden. Dies entspricht nicht der Aufgabe der Motorkomponente, die nur eine Nachricht empfangen soll und den darin enthaltenen Wert am Motor einstellen soll. Es ist also eine weitere Komponente zu kreieren, die das Verteilen der Nachrichten übernimmt. Der Verteiler muss entscheiden können, wann die Nachrichten aus dem Service benutzt werden und wann die Nachrichten der Fernbedienungskomponente Vorrang haben. Im Abschnitt über die Komponente der Fernbedienung wurde die Entscheidung getroffen, dass ein extra Kanal zur Kontrollübernahme festgelegt ist. Die Komponente muss also, wenn dieser Kanal sich geändert hat eine Nachricht an den Verteiler übergeben, dass die Kontrolle übernommen worden ist. Sobald der Verteiler diese Nachricht erhalten hat, müssen alle Nachrichten aus dem Service ignoriert werden. Der Einfachheit halber werden Nachrichten zwar weiter empfangen jedoch nicht bearbeitet. Eine andere Möglichkeit wäre, es den Nachrichtenservice über einen Kommunikationsstopp zu informieren und keine weiteren Nachrichten mehr auszutauschen. Der Kommunikationsservice bietet keine solche Funktion und müsste zusätzlich implementiert werden. Eine weitere Möglichkeit bestünde darin, die

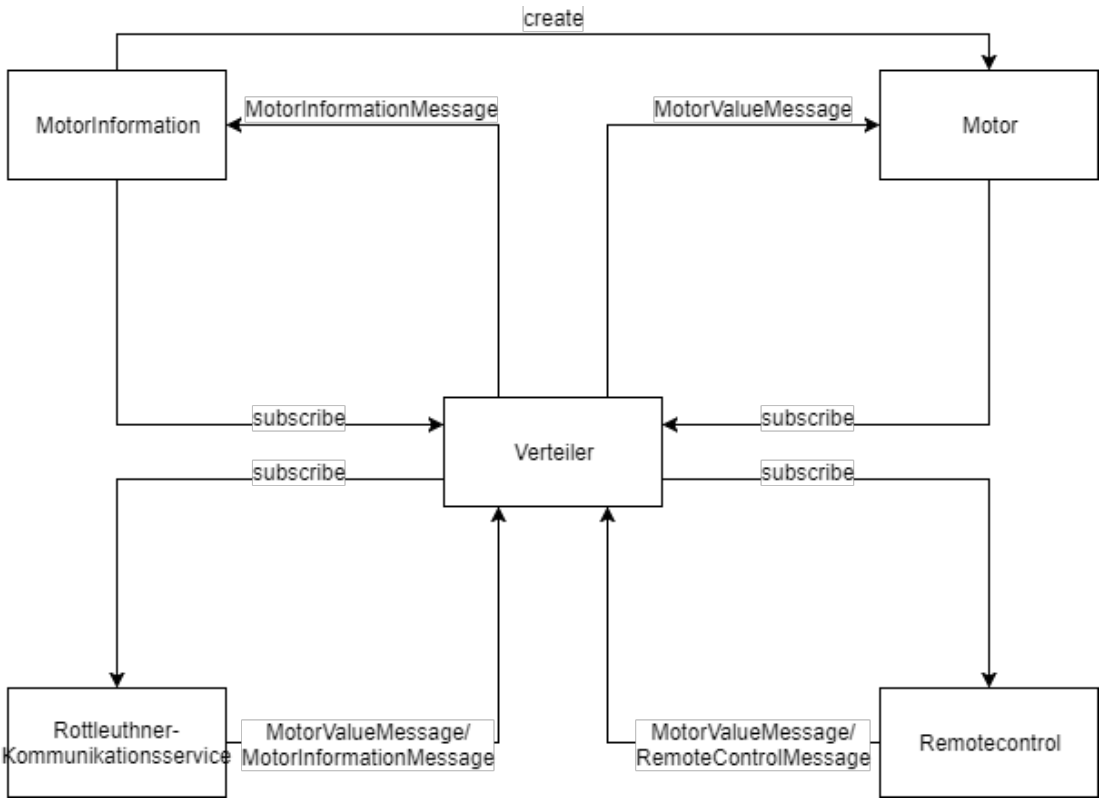


Abbildung 5.1: Nachrichtenaustausch unter den Komponenten

Nachrichten aus dem Kommunikationsservice nicht mehr zu bearbeiten, allerdings ist an dieser Stelle unklar wie die beinhaltete USB-Bibliothek den Speicher verwaltet und es könnte nach Wiederaufnahme der Verarbeitung zu Komplikationen mit veralteten Informationen kommen. Der Verteiler soll nicht jeden einzelnen Motor ansprechen, sondern Nachrichten aus dem Kommunikationsservice an alle angeschlossenen Motoren weiterleiten. Diese Funktionalität würde vom Verteiler erwarten, die Payload einer jeden Nachricht zu kontrollieren, für dieses Projekt wird das als unzulängliche Funktion eines Verteilers angesehen und jede Nachricht, die an Motoren gerichtet wird, ist an alle Motoren weiterzuleiten. Die Motoren überprüfen dann, ob die in der Payload enthaltene Identifikationsnummer die ihre ist und reagieren entsprechend, indem die Nachricht entweder verworfen oder der Motorwert eingestellt wird. Der Verteiler agiert also als Zustandsautomat. Der Automat entscheidet darüber, ob Nachrichten von der Fernbedienung oder vom Kommunikationsservice verarbeitet werden.

5.1.4 Konfiguration

An dieser Stelle ist das Projekt dazu in der Lage Nachrichten von Fernbedienung und Kommunikationsservice zu unterscheiden und entsprechend zu priorisieren. Eine weiter Funktion, die gefordert ist, ist die Konfiguration von Motoren von Seiten des Android-Gerätes aus. Hierfür ist eine weitere Komponente vonnöten. Diese Komponente soll eine Nachricht erhalten und die darin enthaltenen Informationen dazu nutzen um einen Motor zu initialisieren. Es muss also eine Motorkomponente erstellt werden, die sich dann wiederum beim Verteiler anmeldet, um Nachrichten von der Android Seite zu erhalten.

5.2 Android

In diesem Abschnitt soll die Software auf Seite des Android-Mobilgerätes in konzeptueller Form dargelegt werden. Diese Sektion lässt sich grob in zwei Teile zerlegen. Der erste Teil ist eine Komponente, die eine Schnittstelle und Abstraktion zum Rottleuthner Kommunikationsservice herstellt. Der zweite Teil soll sich mit der Spurerkennung unter dem Android-Betriebssystem beschäftigen.

5.2.1 Abstraktion

Die Abstraktion vom Kommunikationsservice kann in einem sehr schmalen Wrapper geschafft werden. Es müssen fürs Erste nur zwei verschiedene Nachrichten verschickt werden können.

Die erste Nachricht soll dazu in der Lage sein, Konfigurationsinformationen zu einem Motor zu enthalten. Die zweite Nachricht soll nur einen Motorenwert enthalten. Allerdings ist es notwendig, dass jeder Motor eine eindeutige Identifikation zugeordnet bekommt, also muss die Abstraktionsschicht auch dafür sorgen, dass eindeutige Identifikationsnummern vergeben werden. Jeder Nachricht, die nur einen Motorenwert enthält muss die dazugehörige ID beinhalten. Damit die Abstraktionsschicht mit dem Kommunikationsservice Nachrichten austauschen kann, muss sie ein Objekt der Klasse *ServiceBridge* aus dem Bibliotheksprojekt *aOAPClientLib* enthalten. Bibliotheksprojekte dienen unter der IDE Android Developer Tools dazu „zusätzliche Funktionen um Berechtigungen und Kompatibilitätsanforderungen in der Zielanwendung einzufordern“([25], Kapitel 6.2.5) und kann unter Android Studio mit den gleichen Vorteilen verwendet werden.

5.2.2 Kamerasetup

Um die Camera2 API vorzustellen, wurde eine Beispielapplikation in Form eines Android-Fragments zu Verfügung gestellt. Innerhalb des Fragments werden grundlegende Operationen durchgeführt um ein Kamerabild auf dem Speicher des Geräts abzulegen und eine Vorschau anzuzeigen. Zusätzlich werden viele kleine Funktionen gezeigt um die Kamerabilder in der gewünschten Ausrichtung oder dem gewünschten Format zu erhalten. Dieses Fragment kann, um die in diesem Projekt angestrebten Funktionen erweitert werden und wird deshalb genutzt, um weiteren Implementationsaufwand zu vermeiden.

5.2.3 Spurerkennung

In Abschnitt 4.3 wurde schon sehr ausführlich, beschrieben welche Schritte unternommen werden müssen, um Bilder von der Kamerahardware zu erhalten. Es werden die Bilddaten gelesen werden müssen, um das zu erreichen muss ein *ImageReader* implementiert werden. Der *ImageReader* muss dann aus den Bilddateien die Spurinformationen herausfiltern. Zuerst muss geklärt werden, in welchen Formaten die Bilder der Kamera ausgeliefert werden. Das Bildformat, das auf jedem Android-Gerät mit einer Kamera, unterstützt wird ist JPEG (vgl. [12]). JPEG ist ein Format, das Bilder komprimiert. Das Format erweist sich für dieses Projekt als ungeeignet, da mit Hilfe von Farbinformationen eine Spur erkannt werden soll. Um diese Farbinformationen zurückzuerhalten, muss die Komprimierung rückgängig gemacht werden. Dieses Vorgehen würde zu viel Rechenzeit in Anspruch nehmen. Eine weitere Möglichkeit, Bildinformationen unter Android zu erhalten, ist das *RAW10*-Format. Hierbei handelt es sich um vom Bildsensor aufgenommenen Daten, die unbearbeitet weitergeleitet werden. Allerdings

hängt das Bereitstehen dieses Formats stark von der genutzten Hardware ab und ist für dieses Projekt ungeeignet. Das für dieses Projekt genutzte Format ist *YUV_420_888*. Es liefert Bildinformationen in drei Ebenen aus.
Die erste Ebene (Y) enthält Informationen über die Grundhelligkeit der Pixel. Die nächste Ebene beinhaltet Farbinformationen der Blau-Gelb-Chrominanz (U). Die dritte Ebene gibt Auskunft über die Rot-Grün-Chrominanz (V). Chrominanz ist in diesem Falle als Farbigkeit des Pixels anzusehen. Aus diesen Informationen lassen sich Rot-Grün-Blau-Werte errechnen auf denen dann die Spurerkennung durchgeführt werden kann. Die Werte errechnen sich mit der folgenden Formel für die Konvertierung von YCbCr-Werten zu RGB-Werten:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.164 & 0.000 & 1.596 \\ 1.164 & -0.392 & -0.813 \\ 1.164 & 2.017 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ U \\ V \end{pmatrix} \tag{5.1}$$

Der Datensatz mit RGB-Werten kann nun nach der Farbe der Fahrbahn durchsucht werden. Es ist nicht nötig, den gesamten Datensatz zu durchsuchen, sondern es genügt einige Reihen des Bildes auf die Farbe zu überprüfen. Vorstellbar wäre es, aus diesen Informationen Punkte auf der Fahrbahn zu bestimmen. Aus zwei Punkten kann dann ein rechtwinkliges Dreieck konstruiert werden, mit Hilfe des Dreiecks kann ein Lenkwinkel bestimmt werden. Es soll einen Fixpunkt im Blickfeld der Kamera geben, an dem die Farbe der Fahrbahn immer zu erkennen ist, damit bei eventuellen Veränderungen der Farbe durch Schatten oder Licht keine Komplikationen auftreten.

5.3 Kommunikation

In diesem Abschnitt soll ein Konzept entwickelt werden, wie die vorhandene Kommunikationskomponente genutzt werden soll, um die notwendigen Funktionen für dieses Projekt zu bieten. Der Rottleuthner Kommunikationsservice nutzt zum Serialisieren von Nachrichten Protocol Buffers von Google. Auf diesem Wege sollen auch die hier gebrauchten Nachrichten definiert werden, um diese mit Hilfe der Protocol Buffer zu serialisieren. Die neu definierten Nachrichten werden dann in der Payload der auf Abbildung 5.2 zu sehenden Nachricht zwischen den Kommunikationspartner übertragen. Es werden insgesamt zwei Nachrichtentypen genutzt. Die erste Nachricht muss Informationen zu den genutzten Motoren beinhalten (siehe Listing 5.1). Das Feld *isCritical* sagt über den Motor aus, ob dieser im Falle der Kontrollübernahme der Fernbedienung weiterhin Nachrichten bekommen soll. Die Felder *motorRole* und *motorType*

Abbildung 5.2: Nachrichtenaufbau

dienen dazu, den Motor näher zu beschreiben. Die andere Nachricht muss nur eine MotorID und den am Motor einzustellenden Wert enthalten (siehe Listing 5.2). Die Identifikation der Motoren soll in diesem Fall eine fortlaufende Zahl sein, die bei 123 startet. Die ID wird von der Android Seite erstellt und auf der Mikrocontroller-Seite nur übernommen.

```
message MotorInformationMessage{
        required int32 motorID    = 1;
        required int32 motorRole  = 2;
        required int32 motorType  = 3;
        required bool  isCritical = 4;
}
```

Listing 5.1: Nachrichtendefinition: MotorInformationMessage

```
message MotorValueMessage{
        required int32 motorID = 1;
        required float motorValue = 2;
}
```

Listing 5.2: Nachrichtendefinition: MotorValueMessage

Weiterhin kann es von Vorteil sein, den physischen Link zwischen Android und MCU und die Funktionsfähigkeit der Geräte mit einem Überwacher (Watchdog) zu kontrollieren. Hierfür muss noch ein weiterer Nachrichtentyp implementiert werden. Der Nachrichtentyp könnte nur ein Feld für einen Zähler beinhalten, dieser Zähler wird dann von beiden Seiten bei jedem Empfangen um eins erhöht. Der Ausfall beider Geräte ist allerdings eher unwahrscheinlich und USB als Kommunikationsmedium gilt als zuverlässig. Somit ist ein Watchdog nicht unbedingt notwendig.

5.4 Fahrbahn

Bei der Fahrbahn handelt es sich um eine Spur die sich farblich stark von ihrem Umfeld abhebt. Denkbar wäre es, eine Strecke mit Isolierband auf den Boden zu kleben, und diese als Fahrbahn zu nutzen. Wichtig ist, dass die Lichtverhältnisse möglichst gleichmäßig sind, sodass sich die Farben im Streckenverlauf nicht zu stark unterscheiden. Ein optimales Verhältnis von Fahrbahnfarbe zu Hintergrund ist in Abbildung 5.3 zu sehen.

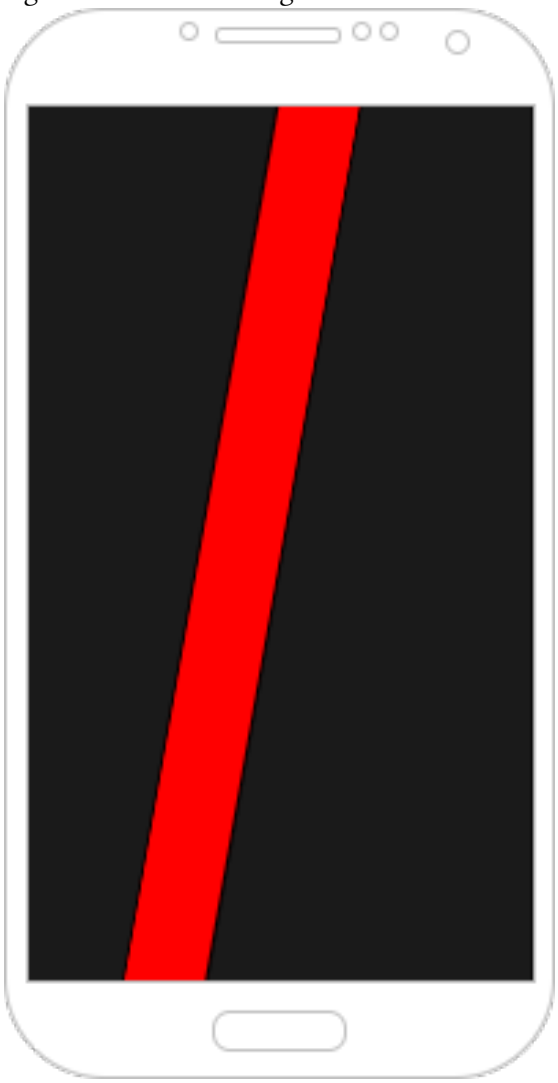


Abbildung 5.3: Fahrbahn aus Sicht der Android-Applikation

6 Realisierung

Im folgenden Kapitel soll genauer auf die Implementation des ausgearbeiteten Konzepts eingegangen werden. Es werden wichtige Funktionen anhand von Codeauszügen erklärt. Das Kapitel wird aufgeteilt in die verschiedenen implementierten Klassen aufgeteilt.

6.1 RemoteControlService

Um USART benutzten zu können müssen einige Schritte unternommen werden. Als erster Schritt muss entschieden werden welcher der acht verschiedenen USART-Kanäle benutzt werden soll, damit der korrekte Peripherie-Bus initialisiert werden können. In diesem Fall wurde

U(S)ARTx	TX	RX	APB
USART1	PA9, PB6	PA10, PB7	2
USART2	PA2, PD5	PA3, PD6	1
USART3	PB10, PC10, PD8	PB11, PC11,PD9	1
UART4	PA0,PC10	PA1, PC11	1
UART5	PC12	PD2	1
USART6	PC6, PG14	PC7, PG9	2
UART7	PE8, PF7	PE7, PF6	1
UART8	PE1	PE0	1

Tabelle 6.1: Pinbelegung für U(S)ART STM32F4

USART2 für die Kommunikation mit der Fernbedienung gewählt. Als erstes muss der GPIO-Bus für den gewählten USART aktiviert werden. Hierbei kommt es darauf an, welche Pins für eingehende Daten (RX) und ausgehende Daten (TX) verwendet werden sollen. Für dieses Projekt wurden die Pins PA3 und PA2 ausgewählt, somit muss der GPIOA-Bus aktiviert sein um USART nutzen zu können . Da Pins auf dem STM32F4 mit Funktionen überladen sind, muss festgelegt werden welche Funktion die Pins in diesem Fall bedienen sollen (siehe Listing 6.1). Danach muss der USART-Peripheriebaustein aktiviert und initialisiert werden. An dieser Stelle werden Optionen wie Baurate, Parität und Anzahl der Stopbits eingestellt. Mit USART eingehende Daten werden über Interrupts ausgeliefert, deshalb muss der für den USART-Baustein relevante Interrupt

aktiviert und im Nested Vector Interrupt Controller (NVIC) vermerkt werden (siehe Listing 6.2).

```
GPIO_InitTypeDef     GPIO_InitStruct;

RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

GPIO_PinAFConfig(GPIOA, GPIO_PinSource2, GPIO_AF_USART2);
GPIO_PinAFConfig(GPIOA, GPIO_PinSource3, GPIO_AF_USART2);
GPIO_InitStruct.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_Init(GPIOA, &GPIO_InitStruct);
```

Listing 6.1: Initialisierung des GPIOA-Bausteins, Quelle [3]

```
USART_InitTypeDef USART_InitStruct;
NVIC_InitTypeDef NVIC_InitStruct;

RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);

USART_InitStruct.USART_BaudRate = 125000;
USART_InitStruct.USART_HardwareFlowControl =
                        USART_HardwareFlowControl_None;
USART_InitStruct.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
USART_InitStruct.USART_Parity = USART_Parity_No;
USART_InitStruct.USART_StopBits = USART_StopBits_1;
USART_InitStruct.USART_WordLength = USART_WordLength_8b;
USART_Init(USART2, &USART_InitStruct);
USART_Cmd(USART2, ENABLE);
USART_ITConfig(USART2, USART_IT_RXNE, ENABLE);

NVIC_InitStruct.NVIC_IRQChannel = USART2_IRQn;
NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0;
NVIC_Init(&NVIC_InitStruct);
```

Listing 6.2: Initialisierung USART und NVIC, Quelle [3]

Zur Steigerung der Benutzerfreundlichkeit wird für dieses Projekt die Bibliothek aus Quelle

[3] verwendet. Hier gibt es die Möglichkeit einen eigenen Interrupt zu verwenden, dieser wird dann an die Klasse RemoteControlService weitergeleitet. In Sektion 4.2 wird beschrieben, wie groß ein vollständiges Datenpaket des genutzten Empfängers ist. Es müssen also 16 Interrupts erfolgen bis ein Datenpaket in seiner Gesamtheit übertragen ist. Danach kann mit den empfangenen Daten gearbeitet werden. Es wurde festgelegt, dass ein zusätzlicher Kanal die Kontrollübernahme bestimmt. Hierfür wird nach dem vollständigen Empfangen eines Pakets das Vorhandensein des Kanals überprüft. Falls der Gear-Kanal vorhanden ist und seinen Wert geändert hat, wird eine Nachricht an den Dispatcher gesendet, dass die Kontrolle übernommen wird. Ab diesem Zeitpunkt muss jedes Paket nach den Kanälen Aileron und Elevator durchsucht werden. Wenn die Kanäle vorhanden sind, müssen die enthaltenen Werte entnommen und umgerechnet werden. Die Umrechnung muss passieren, da der Motor Gradzahlen erwartet und die Fernbedienung Werte im Bereich von 0 - 2048 liefert. Danach werden die Werte an den Dispatcher weitergereicht.

6.2 Dispatcher

Die Dispatcher-Klasse hat hauptsächlich die Aufgabe zwischen Nachrichten der Fernbedienung und des Kommunikatonsservice zu unterscheiden. Hierfür wurden vier Nachrichtentypen definiert.
Der Typ **MSG_TYPE_MOTOR** ist dabei der im Normalbetrieb einzig vorkommenden Nachrichtentyp. Bei diesem Typ handelt es sich um Nachrichten die ihren Ursprung auf der Android-Seite gefunden haben und der schon in Listing 5.2 beschrieben wird. Diese Nachricht wird unverändert an alle Abonnenten weitergeleitet.
Der Typ **REMOTE_ACCESS** zeigt an, dass die Fernbedienung die Kontrolle übernimmt. Wenn diese Nachricht empfangen ist, wird eine Boolean-Variable gesetzt, sodass Nachrichten vom Typ MSG_TYPE_MOTOR nicht mehr bearbeitet werden.
Um die Verzögerung der Bearbeitung durch das Deserialisieren von Nachrichten zu vermeiden, werden die nächsten beiden Nachrichtentypen mit direkten Referenzen auf Werte übergeben. Das Serialisieren ist für Nachrichten, die innerhalb des Klassengefüges auf der Seite des Mikrocontrollers ausgetauscht werden, nicht sinnvoll. Bei den Nachrichtentypen handelt es sich um **MSG_TYPE_REMOTE_DIR** und **MSG_TYPE_REMOTE_SPEED**. Hierbei ist zu beachten, dass sich nur jeweils ein Motor für diese Nachricht als Empfänger beim Dispatcher anmelden kann. Für den Nachrichtentype MSG_TYPE_REMOTE_DIR sollte sich derjenige Motor anmelden, der die Lenkung kontrolliert. Für den Nachrichtentyp MSG_TYPE_REMOTE_SPEED

sollte sich derjenige Motor anmelden, der die Geschwindigkeit kontrolliert.
Der Dispatcher implementiert das Interface IPCClientInterface, damit vom IPCService Nachrichten empfangen werden können. Zusätzlich muss der Dispatcher die oben genannte MSG_TYPE _MOTOR beim IPCService abonnieren. Hierfür muss die Methode aus Listing 6.3 mit den folgenden Parametern aufgerufen werden. Man benötigt erstens einen Zeiger auf den Dispatcher und zweitens eine Nachricht vom Typ MSG_TYPE _MOTOR. Der Rückgabewert der Methode hat in der genutzten Implementierung keinerlei Bedeutung, da er immer 0 ist.

```
1 int subscribeForMessages(IPCClientInterface* client, uint8_t msgType);
```

Listing 6.3: Abbonieren einer Nachricht

Falls der Dispatcher um Nachrichtentypen erweitert werden soll, muss das an zwei Stellen ermöglicht werden. Zunächst muss der Dispatcher die neuen Nachrichtentypen beim IPCService abonnieren. Sodann muss die Methode aus Listing 6.4 um die Bearbeitung der Nachrichtentypen erweitert werden.

```
1 int onReceiveData(uint16_t src, uint8_t type,
2                                 uint8_t* data, uint16_t dataLen);
```

Listing 6.4: Methoden-Aufruf zum Bearbeiten von Nachrichten

Derzeit ist der Dispatcher nicht dazu in der Lage, Nachrichten in Richtung der Android-Seite zu verschicken. Allerdings ist das auch nicht Aufgabe des Dispatchers und sollte durch eine andere Komponente erledigt werden. Diese Komponente müsste ebenfalls das IPCClientInterface implementieren. Die Methode process() müsste dann das Verschicken von Nachrichten erledigen, da diese periodisch vom IPCService aufgerufen wird, um eben diese Funktion zu bieten.

6.3 MotorValueClient

Jeder MotorValueClient repräsentiert einen Servomotor. Servomotoren erwarten ein PWM-Signal mit einem Duty Cycle zwischen 1000 μs und 2000μs um einen Winkel zwischen 0 und 180 Grad einzustellen. Um dieses Signal generieren zu können, muss ein Timerbaustein auf dem STM32F4 mit der notwendigen Funktion initialisiert werden. In Tabelle 6.2 sind die vorhandenen Bausteine und die dazugehörigen Pins aufgelistet. Hierbei handelt es sich allerdings um die Timer eines STM32F429-Boards. Die Pins auf dem genutzten STM32F407-Boards belegen die gleichen Funktionen, sodass Unterschiede zwischen den Boards nicht relevant sind.

Timer	Channel 1			Channel 2			Channel 3		Channel 4	
	PP1	PP2	PP3	PP1	PP2	PP3	PP1	PP2	PP1	PP2
TIM1	PA8	PE9		PA9	PE10		PA10	PE13	PA11	PE14
TIM2	PA0	PA5	PA15	PA1	PB3		PA2	PB10	PA3	PB11
TIM3	PA6	PB4	PC6	PA7	PB5	PC7	PB0	PC8	PB1	PC9
TIM4	PB6	PD12		PB7	PD13		PB8	PD14	PB9	PD15
TIM5	PA0	PH10		PA1	PH11		PA2	PH12	PA3	PI0
TIM8	PC6	PI5		PC7	PI6		PC8	PI7	PC9	PI2
TIM9	PA2	PE5		PA3	PE6					
TIM10	PB8	PF6								
TIM11	PB9	PF7								
TIM12	PB14	PH6		PB15	PH9					
TIM13	PA6	PF8								
TIM14	PA7	PF9								

Tabelle 6.2: Timer mit PWM Einstellungsmöglichkeiten

Der MotorValueClient empfängt drei verschiedene Nachrichten. Im Normalbetrieb ist die einzige Nachricht, die empfangen wird MSG_TYPE_MOTOR. Für diese Nachricht muss der Inhalt deserialisiert werden. Dies wird mit dem Methodenaufruf in Listing 6.5 durchgeführt. Wenn die Deserialisierung erfolgreich ist, muss die MotorID überprüft werden, wenn die MotorID der eigenen entspricht, kann der gelieferte Motorwert eingestellt werden. Für die Ansteuerung der Servomotoren wird die Bibliothek [5] genutzt. Die Bibliothek bietet die Möglichkeit PWM-Signale im definierten Bereich zu generieren. Die Pulslängen können über zwei Varianten eingestellt werden. Die erste Variante besteht darin, eine Gradzahl zu übergeben die zwsichen 0 und 180 liegen muss. Die zweite Variante ist, eine Pulslänge direkt in μs anzugeben. Es sind nur Pulslängen zwischen 1000 und 2000μs erlaubt.

```
1 bool pb_decode(pb_istream_t *stream,
2                const pb_field_t fields[], void *dest_struct);
```

Listing 6.5: Deserialisierung

6.4 MotorInformationClient

Die MotorInformationClient-Klasse empfängt Nachrichten des Typs 5.1 direkt vom IPCService und muss somit Nachrichten des Typs MSG_TYPE_MOTOR_INFO abonnieren. Die Informationen werden ausgelesen und daraus wird ein neues Objekt der Klasse MotorValueClient gebildet, dieses Objekt wird dann beim Dispatcher angemeldet. Um weitere Motorkonfigurationen zu ermöglichen, kann die Nachricht aus dem genannten Listing um Informationen zu dem zu

nutzenden Pin erweitert werden. Falls keine Nachrichten empfangen werden, initialisiert diese Klasse zwei Motoren auf den Pins PA5 und PB3.

6.5 MotorMessageHandler

Der MotorMessageHandler dient als Abstraktionsschicht zwischen dem Rottleuthner- Kommunikationsservice und der Applikation, die die Motoren des Model-Trucks ansteuert. Die Abstraktion beinhaltet wenig eigene Funktionalität. Sie schirmt nur den Benutzter von den komplizierten Vorgängen des Serialisierens und Versenden von Nachrichten ab. Zu diesem Zweck dienen die zwei Methoden Listing 6.6 und 6.7. Beide Nachrichtentypen arbeitet nach dem Prinzip Fire and Forget, das heißt, es handelt sich hierbei um asynchrone Nachrichten. Es ist davon auszugehen, dass dieses Vorgehen vor allem für die Motorwertnachricht keine Auswirkungen hat. Die Methode *sendMotorInformationMessage* gibt die Motoridentifikationsnummer für den generierten Motor zurück, diese ID muss für jede Nachricht, die diesen Motor erreichen will an die Methode *sendMotorValue* übergeben werden. Der einzustellende Wert *degrees* muss im Bereich zwischen 0 und 180 Grad liegen.

```
1 public void sendMotorValue(int motorID, float degrees);
```

Listing 6.6: Methode zum Versenden von Motorwerten

```
1 public int sendMotorInformationMessage();
```

Listing 6.7: Methode zum Versenden von Motorinformationen

Damit das Verschicken von Nachrichten funktionieren kann, muss die MotorMessageHandler-Klasse ein Objekt der Klasse ServiceBridge erstellen. Die ServiceBridge ist im AOAPClientLib-Projekt von Michel Rottleuthner hinterlegt. Der ServiceBrige muss beim initialisieren eine Referenz auf ein Objekt der Klasse AOAPClientInterface übergeben werden. Mit dem Aufruf *connectToService()* wird der Handshake mit dem Rottleuthner-Kommunikationsservice durchgeführt, falls der Verbindungsaufbau erfolgreich war, wird dies über die Callback-Methode des AOAPClientInterface *onClientRegistered()* mitgeteilt. Für die asynchrone Übertragung von Daten wird die Methode *transmitDataAsync(int src, int dst, int msgType, byte[] data)* des von der ServiceBridge eingeholten Service-Stubs genutzt. In diesem Anwendungsfall handelt es sich bei dem Byte-Array um eine MotorValueMessage oder MotorInformationMessage, die mit Hilfe des aus den Protocol Buffern autogenerierten Codes serialisiert wird.

6.6 Spur Erkennung

Im Abschnitt 5.2.3 wurde das verwendete Bildformat und eine Formel zur Berechnung von RGB-Werten vorgestellt. Bei dem hier angewandten Algorithmus handelt es sich um eine Integer-Version (Quelle: [6]) der Formel aus dem Abschnitt 5.2.3. Das Android-Betriebssystem liefert das YUV-Format in drei Ebenen aus. Die U- und V-Ebene sind halb so groß wie die Y-Ebene. Das bedeutet, dass vier Werten der Y-Ebene jeweils ein Wert der anderen Ebenen zugeordnet werden muss. Für ein besseres Verständnis soll Abbildung 6.1 sorgen. Die Funktion *getRGBIntFromPLanes()* liefert ein Array mit ARGB-Werten zurück, der Wert A ist hierbei die Transparenz der Farbe. Wert 0 bedeutet völlig durchsichtig, Wert 255 nicht transparent. In diesem Fall ist A immer 255 damit die Farbe deutlich zu erkennen ist. Um die Verständlichkeit der errechneten Werte zu erhöhen, wird aus ihnen eine Bitmap geformt. Nun ist es möglich, auf die Werte über Reihen und Spalten des Bildes zuzugreifen.
Da die Kamera statisch auf dem Model-Truck angebracht ist, kann ein Punkt bestimmt werden bei dem immer die Farbe der Fahrbahn zu finden ist. Bei diesem Punkt handelt es sich um die untere linke Ecke des Bildes, anders gesagt die Koordinaten (0 , Bitmap.height()). Um Rechenzeit zu minimieren werden nur einzelne Zeilen des Bildes nach der Farbe durchsucht. Die Farbe wird identifiziert, indem die drei Werte Rot, Grün und Blau mit dem Wert am Vergleichspunkt verglichen werden, wenn sie sich einem Wertebereich von ± 10 befinden wird die Farbe als dieselbe angenommen. Dieses Verfahren zieht keine ungleichmäßigen Belichtungsverhältnisse in Betracht. Die Umgebung muss also ausreichend gleichmäßig beleuchtet sein, damit das angewendete Verfahren erfolgreich ist.

```
1  private int[] getRGBIntFromPlanes(Image.Plane[] planes, int height,
2          int mwidth) {
3    ByteBuffer yPlane = planes[0].getBuffer();
4    ByteBuffer uPlane = planes[1].getBuffer();
5    ByteBuffer vPlane = planes[2].getBuffer();
6
7    int bufferIndex = 0;
8    final int total = yPlane.capacity();
9    final int uvCapacity = uPlane.capacity();
10   final int width = planes[0].getRowStride();
11
12   int [] mRgbBuffer = new int[height * mwidth];
13
14   int yPos = 0;
```

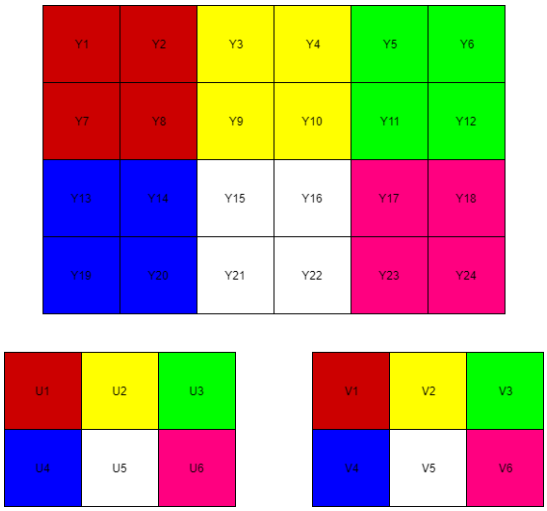


Abbildung 6.1: Visuelle Repräsentation der YUV-Ebenen

```
15   for (int i = 0; i < height; i++) {
16     int uvPos = (i >> 1) * width;
17
18     for (int j = 0; j < width; j++) {
19       if (uvPos >= uvCapacity-1) break;
20       if (yPos >= total) break;
21       final int y1 = yPlane.get(yPos++) & 0xff;
22       final int u = (uPlane.get(uvPos) & 0xff) - 128;
23       final int v = (vPlane.get(uvPos) & 0xff) - 128;
24       if ((j & 1) == 1) {
25         uvPos += 2;
26       }
27 // This is the integer variant to convert YCbCr to RGB, NTSC values.
28 // formulae found at
29 // https://software.intel.com/en-us/android/articles/
30 // trusted-tools-in-the-new-android-world-optimization-
31 // techniques-from-intel-sse-intrinsics-to
32       final int y1192 = 1192 * y1;
33       int r = (y1192 + 1634 * v);
34       int g = (y1192 - 833 * v - 400 * u);
35       int b = (y1192 + 2066 * u);
36
37       r = (r < 0) ? 0 : ((r > 262143) ? 262143 : r);
38       g = (g < 0) ? 0 : ((g > 262143) ? 262143 : g);
```

```
39          b = (b < 0) ? 0 : ((b > 262143) ? 262143 : b);
40  //blue b and red r swapped due to android being weird
41          mRgbBuffer[bufferIndex++] = (((b << 6) & 0xff0000) |
42          ((g >> 2) & 0xff00) | ((r >> 10) & 0xff)) | 0xff000000;
43          }
44    }
45    return mRgbBuffer;
46  }
```

Listing 6.8: Umrechnung von YUV- zu ARGB-Bildformat

Die genutzte Beispielapplikation muss in einigen Punkten verändert werden, um die Bildinformation in geeigneter Form zu erhalten. Diese Veränderungen sollen im Folgenden vorgestellt werden.
Als erstes muss das gewünschte Ausgabeformat in der Methode textitsetUpCameraOutputs() angepasst werden, um das Format YUV_420_888 auszugeben.

```
1 Size largest = Collections.max(
2   Arrays.asList(map.getOutputSizes(ImageFormat.YUV_420_888)),
3   new CompareSizesByArea());
4 mImageReader = ImageReader.newInstance(largest.getWidth()
5   /*smaller size preview*//16,
6   largest.getHeight()/*smaller size preview*/ /16,
7   ImageFormat.YUV_420_888, /*maxImages*/2);
```

Listing 6.9: Anpassung an die Kameraoutputs

Um die Rechenlast weiter zu reduzieren, wird die Größe des an den ImageViewer gelieferte Bild, um einen Faktor von 16 verringert (siehe Listing 6.9).
Als nächstes muss die Suface des ImageViewers zu der CapterRequest der Vorschau hinzugefügt werden. Hierfür muss zuerst die Surface vom ImageViewer geholt werden und danach diese Surface hinzugefügt werden (siehe Listing 6.10).

```
1 Surface mImageSurface = mImageReader.getSurface();
2 mPreviewRequestBuilder.addTarget(mImageSurface);
```

Listing 6.10: Hinzufügen eines Ziels für eine CaptureRequest

Hiermit sind alle Elemente der Realisierung beschrieben.

7 Evaluation

Um einschätzen zu können, ob die Realisierung vor allem für die Android-Seite und die Spurerkennung die notwendigen Daten liefert, um den Model-Truck sicher zu steuern, muss zuerst festgelegt werden wie viele Steuersignale übertragen werden müssen, um die sichere Bewegung zu gewährleisten. Bei der sicheren Bewegung geht es darum, die Fahrbahn nicht aus dem Blickfeld zu verlieren. Das heißt, die Änderungen im Lenkwinkel müssen schnell genug erfolgen, damit der Fahrspur weiterhin gefolgt wird. Leider hat die Zeit während der Bearbeitung nicht ausgereicht, um ausführliche Tests durchzuführen. Um jedoch in etwa einschätzen zu können, ob die errechneten Steuerdaten in regelmäßigen und kleinen Intervallen berechnet werden, wurde die Bitmap, die aus den ARGB-Werten generiert wurde, auf einer zusätzlichen Oberfläche angezeigt. Es war mit dem menschlichen Auge keine Verzögerung des Anzeigebildes im Vergleich zum Vorschaufenster zu beobachten.
Im Kapitel drei Anforderungen wurden einige Funktionen, definiert die das System an dieser Stelle erfüllen sollte. Im folgenden wird noch einmal auf diese eingegangen und eingeschätzt inwiefern ihnen Genüge getan wird.
Die erste Anforderung, die an das System gestellt wurde, ist, dass es dynamisch gegenüber verschiedenen Motoren reagieren soll. Diese Anforderung ist erfüllt, da mehrere Motoren angeschlossen werden können und an diesen verschiedene Funktionen über die Konfigurationsnachrichten eingestellt werden können.
Als nächstes wurde Sicherheit gegenüber der Umgebung und sich selbst gefordert. Diese Anforderung ist erfüllt durch das Implementieren der Fernbedienung. Allerdings ist hier immer noch die Aufmerksamkeit des Benutzers gefordert.
Als dritte Anforderung wurde das Benutzen der Rottleuthner-Kommunikation gefordert. Sie wurde genutzt und implementiert. Außerdem wurde die Service-Komponente auf eine aktuellere Android Version angehoben.
Als vierte Anforderung wurde die Kompatibilität mit verschiedenen Hardwareauslegungen gefordert. Auf Mikrocontroller-Seite ist zumindest Kompatibilität zu verschieden Boards des Herstellers STMicroelectronics geschaffen, durch die Benutzung der Standardbibliotheken. Auf Android-Seite ist die Kompatibilität ist durch das Betriebssystem weitestgehend abgedeckt.

Allerdings muss das Gerät Anschlüsse für USB und Kamera haben.
Als nächstes wurde die Benutzbarkeit für Programmieranfänger gefordert. Diese ist immerhin für die Kommunikation durch den MotorMessageHandler gegeben. Allerdings dürften Programmieranfänger überfordert sein, wenn sie das relativ komplizierte Programmierinterface auf einem Android-Gerät benutzen sollen, welches die Vielfalt von Sensoren abstrahiert.
Zuletzt wurde gefordert, dass die Kommunikationskomponente Daten es erlaubt, schnell genug Daten zu übertragen. Zwar wurden keine eigenen Tests durchgeführt, es kann aber durch die Daten aus Michel Rottleuthners Arbeit davon ausgegangen werden, dass die Kommunikationskomponente ihre Aufgabe ausreichend erfüllt.
Viele Details können noch verbessert werden. Vor allem im Rahmen der Konfiguration von Motoren kann noch einiges getan werden. Die Konfiguration sollte es erlauben, den PWM-Ausgang für den Motor selbst zu wählen. Der derzeitig genutzte Algorithmus zur Spurerkennung greift auf keine bekannten Mechanismen der Bildbearbeitung, wie zum Beispiel Kantenerkennung, zurück. Außerdem ist die Umrechnung der Bildinformationen zu kostspielig im Bezug auf die Rechenleistung. Mit dem API-Level 23 ist es möglich, direkt ARGB-Informationen von Bildern zu erhalten. Diese Information wurde leider erst nach der Entscheidung für API-Level 21 offengelegt und somit nicht mehr in Betracht gezogen.

8 Zusammenfassung

In dieser Bachelorarbeit wurden folgende Ziele erreicht: Als erstes wurde ein Programm für den Mikrocontroller geschrieben, dass die Ansteuerung von Motoren ermöglicht. Als nächstes wurde für den Mikrocontroller eine Komponente geschaffen, die es ermöglicht Informationen über USART von einer Fernbedienung zu erhalten. Daraufhin wurde der Rottleuthner-Kommunikationsservice auf der Seite des Mikrocontrollers und auf der Android-Seite implementiert. Die Servicekomponente konnte auf Seite des Android-Geräts nicht in der gelieferten Version benutzt werden und musste an eine aktuellere Version des Betriebssystems angepasst werden. An dieser Stelle wurde eine Abstraktionsschicht geschaffen, die die Kommunikationsschicht in ein Werkzeug zum Verschicken von Nachrichten, die Motorinformationen und Motorwerte enthalten, macht. Als letztes wurde eine Applikation programmiert, die Bildinformationen vom Kamerasensor nutzt, um einer Spur zu folgen, indem die Motoren eines Model-Trucks über das entwickelte Werkzeug angesteuert werden.
Die Definition eines Frameworks sagt aus, dass ein Framework eine fast komplette Applikation ist, die eine wiederverwendbare Struktur für verschiedene Applikationen zu Verfügung stellt. Entwickler können diese dann mit ihren eigenen Ideen erweitern. Das in dieser Arbeit geschaffene Produkt stellt eine sehr spezielle Lösung für ein sehr spezielles Problem dar und sollte aus diesem Grund eher als Werkzeug bezeichnet werden.

Literaturverzeichnis

[1] Industry Leaders Announce Open Platform for Mobile Devices. http://www.openhandsetalliance.com/press_110507.html, 2007. Accessed: 2017-10-19.

[2] Building great multi-media experiences on Android. https://www.google.com/events/io/io14videos/519d6e77-37b4-e311-b30e-00155d5066d7, 2014. Timestamp: 36:16.

[3] Library 04- USART for STM32F4. http://bit.ly/2zyP7vb, 2014. Accessed: 2017-20-22.

[4] Library 33 - PWM for STM32F4. https://stm32f4-discovery.net/2014/09/library-33-pwm-stm32f4xx/, 2014. Accessed: 2017-10-23.

[5] Library 42 - control rc servo with stm32f4. https://stm32f4-discovery.net/2014/10/library-42-control-rc-servo-stm32f4/, 2014. Accessed: 2017-10-23.

[6] Optimization Techniques - from Intel SSE Intrinsics to Intel Cilk Plus . http://intel.ly/2zMC1f0, 2014. Accessed: 2017-10-24.

[7] Android Reference - CameraCharacteristics. https://developer.android.com/reference/android/hardware/camera2/CameraCharacteristics.html, 2017. Accessed: 2017-08-29.

[8] Android Reference - CameraManager. https://developer.android.com/reference/android/hardware/camera2/CameraManager.html, 2017. Accessed: 2017-08-29.

[9] Android Reference - CaptureRequest. https://developer.android.com/reference/android/hardware/camera2/CaptureRequest.html, 2017. Accessed: 2017-08-29.

[10] Android Reference - CaptureRequestSession. https://developer.android.com/reference/android/hardware/camera2/CameraCaptureSession.html, 2017. Accessed; 2017-08-29.

[11] Android Reference - CaptureResult. https://developer.android.com/reference/android/hardware/camera2/CaptureResult.html, 2017. Accessed: 2017-08-29.

[12] Android Reference - ImageFormat. https://developer.android.com/reference/android/graphics/ImageFormat.html, 2017. Accessed: 2017-10-21.

[13] Android Reference - Platform Architecture. https://developer.android.com/guide/platform/index.html, 2017. Accessed: 2017-10-19.

[14] Content Providers. https://developer.android.com/guide/topics/providers/content-providers.html, 2017. Accessed: 2017-10-20.

[15] Introduction to Activities. https://developer.android.com/guide/components/activities/intro-activities.html, 2017. Accessed: 2017-10-20.

[16] Requesting Permissions. https://developer.android.com/guide/topics/permissions/requesting.html, 2017. Accessed: 2017-10-20.

[17] Resources Overview. https://developer.android.com/guide/topics/resources/overview.html, 2017. Accessed: 2017-10-20.

[18] Smartphone OS Market Share, 2017Q1. https://www.idc.com/promo/smartphone-market-share/os, 2017. Accessed: 2017-10-19.

[19] The Activiy Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.html, 2017. Accessed: 2017-10-20.

[20] UI Overview. https://developer.android.com/guide/topics/ui/overview.html, 2017. Accessed: 2017-10-20.

[21] Horizon Hobby LLC. SPM8800 Manual. https://www.spektrumrc.com/ProdInfo/Files/SPM8800-Manual_DE.pdf. Accessed: 2017-07-21.

[22] Petar Tahchiev Felipe Leme Vincent Massol Gary Gregory. *JUnit in Action.* Manning Publications Co., 2nd edition, 2011.

[23] Horizon Hobby LLC. Specification for Spektrum Remote Receiver Interfacing. https://www.spektrumrc.com/ProdInfo/Files/Remote%20Receiver%20Interfacing%20Rev%20A.pdf, 2016. Accessed: 2017-07-21.

[24] Ziguard Mednieks Liard Dornin G.Blake Meike Masumi Nakamura. *Android Programmierung.* O'Reilly, 2nd edition, 2013.

[25] Michel Rottleuthner. Interprozesskommunikation zwischen Android- und Mikrocontrollersystemen. 2015.

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

Hamburg, 27. Oktober 2017 Wassilij Beaucamp