

ENTWICKLUNG EINES ECHTZEITFÄHIGEN RC-MISCHERS MITTELS ARDUINO

HAUSARBEIT

HOCHSCHULE BOCHUM
Campus Velbert/Heiligenhaus
Höseler Platz 2
42579 Heiligenhaus

INHALTSVERZEICHNIS

1 Motivation	1
2 Grundlagen	2
2.1 Echtzeitfähigkeit	2
2.1.1 Kooperatives Multitasking	2
2.1.2 Präemptives Multitasking	2
2.2 RC – Radio Control	2
2.2.1 RC im Modellflug auf Stand der Technik	2
2.2.2 Übersicht etablierter Protokolle	3
2.3 Arduino Leonardo	4
2.4 Quadrocopter	5
3 Anforderungen	7
4 Entwicklung der Applikation	8
4.1 Arbeitsumgebung	8
4.2 Implementierung der Taskverwaltung	9
4.2.1 Scheduler.h	10
4.2.2 Scheduler.cpp	11
5 Erstellung eines allgemeinen Mischer-Frameworks	19
5.1 Implementierung der Mischerfunktionalität	19
5.1.1 Implementierung der Inputs und Outputs	19
5.1.2 Realisierung des Mischerverhaltens	19
5.2 Vernetzung der Systemkomponenten	20
5.2.1 Anbindung an die Android-Applikation	21
5.2.2 Anbindung an die RC-Fernsteuerung	22
5.2.3 Anbindung an den Flugcontroller	24
6 Tests	27
6.1 Parametrierung des Mischers	27
6.2 Testbedingungen	27
6.3 Ergebnisse	29
7 Zusammenfassung	31
8 Ausblick	32
9 Literatur	33
10 Anhang	34
10.1 Persönliches Fazit	34
10.2 Lizenzen	35
10.3 gitlab-Repository	36
10.4 Eidesstattliche Erklärung	36

ABBILDUNGSVERZEICHNIS

2.1	Typische Wellenform eines PWM-Signals [The01, S. 1]	3
2.2	Typische Wellenform eines PPM-Summen-Signals [The01, S. 1]	3
2.3	Arduino Leonardo kompatibler OLIMEXINO-32U4 – Quelle: Olimex	4
2.4	Block Diagramm des Atmega 32u4 [Atm16, S. 4]	5
2.5	Der in diesem Projekt verwendete Quadrocopter	6
3.1	Naze32 Flugsteuerung – Quelle: commons.wikimedia.org	7
3.2	Spektrum Remote Receiver – Quelle: [Hor16, S. 2]	7
4.1	Split-View \LaTeX - & PlatformIO-Projekt in Atom	8
4.2	Hinzufügen eines Projekt-Stammverzeichnisses in Atom / PlatformIO	9
5.1	Blockdiagramm des Systems	21
6.1	Nahansicht des Versuchsaufbaus im Profil	28
6.2	Draufsicht auf die Oberseite des Versuchsaufbaus. Die Hauptdiagonale zwischen zwei Rotoren beträgt 415mm.	30
6.3	Draufsicht auf die Unterseite des Versuchsaufbaus	30

LISTINGS

4.1	platformio.ini - Konfigurationsbeispiel	9
4.2	Deklaration der verwendeten Datenstrukturen	10
4.3	Festlegen der Threshold-Werte	11
4.4	Konstruktor	11
4.5	setNRtTasks	11
4.6	setRtTasks	12
4.7	sortTasks	13
4.8	convertToLinkedList	14
4.9	newLinkedListElement	15
4.10	sortRtTasks	15
4.11	schedule	16
4.12	perform	17
4.13	setPriorities	18
4.14	exterminate	18
5.1	RC.h	19
5.2	Mixer.h	19
5.3	Mixer::mix()	20
5.4	RawSerial.h - Auszug	21
5.5	RawSerial::getData()	22
5.6	Spektrum.h - Auszug	23
5.7	Spektrum::dataReceive()	24
5.8	MultiWiiSerial.h - Auszug	25
5.9	MultiWiiSerial::SerialEncode()	25
6.1	Parametrisierung des Schedulers	27

1 MOTIVATION

Ziel dieser Arbeit ist die Entwicklung eines echtzeitfähigen Systems, welches eingehende Signale aus mehreren, ggf. unterschiedlichen Quellen erfassen und anschließend zu einem kombinierten Ausgangssignal verarbeiten kann. Konkret sollen hierbei die im Modellbau etablierte Protokolle Spektrum-Remote-Receiver und das Multiwii Serial Protocol erfasst bzw. ausgegeben werden können. Die Applikation soll auf Mikroprozessoren der Reihe Atmega implementiert werden.

Motiviert ist die Entwicklung dieses Systems durch die Notwendigkeit einer Signalkonvertierung und -verarbeitung für ein bedingt autonomes, unbemanntes Flugobjekt. An dieser Stelle der Verweis auf das Schwesterprojekt Visual Based Landing System – kurz VBLS¹. Um dessen sicheren Betrieb zu ermöglichen, sind manuelle Steuerimpulse von einer konventionellen Fernsteuerung mit von einer Android Applikation generierten Steuerimpulsen zu kombinieren. Hierbei muss ein manuelles Eingreifen durch den Menschen jederzeit möglich sein, darüber hinaus muss die Signalverarbeitung in einem fest definierten Zeitrahmen abgeschlossen werden. Ein mögliches Einsatzgebiet ist beispielsweise die Schnittstelle zur Integration diverser Assistenzsysteme in eine manuelle Steuerung (vgl. bspw. Spurhalteassistentensystem im Auto).

¹VBLS - <https://gitlab.cvh-server.de/lf.ps/vbls/tree/master/Visual-Based-Landing-System>

2 GRUNDLAGEN

2.1 Echtzeitfähigkeit

Die Definition von Echtzeit nach DIN 44300 lautet:

Unter Echtzeit versteht man den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen. (vgl. DIN 44300 (Informationsverarbeitung), Teil 9 (Verarbeitungsabläufe))

Wie obenstehende Definition verdeutlicht, sagt der Begriff der Echtzeit nicht direkt etwas über die Geschwindigkeit eines Systems aus. Er beschreibt lediglich ein System, welches schnell genug ist, innerhalb eines vorgegebenen Zeitrahmens, also rechtzeitig, auf Ereignisse zu reagieren. Echtzeitfähigkeit beschreibt in der Informationstechnik also die Fähigkeit, jederzeit Daten innerhalb eines fest definierten Zeitraumes verarbeiten zu können.

2.1.1 Kooperatives Multitasking

Multitasking im Allgemeinen bezeichnet die Verarbeitung einer Vielzahl konkurrierender, quasiparalleler Tasks. Ein Scheduler dient der Ressourcen- und Taskverwaltung. Beim kooperativen Multitasking können Tasks gar nicht oder nur dann unterbrochen werden, wenn sie dazu bereit sind. Unkooperative Tasks können somit das ganze System anhalten. Dieses Verfahren ist nur dann für echtzeitkritische Anwendungen geeignet, wenn die einzelnen Tasks selbst so programmiert sind, dass ihre Ausführungszeit begrenzt ist. Weiterhin muss darauf geachtet werden, dass keine Tasks aufgrund von Nichtbeachtung bzw. Nichtausführung verhungern (engl.: to starve). Dieser Starvation ist vorzubeugen, indem Tasks nicht nur statisch, sondern dynamisch unter Berücksichtigung ihrer letzten Ausführungszeit, priorisiert werden.

2.1.2 Präemptives Multitasking

Beim präemptiven Multitasking können Tasks jederzeit unterbrochen werden. In Kombination mit bspw. Hardware-Interrupts lässt sich so eine harte Echtzeitfähigkeit realisieren.

2.2 RC – Radio Control

Als Radio Control – alternativ Remote Control – bezeichnet man im Allgemeinen die Steuerung aus der Ferne, mittels Funk- oder Licht-Signalen. Nikola Tesla soll bereits 1898 ein funkferngesteuertes Schiffsmodell vorgeführt haben. Einen guten Überblick über die historische Entwicklung der Fernsteuerungstechnik bietet folgender Wikipedia unter dem Stichwort Funkfernsteuerung [Wikib].

2.2.1 RC im Modellflug auf Stand der Technik

Die Digitalisierung hat auch im Modellbau Einzug gehalten. So arbeiten alle heute erhältlichen Anlagen mit einer digitalen Signalübertragung zwischen Fernsteuerung und Empfänger. Hierzu wird i.d.R. ein digitaler Datenstrom auf eine Trägerfrequenz, heute meist die im ISM-Band liegenden 2,4GHz, in einigen Fällen auch die im SRD-Band liegenden 868MHz, aufmoduliert. Die Nutzung dieser Frequenzen ist regional reglementiert. In Deutschland hat die Bundesnetzagentur diese für eine Nutzung wie beispielsweise den Modellflug freigegeben, sofern die verwendeten Anlagen eine entsprechende Zulassung besitzen.

Nach derzeitigem Stand sind mit diesem digitalen Verfahren Funkübertragungen von bis zu 32 Kanälen über Entfernungen mehrerer Kilometer möglich. Die Auflösung der einzelnen übertragenen Kanäle kann hierbei variieren, typisch sind in höherwertigen Anlagen Auflösungen von bis zu 4096 Schritten – 12 Bit – bei Wiederholraten teilweise deutlich über 50Hz.

2.2.2 Übersicht etablierter Protokolle

Die folgend beschriebenen Protokolle beziehen sich ausschließlich auf die Datenübertragung zwischen RC-Empfänger und RC-Komponenten wie Servos, Drehzahlstellern und Flugsteuerungen. Die drahtlose Datenübertragung von RC-Fernsteuerung zu RC-Empfänger ist ein gänzlich anderes, ebenfalls sehr spannendes Thema, welches für das Verständnis dieser Arbeit jedoch nicht weiter relevant ist und somit an dieser Stelle nicht vertieft werden soll.

Analog

Historisch bedingt existieren im Modellbau auch heute noch analoge Signale. Analog bezieht sich in diesem Fall nicht auf den Wertebereich des übertragenen Spannungspegels, sondern auf die der Datenübertragung zugrundeliegende Zeitdomäne. Folglich ist die Dauer des übertragenen High-Pegels abhängig von dem zu übertragenden Wert.

- PWM

Puls-Weiten Modulation – kurz PWM – bezeichnet ein Verfahren, bei dem mit einer gegebenen Frequenz Pulse variabler Länge generiert werden. Die Länge des Pulses ist proportional zu übertragenden Wert. Im Modellbau findet dieses Verfahren zum einen zur Leistungsregelung von Gleichstrom-Motoren Verwendung, zum anderen stellt es den etablierten Standard der Ansteuerung von Servo-Motoren (siehe Abb. 2.1) und Drehzahlstellern, wie etwa bei einem Quadrocopter, von Seiten eines RC-Empfängers oder einer Flugsteuerung dar.

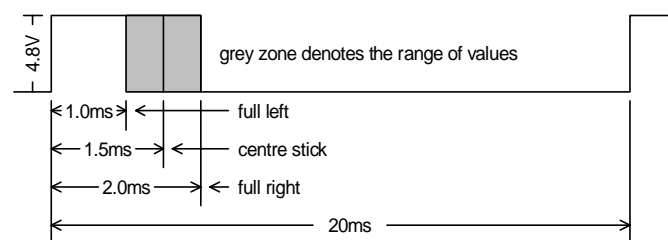


Abbildung 2.1: Typische Wellenform eines PWM-Signals [The01, S. 1]

- PPM

Puls-Positions Modulation – kurz PPM – wird oft auch als PWM-Summensignal bezeichnet. Es findet im Modellbau beispielsweise Verwendung, um bis zu neun PWM-Signale über eine einzige Leitung zu senden. Dies geschieht durch Aneinanderreihung der unter PWM beschriebenen Pulse (siehe Abb. 2.2). Aufgrund der simplen Verdrahtung stellt es eine beliebte Möglichkeit dar, Flugsteuerung und RC-Empfänger zu verbinden.

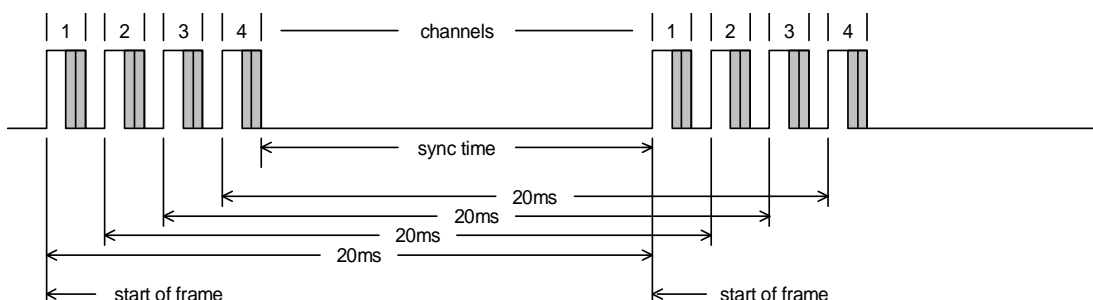


Abbildung 2.2: Typische Wellenform eines PPM-Summen-Signals [The01, S. 1]

Digital

Bei den im Folgenden beschriebenen Protokollen handelt es sich ausschließlich um digitale, unidirektionale & asynchrone serielle Datenströme, welche von einem UART eines Mikrocontrollers generiert und gelesen werden können. Jedes dieser Protokolle ist dazu geeignet, mindestens sechs Steuerkanäle zu übertragen.

- Spektrum-Remote-Receiver

Das Spektrum-Remote-Receiver Protokoll wird von sog. Satelliten-Empfängern des Herstellers Spektrum bzw. Horizon Hobby und dazu kompatiblen Systemen verwendet, um mit einem Hauptempfänger oder einem Flugcontroller zu kommunizieren. Ursprünglich diente es lediglich der Erhöhung der Ausfall- und Übertragungssicherheit der Funkverbindung zwischen Fernsteuerung und Empfänger durch das Vernetzen mehrerer Satelliten-Empfänger mit einem Hauptempfänger.

Mehrere erfolgreiche Reverse-Engineering Ansätze ermöglichten jedoch bald die unabhängige Verwendung dieser kostengünstigen Satellitenempfänger mit Flugsteuerungen. Horizon Hobby reagierte schließlich mit dem begrüßenswerten Entschluss, eine offizielle Spezifikation dieses Protokolls mitsamt einer rudimentären Implementierungsanleitung zu veröffentlichen [Hor16, S. 2].

- SRXL

Das Serial Receiver Link Protocol stellt ein offenes Protokoll dar, mit dem Ziel die Kommunikation zwischen Modellbau-Komponenten unterschiedlicher Hersteller zu ermöglichen. Es können in SRXL Version 2 bis zu 16 Kanäle übertragen werden [Mul13, S. 1].

- MSP

Das MultiWii-Serial Protokoll ist eine offenes Protokoll zur Kommunikation unterschiedlichster Komponenten mit einer Flugsteuerung, auf welcher MultiWii selbst oder eine der Folgeentwicklungen läuft. Es dient nicht primär der Übertragung von Kanaldaten, es bietet darüber hinaus zahlreiche Kommandos zur Konfiguration der Flugsteuerung und zur Abfragung von Telemetriedaten.

Das MSP ermöglicht jedoch das Überschreiben der auf der Flugsteuerung vorliegenden Kanaldaten und somit auch eine direkte Steuerung des entsprechenden Fluggerätes. Leider gibt es zu dem MSP keine zusammenhängende Dokumentation, ein Wiki-Eintrag (vgl. [Mul15]) beschreibt jedoch gut die zur Verfügung stehenden Kommandos, und in dem MultiWii-Forum gibt es einen entsprechenden Beitrag (vgl. [Mul12]) mit sporadischer Aktivität.

2.3 Arduino Leonardo

Der Arduino Leonardo stellt eine Entwicklungsplatine für den Atmel-Atmega-32u4 dar.

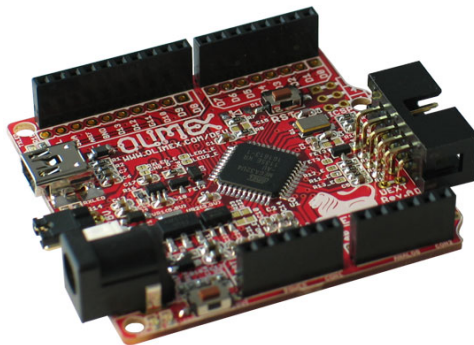


Abbildung 2.3: Arduino Leonardo kompatibler OLIMEXINO-32U4 – Quelle: Olimex

Gewählt wurde der Leonardo, da er der günstigste und kompakteste Arduino ist, welcher neben einer in Hardware implementierten seriellen Schnittstelle auch eine zweite serielle Schnittstelle in Form einer virtuellen USB-CDC Schnittstelle bietet. Diese ist essentiell für die Kommunikation des RC-Mischers mit dem Mobiltelefon. Beide Schnittstellen sind auf dem Blockdiagramm des Atmega 32u4 (siehe Abb. 2.4) zu erkennen, namentlich USB2.0 und USART1. Verwendung fand in diesem Projekt eine Arduino-kompatible Platine (siehe Abb.

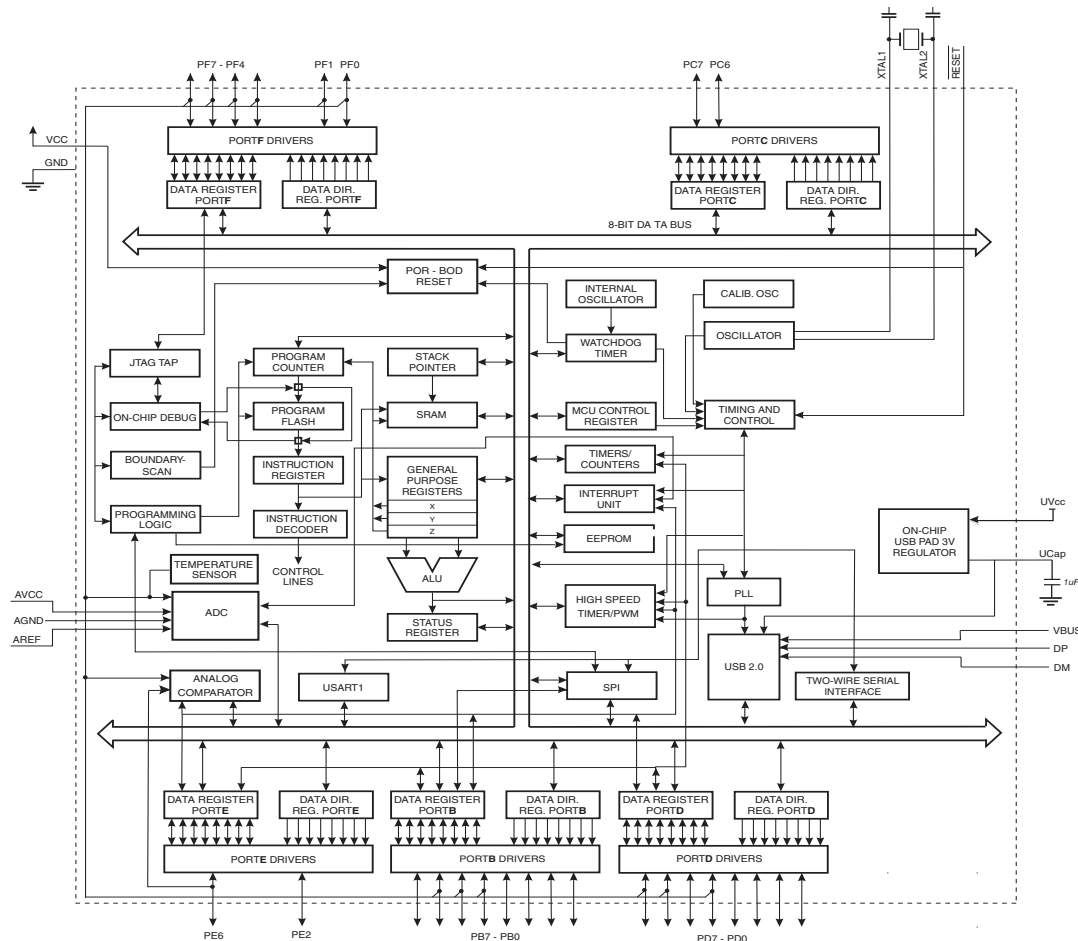


Abbildung 2.4: Block Diagramm des Atmega 32u4 [Atm16, S. 4]

2.3) des bulgarischen Herstellers Olimex¹, welche gegenüber dem offiziellen Arduino Leonardo einige Vorteile sowie ein besseres Preis-Leistungs-Verhältnis bietet.

2.4 Quadrocopter

Als Quadrocopter wird im Allgemeinen ein Fluggerät bezeichnet, welches über vier starr am Rahmen befestigte, drehbar gelagerte Rotoren verfügt. Angetrieben werden diese Rotoren jeweils durch einen Elektromotor. Die Drehzahl dieser Motoren und der meist direkt mit diesen gekoppelten Rotoren kann mittels sog. Fahrtregler gestellt werden. Hierdurch ist eine Steuerung des durch die einzelnen Rotoren generierten Auftriebs möglich.

Eine individuelle Steuerung der Rotorgeschwindigkeiten ermöglicht ein Neigen des Quadrocopters um seine Roll- und Nick-Achse und somit eine Fortbewegung in der Horizontalebene. Durch kollektive Drehzahländerungen aller Rotoren kann die Vertikalgeschwindigkeit des Quadrocopters beeinflusst werden. Für weitergehende Informationen bzgl. der

¹<https://www.olimex.com/Products/Duino/AVR/OLIMEXINO-32U4/open-source-hardware>

Funktionsweise eines Quadropters kann der entsprechende Eintrag in der Wikipedia herangezogen werden [Wika].

Die Regelung der Lage und Geschwindigkeit eines Quadropters erfolgt mittels einer Flugsteuerung an Bord. Diese stellt ein für sich eigenständiges eingebettetes System dar, welches über entsprechende Sensorik zur Lage- und ggf. Positionserkennung verfügt. Die Steuerung eines Quadropters durch den Piloten erfolgt bspw. konventionell mittels einer Funkfernsteuerung oder auch per App über ein Mobiltelefon.

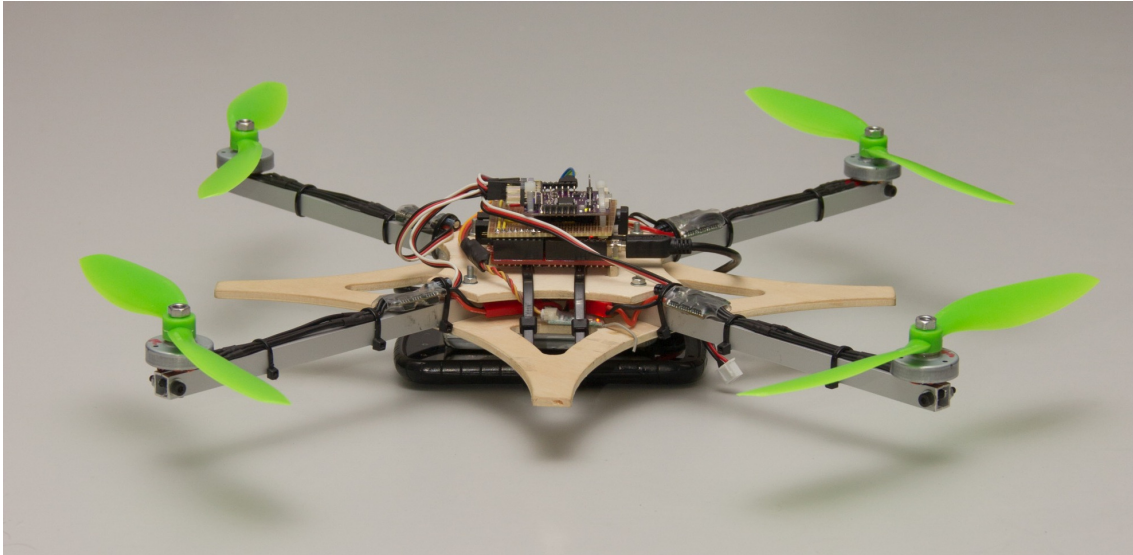


Abbildung 2.5: Der in diesem Projekt verwendete Quadropters

3 ANFORDERUNGEN

Die gegenwärtige Aufgabe des RC-Mischers ist die Signalaggregation, -verarbeitung und -ausgabe mit dem Zweck, folgende Komponenten eines übergeordneten Systems zu vernetzen:

- Eine Flugsteuerung auf Basis eines STM32-F303, auf welchem die freie Software Cleanflight¹ läuft. Diese kommuniziert mittels des MultiWii-Serial-Protocol.

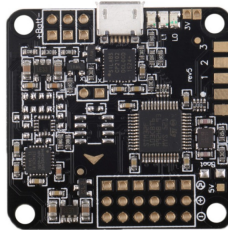


Abbildung 3.1: Naze32 Flugsteuerung – Quelle: commons.wikimedia.org

- Ein RC-Empfänger, welcher das Spektrum-Remote-Receiver Protokoll spricht.



Abbildung 3.2: Spektrum Remote Receiver – Quelle: [Hor16, S. 2]

- Eine Android-Applikation, welche auf einem konventionellen Mobiltelefon läuft und über die USB-OTG Schnittstelle in Form eines virtuellen seriellen Ports mittels eines im Rahmen des Schwesterprojektes VBLS² definierten Protokolls kommuniziert.

Neben diesen funktional-bedingten Anforderungen gilt es insbesondere zwei sicherheitskritische Anforderungen umzusetzen:

- Harte Echtzeitfähigkeit: Signalwiederholungsraten von mindestens 50Hz – daraus resultierend Zykluszeiten kleiner 20ms.
- Operative Sicherheit: Ein manuelles Eingreifen muss zu jedem Zeitpunkt möglich sein. Darüber hinaus muss ein Deaktivieren des Mischer-Verhaltens im Flugbetrieb möglich sein.

Die Umsetzung dieser Anforderungen soll Gegenstand des folgenden Kapitels sein.

¹Cleanflight - <http://cleanflight.com>

²VBLS - <https://gitlab.cvh-server.de/lf.ps/vbbs/tree/master/Visual-Based-Landing-System>

4 ENTWICKLUNG DER APPLIKATION

4.1 Arbeitsumgebung

Die Wahl der Arbeitsumgebung und der verwendeten Software-Werkzeuge lässt sich in einem Wort ausdrücken: PlatformIO¹

PlatformIO is an open source ecosystem for IoT development
Cross-platform build system and library manager. Continuous and IDE integration. Arduino, ESP8266 and ARM mbed compatible. [Pla17]

Auch wenn dieses Zitat von der PlatformIO Web-Präsenz prägnant zusammenfasst, was diese Arbeitsumgebung leistet, so möchten wir im Folgenden doch kurz ihre Vorteile und unsere Entscheidung für diese Umgebung erläutern. PlatformIO ist eine plattformübergreifende Entwicklungsumgebung, welche auf dem Atom-Texteditor² aufsetzt und diesen neben zahlreichen Funktionen um mehrere Toolchains zur Entwicklung auf eingebetteten Systemen ergänzt. Derzeit unterstützt werden folgende Plattformen (Auswahl): Atmel AVR & SAM, Espressif, Freescale Kinetis, ST STM32, TI MSP430 & Tiva, Teensy, Arduino, mbed u. a. (vgl. [Pla17]). Konkret Verwendung findet in diesem Projekt die Arduino- und Atmel-AVR-Unterstützung. So greifen wir beispielsweise für die serielle Kommunikation zwischen den einzelnen Hardware-Komponenten auf Arduino-Bibliotheken zurück. Das Arduino-Framework basiert auf C++, welches auch als Programmiersprache für dieses Projekt genutzt wird. Kompiliert wird mittels avr-gcc, für das Schreiben des Flash-Speichers des Mikrocontrollers wird AVRDUDE genutzt. All diese Programme installiert PlatformIO als Abhängigkeiten mit, wenn man es mittels des Atom-eigenen Paketverwalters installiert und den Atmel-AVR als Zielplattform auswählt. Atom stellt darüber hinaus einen sehr mächtigen Texteditor für zahlreiche Betriebssysteme dar, und lässt sich um Funktionen wie einen PDF-Betrachter und LaTeX-Unterstützung erweitern (siehe Abb. 4.1). Atom selbst ist ebenso wie PlatformIO,

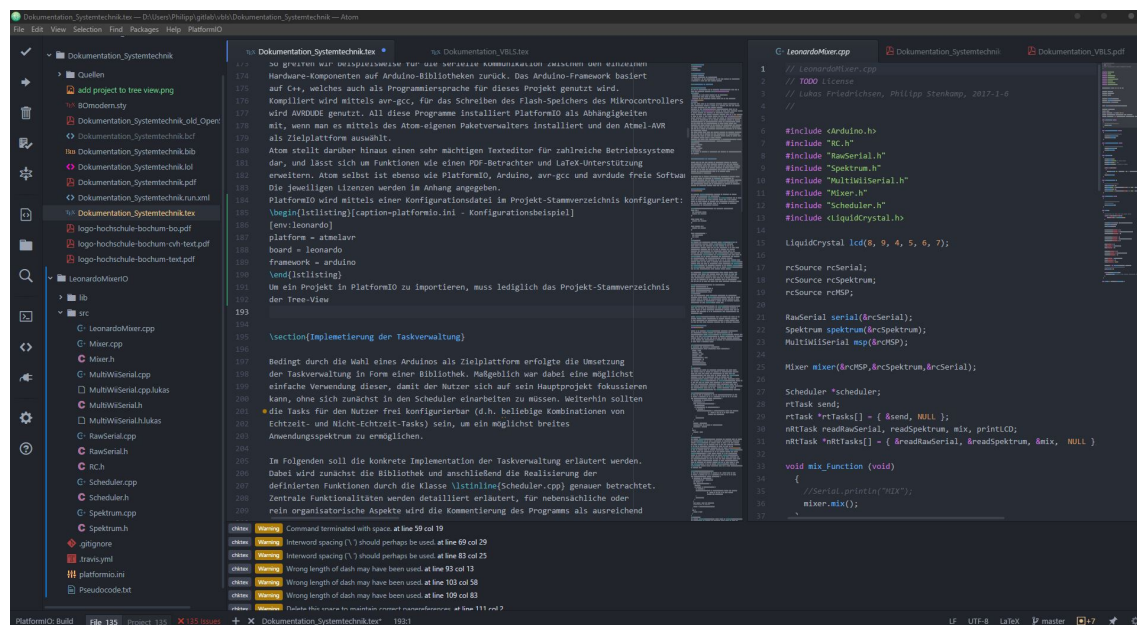


Abbildung 4.1: Split-View L^AT_EX- & PlatformIO-Projekt in Atom

¹PlatformIO - <http://platformio.org/>

²Atom Texteditor - <https://atom.io/>

Arduino, avr-gcc und avrdude freie Software. Die jeweiligen Lizenzen werden im Anhang angegeben. PlatformIO wird mittels einer Konfigurationsdatei im Projekt-Stammverzeichnis konfiguriert:

Listing 4.1: platformio.ini - Konfigurationsbeispiel

```
1 [env:leonardo]
2 platform = atmelavr
3 board = leonardo
4 framework = arduino
```

Um ein Projekt in PlatformIO zu importieren, muss lediglich das Projekt-Stammverzeichnis der Tree-View in Atom hinzugefügt werden (siehe Abb. 4.2).

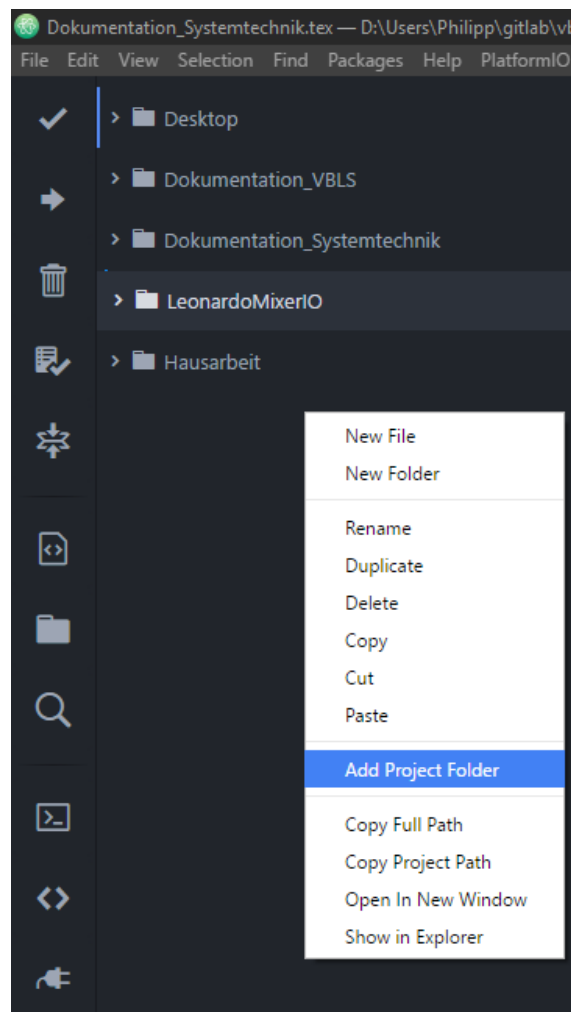


Abbildung 4.2: Hinzufügen eines Projekt-Stammverzeichnisses in Atom / PlatformIO

4.2 Implementierung der Taskverwaltung

Bedingt durch die Wahl eines Arduinos als Zielplattform erfolgte die Umsetzung der Taskverwaltung in Form einer Bibliothek. Maßgeblich war dabei eine möglichst einfache Verwendung dieser, damit der Nutzer sich auf sein Hauptprojekt fokussieren kann, ohne sich zunächst in den Scheduler einarbeiten zu müssen. Weiterhin sollten die Tasks für den Nutzer frei konfigurierbar (d. h. beliebige Kombinationen von Echtzeit- und Nicht-Echtzeit-Tasks) sein, um ein möglichst breites Anwendungsspektrum zu ermöglichen.

Im Folgenden soll die konkrete Implementation der Taskverwaltung erläutert werden. Dabei wird zunächst die Bibliothek und anschließend die Realisierung der definierten Funktionen durch die Klasse `Scheduler.cpp` genauer betrachtet. Zentrale Funktionalitäten werden detailliert erläutert, für nebensächliche oder rein organisatorische Aspekte wird die Kommentierung des Programms als ausreichend erachtet. Der vollständige Quellcode findet sich im Anhang.

4.2.1 Scheduler.h

Die Bibliothek `Scheduler.h` deklariert die Funktionen und globalen Variablen der Implementation der Taskverwaltung. Mittels der Schlüsselwörter `public` und `private` kann an dieser Stelle die Schnittstelle zum Nutzer festgelegt werden; d. h. es kann definiert werden, auf welche Elemente von außerhalb (ohne Überschreiben) zugegriffen werden kann und auf welche nicht. Weiterhin werden in der Bibliothek die verwendeten Datenstrukturen und Threshold-Werte definiert.

Listing 4.2: Deklaration der verwendeten Datenstrukturen

```
1 struct task {
2     void (*activity) (void);
3     unsigned long timestamp;
4 };

6 struct rtTask: task {
7     unsigned long cycleTime;
8 };

10 struct nRtTask: task {
11     unsigned long priority;
12 };

14 struct linkedListElement {
15     task *listElement;
16     void *next;
17 };
```

Der Scheduler kann Realtime-Tasks (abgekürzt mittels „rtTask“) und Non-Realtime-Tasks (abgekürzt mittels „nRtTask“) verwalten. Dabei besteht jeder Task aus einer Funktion, die ihm bei der Initialisierung zugewiesen werden kann und einem Zeitstempel, in dem seine letzte Ausführungszeit gespeichert wird. Es ist dabei zu beachten, dass lediglich Funktionen ohne Übergabeparameter und Rückgabe-Werte zulässig sind, da der Scheduler eine reine Taskverwaltung und keine datenverarbeitende Instanz darstellt. Der Datenaustausch zwischen den Tasks ist über einen gemeinsamen Speicherbereich in Form globaler Variablen wie bspw. dem `rcSource`-Struct möglich.

Die Datenstrukturen `rtTask` und `nRtTask` erben beide direkt von `task` mit dem einzigen Unterschied, dass Realtime-Tasks eine Zykluszeit und Non-Realtime-Tasks eine Ausführungspriorität zugewiesen bekommen. Die Vererbung ermöglicht es, Funktionen, die von der Art des Tasks unabhängig sind, lediglich einmal für beide Typen zu definieren (vgl. beispielsweise `sortTasks`). Weiterhin ergibt sich durch die grundsätzlich gleiche Datenstruktur von `rtTask` und `nRtTask` die Möglichkeit, äquivalent auf die Attribute `cycleTime` und `priority` zuzugreifen (d. h. über die Funktion `getTaskCycleTime` kann bei einem Non-Realtime-Task dessen Priorität erhalten werden).

Die Datenstruktur `linkedListElement` stellt einzelne Elemente einer verketteten Liste bereit. Diese enthalten jeweils einen Pointer auf einen Task (und sind damit zunächst unabhängig davon, ob es sich um einen Realtime- oder einen Non-Realtime-Task handelt) und einen Pointer auf das nächste Listenelement. Die durch die Verkettung der Elemente entstehende Liste ist unidirektional, d. h. sie kann nur von vorne nach hinten, nicht jedoch andersherum durchlaufen werden.

Listing 4.3: Festlegen der Threshold-Werte

```
1 #define MAX_TASK_THRESHOLD 10
2 #define OVERLOAD_THRESHOLD_PERCENT 10
3 #define OVERLOAD_THRESHOLD_TIMES 100
4 #define STARVATION_THRESHOLD 100000
```

Über die Threshold-Werte können verschiedene Eigenschaften der Taskverwaltung beeinflusst werden. Mittels `MAX_TASK_THRESHOLD` lässt sich die maximal zulässige Anzahl an Realtime- und Non-Realtime-Tasks (jeweils) festlegen. Dies dient dazu, einer Überlast vorzubeugen und sicherzustellen zu können, dass das übergebene Task-Array korrekterweise mit einer NULL beendet wird.

`OVERLOAD_THRESHOLD_PERCENT` und `OVERLOAD_THRESHOLD_TIMES` legen fest, wie häufig ein Realtime-Task bei seiner tatsächlichen Ausführung um wieviel Prozent von seiner vorgesehenen Ausführungszeit abweichen darf, bevor der Scheduler sich aus Überlast abschaltet. Somit wird die Echtzeitfähigkeit während des Betriebs gewährleistet.

Mittels `STARVATION_THRESHOLD` lässt sich weiterhin eine Grenze für die Differenz zwischen letzter Ausführungszeit und aktueller Systemzeit festlegen, ab deren Überschreitung ein Non-Realtime-Task temporär in seiner Priorität erhöht und somit für einen Zyklus priorisiert ausgeführt wird. Durch diese Maßnahme wird Starvation (s. Kapitel 2.1.1) vorzubeugen.

4.2.2 Scheduler.cpp

Während in der Bibliothek `Scheduler.h` grundlegende Datenstrukturen, die Funktionsnamen inklusive Übergabe- und Rückgabe-Parametern und die Nutzer-Schnittstelle mittels `public` und `private` definiert wurden, folgt in der Datei `Scheduler.cpp` die konkrete Implementierung der Funktionen. Das Hauptaugenmerk lag hierbei auf einer effizienten Ausführung, um Echtzeitfähigkeit mit den begrenzten Ressourcen eines Mikrocontrollers zu realisieren.

Listing 4.4: Konstruktor

```
1 // Constructor with task-array as input-argument
2 Scheduler::Scheduler (rtTask **newRtTasks, nRtTask **newNRtTask) {
3     rtTasks = NULL;
4     nRtTasks = NULL;
5     taskCounter = NULL;
6     debugger = NULL;
7     listStoragePointer = NULL;
8     overloadCounter = 0;
9     nRtTaskCounter = 0;
10    setRtTasks(newRtTasks);
11    setNRtTasks(newNRtTask);
12 }
```

Der Scheduler wird mittels des Konstruktors initialisiert. Beim Aufruf können zwei separate Arrays mit Pointern auf Realtime- und Non-Realtime-Tasks übergeben werden (das erste Array enthält alle Realtime-Tasks, das zweite Array alle Non-Realtime-Tasks). Beide Array müssen als letzten Eintrag eine NULL enthalten, um das Ende zu kennzeichnen. Sollen nur Echtzeit- oder nur Nicht-Echtzeit-Tasks verwaltet werden, oder sollen die Arrays eigenständig mittels der Funktionen `setRtTasks(...)` und `setNRtTasks(...)` gesetzt werden, so kann dies durch die Übergabe eines Nullpointers an der entsprechend anderen Stelle beim Konstruktoraufruf signalisiert werden.

Listing 4.5: setNRtTasks

```
1 // Sorts and sets the input-array as the new non-realtime-task-
2 // array; the last
3 // element of newNRtTasks HAS TO BE A NULLPOINTER to mark the end
4 // of the array
5 void Scheduler::setNRtTasks (nRtTask **newNRtTasks) {
```



```
6  if (newNRtTasks && newNRtTasks[0]) {
7      nRtTaskCounter = 0;
8      while (newNRtTasks[nRtTaskCounter]) {
9          nRtTaskCounter++;
10         if (nRtTaskCounter >= MAX_TASK_THRESHOLD) {
11             if (debugger) {
12                 debugger->println("Number of non-realtime-tasks exceeds
13                 task threshold or nullpointer at the end of the array
14                 missing! Shutting down scheduler!");
15             }
16             exterminate();
17         }
18     }
19     nRtTasks = newNRtTasks;
20     sortTasks((task **) nRtTasks, 0, nRtTaskCounter);
21     taskCounter = nRtTasks;
22 }
23 else {
24     nRtTasks = NULL;
25 }
26 }
```

Die Funktion `setNRtTasks(...)` verarbeitet das übergebene Array mit Zeigern auf Non-Real-time-Task vor und weist es der Programmintern verwendeten Variable `nRtTasks` zu. Zunächst werden die Tasks in dem Array gezählt. Wie bereits erwähnt, muss das übergebene Array am Ende einen Nullpointer enthalten, um das Ende zu kennzeichnen. Überschreitet die Anzahl der Tasks den für `MAX_TASK_THRESHOLD` festgelegten Wert (siehe Kapitel 4.2.2), so schaltet sich der Scheduler mittels `exterminate()` ab (bzw. konkret wechselt er in eine Dauerschleife). Nachdem die Tasks gezählt wurden, werden sie mittels `sortTasks(...)` ihrer Priorität nach sortiert. Dies hat zur Folge, dass das Array anschließend für die Ausführungsreihenfolge der Tasks einfach von vorne nach hinten durchlaufen werden kann (Effizienzsteigerung). Wurde ein Nullpointer als Parameter übergeben oder ist der erste Eintrag im Array eine `NULL`, so wird die Variable `nRtTasks` ebenfalls auf `NULL` gesetzt, wodurch signalisiert wird, dass es keine Nicht-Echtzeit-Tasks zu verwalten gibt.

Listing 4.6: setRtTasks

```
1 // Sorts the realtime-task-array and converts it to a linked list;
2 // the last element of newRtTask HAS TO BE A NULLPOINTER to mark
3 // the end of the array
4 void Scheduler::setRtTasks (rtTask **newRtTasks) {
5     if (newRtTasks && newRtTasks[0]) {
6         listStoragePointer = listStorage;
7         int counter = 0;
8         while (newRtTasks[counter]) {
9             counter++;
10            if (counter >= MAX_TASK_THRESHOLD) {
11                if (debugger) {
12                    debugger->println("Number of non-realtime-tasks exceeds
13                    task threshold or nullpointer at the end of the array
14                    missing! Shutting down scheduler!");
15                }
16                exterminate();
17            }
18        }
19        task **temp = (task **) newRtTasks;
20        sortTasks(temp, 0, counter);
21    }
22 }
```



```

21     rtTasks = convertToLinkedList(temp);
22 }
23 else {
24     rtTasks = NULL;
25 }
26 }

```

Die Funktion `setRtTasks(...)` dient, ebenso wie `setNRtTasks(...)` der Vorverarbeitung und Zuweisung des übergebenen Arrays, dieses Mal jedoch die Echtzeit-kritischen-Tasks betreffend. Der Aufbau Funktion ist analog zu `setNRtTasks(...)`, bis auf den Unterschied, dass das übergebene Array an dieser Stelle einmalig nach den Zykluszeiten der Realtime-Tasks und nicht nach den Prioritäten sortiert wird. Dadurch wird erreicht, dass die Tasks in der Reihenfolge der Zeitpunkte, wann sie das erste Mal ausgeführt werden müssen, stehen. Anschließend wird das Array mittels `convertToLinkedList(...)` in eine einfach verkettete Liste umgewandelt, da es sich in diesem Datenformat simpler gestaltet, die Echtzeit-Tasks nach ihrer Ausführung an die Stelle ihrer nächsten Ausführungszeit in der Warteschlange einzusortieren.

Listing 4.7: sortTasks

```

1 // Sorts the task-array so that the tasks are in order of their
2 // cycle-time / priority if a non-realtime-task-array is given,
3 // getTaskCycleTime will return the tasks priority left is the
4 // first, right the last position in the array to be sorted
5 void Scheduler::sortTasks(task **tasks, int left, int right) {
6     if (left < right && (right-left+1) > 1 && tasks && *tasks) {
7         // Heapsort (always O(n*log(n)); iterative)
8         int length, start, heapLength, i, j;
9         task *reference;

11        start = left;
12        heapLength = right-left+1;
13        length = heapLength/2+1;

14
15        while (heapLength > 1) {
16            if (length > 1) {
17                reference = tasks[start+ (--length-1)];
18            }
19            else {
20                reference = tasks[start+heapLength-1];
21                tasks[start+heapLength-1] = tasks[start];
22                heapLength--;
23            }
24            i = length;
25            j = length*2;
26            while (j <= heapLength && heapLength > 1) {
27                if (j < heapLength && getTaskCycleTime((rtTask *) tasks[
28                    start+j-1]) < getTaskCycleTime((rtTask *) tasks[start+j]))
29                {
30                    j++;
31                }
32                if (getTaskCycleTime((rtTask *) reference) <
33                    getTaskCycleTime((rtTask *) tasks[start+j-1])) {
34                    tasks[start+i-1] = tasks[start+j-1];
35                    i = j;
36                    j *= 2;
37                }

```

```
38         else {
39             j = heapLength + 1; // Terminates the shift-down
40         }
41     }
42     tasks[start+i-1] = reference;
43 }
44 }
45 }
```

Bei `sortTasks(...)` handelt es sich um einen Heapsort-Algorithmus mit einer Ordnung von $\mathcal{O}(n \cdot \log n)$. Dieser sortiert das übergebene Array entweder nach Zykluszeiten oder nach Prioritäten (abhängig davon, ob ein Array mit Pointern auf Realtime- oder auf Non-Realtime-Task übergeben wurde). Dadurch, dass als Typ des Übergabeparameters die Grund-Datenstruktur `task` gewählt wurde, können sowohl Echtzeit- als auch Nicht-Echtzeit-Task-Arrays verarbeitet werden. Auf die Parameter `cycle-time` bzw. `priority` kann, da, wie bereits in Kapitel 4.2.1 angeführt, sowohl `rtTask` als auch `nRtTask` die gleiche Speicherstruktur haben, analog über die Funktion `getTaskCycleTime(...)` zugegriffen werden. Der Vorteil des Heapsort-Algorithmus gegenüber Sortierverfahren wie bspw. dem Quicksort besteht darin, dass ersterer iterativ abläuft, während letzterer rekursiv aufgerufen wird. Somit wird einem Überlaufen des Stacks des Mikrokontrollers vorgebeugt. Die bereits erwähnte Ordnung des Heapsorts ist darüber hinaus unabhängig vom Inhalt des zu sortierenden Arrays, sie ist konstant. Sein Verhalten ist folglich deterministisch, was ihn für die Verwendung in echtzeitkritischen Anwendung prädestiniert. Dies ist durchaus insofern von Relevanz, da die Neusortierung von Nicht-Echtzeit-Tasks z. B. bei Starvation eines Tasks auch während des laufenden Betriebs erfolgt.

Anmerkung 1: Für eine geringe Anzahl an Tasks bzw. für den Fall, dass Starvation kein Problem in der gegebenen Anwendung darstellen sollte, kann für eine Performanceverbesserung ein Algorithmus wie bspw. Bubblesort verwendet werden. Ist hingegen nicht zwangsläufig gegeben, dass die zu sortierenden Arrays weitestgehend vorsortiert sind, ist der Heapsort aufgrund seines deterministischen Verhaltens zu präferieren.

Anmerkung 2: Für die Herleitung und Grundlagen des Heapsort-Algorithmus vergleiche [Sed92, S.186-192] und [Bau81, S.191-197]. Die hiesige Implementation orientiert sich an [Pre+92, S.336-338].

Listing 4.8: `convertToLinkedList`

```
1 // Converts the given realtime-task-array to a bidirectional
2 // linked list element
3 linkedListElement * Scheduler::convertToLinkedList (task **tasks)
4 {
5     task **taskArray = tasks;
6     linkedListElement *firstElement = NULL;
7     linkedListElement *listElementPointer = NULL;
8     while (*taskArray) {
9         linkedListElement *new_pointer = newLinkedListElement();
10        new_pointer->listElement = *taskArray;
11        new_pointer->next = NULL;
12        if (listElementPointer) {
13            listElementPointer->next = new_pointer;
14        }
15        else {
16            firstElement = new_pointer;
17        }
18        listElementPointer = new_pointer;
19        taskArray++;
20    }
```

```

20     }
21     return firstElement;
22 }

```

Mittels `convertToLinkedList(...)` wird das übergebene Array mit Pointern auf Tasks in eine einfach verkettete Liste umgewandelt und der Zeiger auf das erste Element der Liste zurückgegeben. Grundsätzlich ist die Implementierung der Funktion unabhängig davon, ob Realtime- oder Non-Realtime-Tasks übergeben werden. Konkret werden jedoch lediglich die Echtzeit-Tasks in Listenform verwaltet, da sich durch den Zugriffsalgorithmus an dieser Stelle Zeiteinsparungen erzielen lassen, während sich die Effizienz bei Nicht-Echtzeit-Tasks gegenüber der Array-Speicherform verschlechtern würde (Heapsort ist besser für Arrays als für einfach verkettete Listen geeignet).

Zunächst wird mit `new_pointer` ein Zeiger auf ein mittels der Funktion `newLinkedListElement()` erzeugtes Listenelement initialisiert. Diesem werden der Task an der aktuellen Stelle des übergebenen Task-Arrays und ein Nullpointer als Adresse des nächsten Elements zugewiesen. Anschließend wird überprüft, ob bereits ein vorangegangenes Listenelement existiert. Ist dies der Fall, wird `listElementPointer->next` auf die Adresse des neu erzeugten Elements gesetzt. Falls nicht, wird die Adresse als Einstiegspunkt der Liste in `firstElement` gespeichert und später zurückgegeben. Danach wird mit `listElementPointer` die Variable in der die Adresse des letzten erzeugten `LinkedListElement` gespeichert ist, auf `new_pointer` gesetzt und das übergebene Task-Array um einen hochgezählt (so, dass an erster Stelle des Arrays die Adresse des nächsten Tasks steht). Nachdem das Ende des Arrays erreicht wurde, wird `firstElement` als erstes Element und damit Einstiegspunkt der Liste zurückgegeben.

Listing 4.9: `newLinkedListElement`

```

1 // Returns a pointer to a new linked list element from the array
  ↪ listStorage
2 LinkedListElement * Scheduler::newLinkedListElement (void){
3     if (listStoragePointer){
4         return listStoragePointer++;
5     }
6     else {
7         if (debugger) {
8             debugger->println("Can't initialize any more linked list
9                               elements! Shutting down scheduler!");
10        }
11        terminate();
12    }
13 }

```

Die Funktion `newLinkedListElement()` gibt als Rückgabewert einen mit jedem Aufruf fortlaufenden Zeiger auf ein `LinkedListElement` aus einem in `setRtTasks(...)` initialisierten Array der Größe `MAX_TASK_THRESHOLD` (da dies der maximalen Zahl an zu erstellenden Elementen der verknüpften Liste entspricht) zurück. Durch diese virtuelle Speicherverwaltung kann der Verwendung des relativ aufwendigen `new` bei der dynamischen Erzeugung einer verknüpften Liste mittels `convertToLinkedList(...)` vorgebeugt werden.

Listing 4.10: `sortRtTasks`

```

1 // Places the given element in the linked list based on when it
2 // has to be called next; if only one realtime-task exists,
3 // the list isn't modified
4 LinkedListElement * Scheduler::sortRtTasks (LinkedListElement
5 *tasks) {
6     LinkedListElement *reference = tasks;
7     LinkedListElement *counter = (LinkedListElement *) reference->
8     next;
9     if (counter && (signed long)(getTaskCycleTime((rtTask *)

```

```

10 reference->listElement)-getTaskTimerDiff(reference->
11 listElement)) > (signed long)(getTaskCycleTime((rtTask *)
12 counter->listElement)-getTaskTimerDiff(counter->listElement))) {
13     while (counter->next && (signed long)(getTaskCycleTime((rtTask
14 *) reference->listElement)-getTaskTimerDiff(reference->
15 listElement)) > (signed long)(getTaskCycleTime((rtTask *)
16 counter->listElement)-getTaskTimerDiff(counter->listElement)))
17     {
18         counter = (linkedListElement *) counter->next;
19     }
20     linkedListElement *temp = (linkedListElement *) reference->
21 next;
22     reference->next = counter->next;
23     counter->next = reference;
24     return temp;
25 }
26 else {
27     return reference;
28 }
29 }

```

Die Funktion `sortRtTasks(...)` wird direkt nach Ausführen eines Realtime-Tasks aufgerufen und ordnet das übergebene (standardmäßig das gerade ausgeführte) Element der verknüpften Liste abhängig von seinem nächsten Ausführzeitpunkt in die Liste ein. Grundvoraussetzung dafür ist, dass die Liste in sortierter Form vorliegt (erfüllt durch den Aufruf von `sortTasks(...)` vor der Umwandlung mittels `convertToLinkedList(...)`). Dafür wird zunächst überprüft, ob ein weiteres Listen-Element existiert und ob der Zeitpunkt der nächsten Ausführung vor dem Zeitpunkt des übergebenen Elements liegt. Ist dies nicht der Fall, wird die Liste nicht modifiziert und der Zeiger auf das aktuelle Element zurückgegeben. Ansonsten werden die Elemente solange durchlaufen, wie die Ausführungszeit des betrachteten Tasks vor derjenigen des übergebenen Tasks liegt oder bis die Liste zu Ende ist. Anschließend wird das übergebene Element an der entsprechenden Stelle einsortiert und der Zeiger auf den ehemals zweiten Task (der jetzt der als Nächstes auszuführende ist) zurückgegeben. Somit wird im schlechtesten Fall eine Ordnung von $\mathcal{O}(n)$ erreicht, während der äquivalente Vorgang in einem herkömmlichen Array eine Ordnung von $\mathcal{O}(n^2)$ hätte.

Listing 4.11: schedule

```

1 // Scheduler, called cyclical in the loop-function
2 void Scheduler::schedule (void) {
3     if (rtTasks) {
4         unsigned long timerDiff = getTaskTimerDiff(rtTasks->
5 listElement);
6         unsigned long cycleTime = getTaskCycleTime((rtTask *) rtTasks
7 ->listElement);
8         if (timerDiff >= cycleTime) {
9             if (((timerDiff-cycleTime)*100/cycleTime) >
10 OVERLOAD_THRESHOLD_PERCENT) {
11                 overloadCounter++;
12                 if (overloadCounter > OVERLOAD_THRESHOLD_TIMES) {
13                     if (debugger) {
14                         debugger->println("Capacity overload! Delay between
15 theoretical and actual cycle-time exceeds threshold!
16 Shutting down scheduler!");
17                     }
18                     exterminate();
19                 }

```

```

20     }
21     rtTasks->listElement->activity();
22     updateTaskTimer(rtTasks->listElement);
23     rtTasks = sortRtTasks(rtTasks);
24     if (nRtTasks){
25         setPriorities();
26         taskCounter = nRtTasks;
27     }
28 }
29 else if (nRtTasks) {
30     perform();
31 }
32 }
33 else if (nRtTasks) {
34     perform();
35 }
36 else {
37     if (debugger) {
38         debugger->println("No tasks... Nothing to do!");
39     }
40     // No tasks... Nothing to do!
41 }
42 }

```

Die Funktion `schedule()` wird zyklisch vom Mikrocontroller aufgerufen und bildet das Herzstück der Taskverwaltung. Grundsätzlich werden drei möglich Fälle unterschieden:

- Es sind keine zu verwaltenden Tasks vorhanden (sowohl `rtTasks` als auch `nRtTasks` sind gleich `NULL`). In diesem Fall gibt der Scheduler bei übergebenem seriellen Debugging-Port eine Statusmeldung aus und macht nichts.
- Es sind lediglich Non-Realtime-Tasks vorhanden (`rtTasks` ist gleich `NULL`). In diesem Fall ruft der Scheduler die Funktion `perform()` auf, die für die Verwaltung der Nicht-Echtzeit-Tasks zuständig ist.
- Es sind Realtime-Tasks vorhanden. In diesem Fall überprüft der Scheduler zunächst, ob innerhalb des aktuellen Zyklus noch Zeit für die Ausführung von Non-Realtime-Tasks ist. Hierzu wird der vorraussichtliche Ausführungszeitpunkt des nächsten Realtime-Tasks mit der aktuellen Zykluszeit verglichen. Ist noch Zeit übrig und sind Non-Realtime-Tasks vorhanden, so wird `perform()` aufgerufen. Ansonsten wird der Realtime-Task abgearbeitet, sein Timestamp wird aktualisiert und er wird abhängig von seinem nächsten vorgesehenen Ausführungszeitpunkt mittels `sortRtTasks(...)` in die einfach verkettete Liste der Echtzeit-Tasks einsortiert. Existieren weiterhin zu verwaltende Non-Realtime-Tasks, so wird abschließend die Funktion `setPriorities()` aufgerufen, um gegebenenfalls mittels dynamischer Prioritätenvergabe Starvation zu verhindern. Abschließend wird der Zeiger auf den aktuellen Nicht-Echtzeit-Task zurückgesetzt. Es handelt sich bei dieser Art der Taskverwaltung um einen reaktive Ansatz (auf den Ausführungszeitpunkt eines Echtzeit-Tasks wird nach dessen Eintreten reagiert).

Listing 4.12: perform

```

1 // Handles the functions excluding send
2 void Scheduler::perform (void) {
3     if (taskCounter && *taskCounter) {
4         (*taskCounter)->activity();
5         updateTaskTimer((task *) *taskCounter);
6         taskCounter++;
7     }

```

```

8  else if (nRtTasks) {
9      taskCounter = nRtTasks;
10 }
11 }

```

Die Funktion `perform()` verwaltet die Ausführung der Non-Realtime-Tasks. `taskCounter` zeigt dabei immer auf die Speicheradresse des aktuellen Tasks. Bei jedem Aufruf der Funktion wird dieser abgearbeitet, sein Timestamp aktualisiert und `taskCounter` inkrementiert, so dass er auf den nächsten Task zeigt. Sobald das Ende des Arrays erreicht wurde (gekennzeichnet durch den Eintrag `NULL`), wird der Zeiger erneut auf die Startadresse von `nRtTasks` gesetzt. Diese Form der Abarbeitung der Tasks ist trotz unterschiedlichen Priorisierungen möglich, da sie innerhalb des Arrays dank `sortTasks(...)` bereits in nach Priorität sortierter Form vorliegen.

Listing 4.13: setPriorities

```

1 // Dynamic prioritiy-allocation to prevent starvation
2 void Scheduler::setPriorities (void) {
3     sortTasks((task **) nRtTasks, 0, nRtTaskCounter);
4     taskCounter = nRtTasks;
5     int counter = 0;
6     while (taskCounter && *taskCounter){
7         if (getTaskTimerDiff((task *) *taskCounter) >=
8             ↪ STARVATION_THRESHOLD){
9             nRtTask *temp = *taskCounter;
10            memmove(nRtTasks+1, nRtTasks, counter*sizeof(void *));
11            *nRtTasks = temp;
12        }
13        counter++;
14        taskCounter++;
15    }
16 }

```

Mittels der Funktion `setPriorities()` lassen sich während der Laufzeit der Taskverwaltung durch Verschiebung der Non-Realtime-Tasks innerhalb des Arrays temporär neue Prioritäten vergeben um so Starvation vorzubeugen. Dabei wird das Array zunächst erneut sortiert, um eine mögliche temporäre Prioritätsänderung des vorangegangenen Zyklus zu überschreiben. Anschließend wird `nRtTasks` von Beginn an durchlaufen und für jeden Task überprüft, ob die Differenz zwischen dessen letzter Ausführungszeit und der aktuellen Systemlaufzeit den in `STARVATION_THRESHOLD` festgelegten Wert überschreitet. Ist dies der Fall, so wird der Zeiger auf den Task in der Hilfsvariable `temp` zwischengespeichert, alle vorangegangenen Tasks mittels `memmove(...)` um einen Eintrag weiter verschoben und anschließend der „verhungerte“ Task an erster Stelle wieder eingefügt.

Listing 4.14: exterminate

```

1 // The system switches to a defined state in case of an error,
2 // effectively doing nothing.
3 void Scheduler::exterminate (void){
4     if (debugger) {
5         debugger->println("An error occured! Programm terminated!");
6     }
7     while (true);
8 }

```

Mit Hilfe der Funktion `exterminate()` kann der Scheduler beim Auftreten eines Fehlers in einen definierten Zustand überführt werden. Erreicht wird dies mittels des Eintretens in eine Dauerschleife. Somit wird verhindert, dass unvorhersehbares Verhalten auftritt.

5 ERSTELLUNG EINES ALLGEMEINEN MISCHER-FRAMEWORKS

5.1 Implementierung der Mischerfunktionalität

In Anbetracht der großen Anzahl an bereits existierenden Protokollen zur Kommunikation mit Modellbau-Empfängern und in Hinblick auf die Erweiterbarkeit und Portierbarkeit dieses Projektes erschien es als angemessen und sinnvoll, den RC-Mischer modular zu gestalten. Somit entstand in Verbindung mit dem bereits vorgestellten Scheduler ein möglichst allgemein gehaltenes Mischer-Framework, welches im Modellbau übliche Signalen aus unterschiedlichsten Quellen verarbeiten und mischen kann.

5.1.1 Implementierung der Inputs und Outputs

Der hier vorgestellte Mischer soll mit Daten von unterschiedlichen Funk-Fernsteuerungen aus dem Modellbau-Bereich umgehen können. Im Modellbau ist ein Wertebereich von 1000µs-2000µs Pulsweite für die Ansteuerung von Servo-Motoren und Drehzahlstellern etabliert. Dieser Wertebereich ist für das Mischen von Daten aus mehreren Quellen nicht ideal. Um die Kompatibilität zu etablierten Komponenten zu wahren wurde sich dennoch für eine konsequente Verwendung dieses Wertebereiches entschieden. Die einzelnen Software-Komponenten tauschen die Kanaldaten über global definierte Structs aus. Diese bietet Speicherplatz für bis zu 16 Kanäle sowie einen Zeitstempel.

Listing 5.1: RC.h

```
1 #ifndef RC_h
2 #define RC_h
3 #include <stdint.h>
4
5 struct rcSource
6 {
7     static const uint8_t size = 16;
8     uint16_t data[size];
9     unsigned long timestamp;
10 };
11 #endif
```

5.1.2 Realisierung des Mischerverhaltens

Gehen wir nun davon aus, dass wir Daten aus unterschiedlichen Quellen in Form des soeben vorgestellten Structs vorliegen haben. Diese sollen nun in einer Anwendungs-spezifischen Art und Weise gemischt werden. Die im Folgenden abgebildete Implementierung eines Mischers soll als Grundlage für ähnlich geartete Projekte dienen. Sie stellt die Minimalaufwand-Variante dar, sowohl im Sinne einer möglichst geringen Prozessorlaufzeit als auch im Sinne einer möglichst simplen Umsetzung und einfachen Erweiterbarkeit.

Listing 5.2: Mixer.h

```
1 #include "Arduino.h"
2 #include "RC.h"
3
4 class Mixer
5 {
6     public:
7         Mixer(rcSource *out, rcSource *in0, rcSource *in1);
8         void mix();
9     private:
10         rcSource *_out, *_in0, *_in1;
11 };
```

Wie in Mixer.h zu erkennen, liegt die einfachste Form eines Mischers vor. Daten aus lediglich zwei Quellen werden eingelesen und zu einem einzigen ausgehenden Datenstrom verarbeitet. Der Datenaustausch findet gemäß dem vorangegangenen Abschnitt in Form eines Structs statt.

Listing 5.3: Mixer::mix()

```
1 void Mixer::mix()
2 {
3     // direct passthrough from the radio control by default
4     {
5         for(int i=0 ; i<_out->size && i<_in0->size ; i++){
6             _out->data[i]=_in0->data[i];
7         }
8     }
9     // enable mixing if Spektrum channel 5 > 1100
10    // overwrites relevant channels with mixed channel data
11    if (_in0->data[5-1]>1100)
12    {
13        // calculate the factor by which in1 is mixed to in0
14        float weight = ((float)(_in0->data[5-1]-1000))/1000;
15        // aileron (roll) and elevon (pitch)
16        _out->data[1]=_in0->data[1]-((int)_in1->data[1]-1536)*
17        weight;
18        _out->data[2]=_in0->data[2]-((int)_in1->data[0]-1536)*
19        weight;
20    }
21    _out->timestamp = micros();
22 }
```

Die Verarbeitung bzw. Vermischung der eingehenden Daten findet in der Funktion `mix()` statt. Sicherheit im Sinne von Wahrung der manuellen Kontrolle über das System zu jedem Zeitpunkt ist hier das Hauptaugenmerk. So befindet sich der Mischer zunächst in einem inaktiven Zustand, in welchem er lediglich die an `in0` vorliegenden Daten in `out` kopiert. `in0` stellt hierbei die von der Fernsteuerung empfangenen Daten dar. Erst wenn ein frei definierbarer Kanal der `in0` Quelle, also der Fernsteuerung mit welcher das Fluggerät manuell geflogen wird, einen ebenso frei definierbaren Wert überschreitet, werden die Daten aus der Sekundär-Quelle `in1` mit den Daten aus der Primär-Quelle `in0` gemischt. In der oben dargestellten Implementierung ist die Gewichtung der Sekundärquelle über den Wert ebendieses frei definierbaren Kanales in einem Bereich von 10% – 100% einstellbar. Die untere Grenze von 10% ergibt sich hier aus der Verwendung eines einzelnen Kanales sowohl für die Aktivierung des Mischers als auch für die Wahl der Gewichtung. So ist der Mischer hier erst ab einem Wert von 1100 aktiv, dies entspricht 10% Gewichtung. Andere Konfigurationen sind selbstverständlich möglich, sollte dies die konkrete Anwendung erfordern.

5.2 Vernetzung der Systemkomponenten

Wie in Abbildung 5.1 zu erkennen, ist im Rahmen dieses Projektes die Kommunikation zwischen drei wesentlichen Komponenten herzustellen und aufrecht zu erhalten. Im konventionellen Betrieb eines Flugmodells oder Quadropters besteht zwischen Flugsteuerung und Fernsteuerungs-Empfänger eine direkte Verbindung. Durch Einbringung des Mobiltelefons und den von diesem im Rahmen des VBLS-Projektes generierten Daten ist nun eine weitere Komponenten einzubinden. Nun sollen die aus den beiden Quellen erhaltenen Daten zunächst wie bereits beschrieben gemischt werden. Bevor dies geschehen kann, müssen sie jedoch zunächst einmal in der unter Abschnitt 4.3.1 beschriebenen Form vorliegen. Im folgenden sollen zunächst die Anbindung an die beiden verwendeten Quellen, und im Anschluss die Anbindung an die verwendete Flugsteuerung erläutert werden.

5.2.1 Anbindung an die Android-Applikation

Der Datenaustausch mit der Android-Applikation erfolgt unidirektional, in Form eines simplen Byte-Streams über eine serielle Schnittstelle. Diese serielle Schnittstelle ist im Falle des Arduino Leonardo bzw. des Atmega 32u4 als virtueller CDC-USB-ComPort realisiert. Auf eine bi-direktionale Kommunikation wurde verzichtet, da ein erneutes Anfordern potentiell ungültiger oder falsch übertragender Daten allenfalls zu einer Neuübertragung bereits veralteter Daten führen würde. Hier ist es sinnvoller, ggf. ungültige Daten zu verwerfen und auf den Empfang neuer, gültiger Daten zu warten. An dieser Stelle der Vergleich zur Sprachübertragung via VOIP oder einem Video-Stream via UDP. Die Datenrate beträgt lediglich 9600bps, kann jedoch erhöht werden, sollten größere Datenmengen oder kürzere Übertragungszeiten realisiert werden müssen.

Listing 5.4: RawSerial.h - Auszug

```

1 #define SERIAL_MAX_SUPPORTED_CHANNEL_COUNT 2
2 #define SERIAL_FRAME_SIZE 2

4 class RawSerial
5 {
6     public:
7         RawSerial(rcSource *source);
8         void init(Serial_ *raw_Serial);
9         void init(HardwareSerial *raw_Serial);
10        void dataReceive();
11        uint16_t readRawRC(uint8_t chan);
12        void getData();

14    private:
15        Serial_ *_Serial;
16        rcSource *_source;
17        uint8_t channelCount;
18        uint16_t rxRefreshRate;

20        uint8_t frame[SERIAL_FRAME_SIZE];

22        uint16_t channelData[SERIAL_MAX_SUPPORTED_CHANNEL_COUNT];
23    };

```

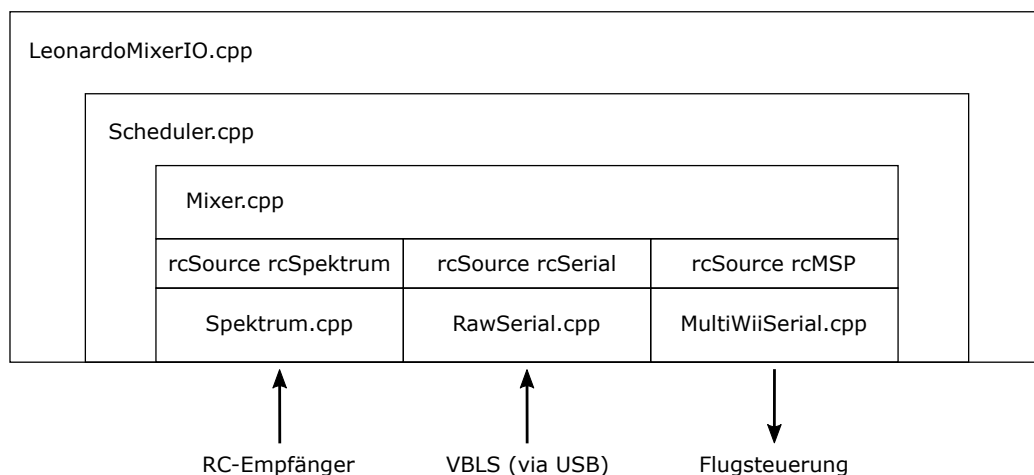


Abbildung 5.1: Blockdiagramm des Systems

Die Klasse `RawSerial` ist bei Instanziierung mit einem Zeiger auf ein `rcSource` Objekt im Konstruktor zu versehen. Dieses dient dem Datenaustausch mit Funktionen anderer Klassen. Die Methode `init(Serial_ *raw_Serial)` bzw. `init(HardwareSerial *raw_Serial)` dient der Übergabe und Initialisierung der zu verwendenden seriellen Schnittstelle. Hier ist derzeit je nach verwendeter Art der seriellen Schnittstelle eine manuelle Anpassung im Programmcode der Klasse notwendig, da im Falle des Arduino Leonards zwischen der virtuellen seriellen Schnittstelle (resp. `Serial_`) und der in Hardware implementierten seriellen Schnittstelle (resp. `HardwareSerial`) unterschieden wird. Die Methode `dataReceive()` liest die Daten von der seriellen Schnittstelle und speichert sie klassenintern zwischen. Die Methode `getData()` formatiert diese Daten und kopiert sie in das der Klasse übergebene `rcSource` Objekt.

Die Android-Applikation sendet zwei Datenkanäle, konkret die Position des erkannten Kreises in der x - y -Ebene. x und y sind hierbei positive Ganzzahlen im Bereich von 0–255, mit 128 gleichbedeutend der Mitte des Erfassungsbereiches. Eine Übertragung der Werte als `uint8` in Form eines einzelnen Bytes pro Kanal ist folglich naheliegend und wurde auch dementsprechend implementiert.

Listing 5.5: `RawSerial::getData()`

```
1 void RawSerial::getData()
2 {
3     uint16_t temp = 1500;
4     for (uint8_t i = 0; i < _source->size; i++)
5     {
6         if (i < SERIAL_MAX_SUPPORTED_CHANNEL_COUNT)
7         { // data has a range of 0-255,
8           // scaling it to 0-1024 and norming it to 1024-2048
9             temp = ((frame[i])*4+1024);
10            if (temp < 1024)
11                _source->data[i] = 1024;
12            else if (temp > 2048)
13                _source->data[i] = 2048;
14            else
15                _source->data[i] = temp;
16        }
17        else
18            // safe (stick center) value in radio control
19            _source->data[i] = 1500;
20    }
21    _source->timestamp = micros();
22 }
```

Die abgebildete Funktion `getData()` skaliert den Wertebereich der empfangenden Daten um den Faktor 4 auf 0-1024 und verschiebt diesen darüber hinaus um 1024, so dass sich ein resultierender Wertebereich von 1024-2048 ergibt, welcher dem angestrebten Wertebereich von 1000-2000 hinreichend entspricht. Werte ausserhalb des erlaubten Bereiches werden entsprechend begrenzt.

5.2.2 Anbindung an die RC-Fernsteuerung

Auch wenn wir großes Vertrauen in die Fähigkeiten der entwickelten Android-Applikation haben, so wäre es doch grob fahrlässig und darüber hinaus kaum praktikabel, dieser die alleinige Kontrolle über das Fluggerät zu überlassen. Wenigstens die manuelle Kontrolle über die mittlere Rotorgeschwindigkeit und somit auch die Flughöhe sollte zu jedem Zeitpunkt erhalten bleiben, auch wenn zukünftige Fortentwicklungen der Applikation eine Regelung der Flughöhe erlauben mögen. Somit ist es nötig, Daten von einer handelsüblichen Funkfernsteuerung zu empfangen und dem Mischer zur Verfügung zu stellen.

Die Wahl des verwendeten Funkempfängers fiel aus mehreren Gründen auf einen kompakten und leichten Spektrum-kompatiblen Empfänger, einen sog. Satelliten-Empfänger. Ausschlaggebend für diese Wahl war die Bereitstellung einer Implementierungs-Anleitung von Seiten Spektrums bzw. Horizon Hobbys (vgl. [Hor16]). Darüber hinaus findet das Spektrum DSMX Protokoll im europäischen und amerikanischen Raum große Verbreitung und es stehen zahlreiche kompatible und preisgünstige Fernsteuerungen zur Verfügung, welche dieses Protokoll unterstützen. Dies unterstützt das Ziel dieser Arbeit, eine möglichst flexible und einfach zu reproduzierende Grundlage für eigene Experimente zu bieten. Die Klasse **Spektrum** ist in ihrem Aufbau grundlegend ähnlich zu der im vorangegangenen Abschnitt vorgestellten Klasse **RawSerial**. Dies ermöglicht eine weitgehend identische Handhabung der Klassen innerhalb eines Projektes. Lediglich die `init(parameter)` Methode unterscheidet sich in der Übergabe ihrer Parameter.

Listing 5.6: Spektrum.h - Auszug

```

1 #define SPEKTRUM_MAX_SUPPORTED_CHANNEL_COUNT 12
2 // [...]
3 class Spektrum
4 {
5     public:
6         Spektrum(rcSource *source);
7         void init(HardwareSerial *spek_Serial,
8                 uint8_t serialrx_provider);
9         void dataReceive();
10        uint16_t readRawRC(uint8_t chan);
11        void getData();
12
13    private:
14        HardwareSerial *_Serial;
15        rcSource *_source;
16        uint8_t serialrx_provider;
17
18        // [...]
19
20        uint8_t frame[SPEKTRUM_FRAME_SIZE];
21
22        uint16_t channelData[SPEKTRUM_MAX_SUPPORTED_CHANNEL_COUNT];
23 };

```

Der Spektrum-Remote-Receiver überträgt die empfangenen Daten ebenfalls uni-direktional mittels einer einfachen seriellen Schnittstelle gemäß der untenstehenden Spezifikation:

In normal operation, the Spektrum remotes issue a 16-byte data packet every 11ms or 22ms, depending upon the selected protocol. The packet is transmitted at 125000bps, 8 bits, No parity, 1 stop (8N1). For those UARTs which are not capable of that speed, they will also work at the more-standard 115200bps. (vgl. [Hor16, S. 2])

Die Formatierung dieser empfangenen Daten unterscheidet sich grundlegend von dem simplen Protokoll, welches im vorangegangenen Abschnitt vorgestellt wurde. Sie ist ausführlich in dem von Horizon Hobby, LLC herausgegebenen Dokument beschrieben:

[...][T]he first two bytes (fieldname "fades" in Section 8.4 below) indicate the fade count, and the remaining 14 bytes are the data packet.[...] The 14-byte data packet is either 1024 or 2048 servo data. DSM2/22 is the only 1024 packet; all others use 2048. All data fields are big-endian, that is, the MSB is transmitted before the LSB. Bit 0 is the lsb, bit 15 is the msb. (vgl. [Hor16, S. 3,4])

Listing 5.7: Spektrum::dataReceive()

```
1 void Spektrum::dataReceive()
2 {
3     if (_Serial->available() >= SPEKTRUM_FRAME_SIZE)
4         _Serial->readBytes(frame, SPEKTRUM_FRAME_SIZE);
5
6     for (uint8_t b = 3; b < SPEKTRUM_FRAME_SIZE; b += 2)
7     {
8         uint8_t spekChannel = (frame[b - 1] & chanMask) >>
9             bitShift;
10
11         if (spekChannel < channelCount && spekChannel <
12             SPEKTRUM_MAX_SUPPORTED_CHANNEL_COUNT)
13         {
14             channelData[spekChannel] = ((uint16_t)(frame[b - 1]
15                 & dataMask) << 8) + frame[b];
16         }
17     }
18 }
```

Die obenstehende Methode `dataReceive()` liest zunächst die vom Empfänger geschriebenen Daten aus dem Puffer der seriellen Schnittstelle. Anschließend werden diese nach folgendem Schema ausgelesen und klassenintern in `channelData[]` gespeichert:

Servo Field 2048 Mode

This format is used by all protocols except DSM2/22ms mode.

Bits 15 Servo Phase

Bits 14-11 Channel ID

Bits 10-0 Servo Position

(vgl. [Hor16, S. 4])

Analog zu der bereits vorgestellten Klasse `RawSerial` bietet auch `Spektrum` eine `getData` Methode, welche analog ihrem bereits vorgestellten Pendant die lokal in der Klasse gespeicherten Kanaldaten auf den Wertebereich 1024-2048 normiert und in das der Klasse übergebene `rcSource`-Objekt schreibt.

5.2.3 Anbindung an den Flugcontroller

Gehen wir nun davon aus, dass wir Daten aus mindestens zwei Datenquellen empfangen und gemischt haben, so müssen diese gemischten Daten nun einer externen Datensenke, in diesem Projekt einem kommerziell erhältlichen Flugsteuerung der Naze32-Familie, zur Verfügung gestellt werden. Die auf diesem und anderen Flugsteuerungen verwendete Software Cleanflight¹ basiert in ihren Grundzügen auf dem MultiWii-Projekt². Dies ist insofern relevant, da sie auch das in den Grundlagen beschriebene MultiWiiSerial-Protokoll (kurz MSP) spricht, welches auch eine digitale Übertragung von Kanaldaten ermöglicht. Hierzu ist lediglich eine Zweidraht-Leitung zwischen dem Arduino Leonardo und dem Flugsteuerung nötig. Aufgrund dieser einfachen Verdrahtung und der standardmäßigen Kompatibilität mit Software wie MultiWii und Cleanflight stellt sich das MSP als ideales Protokoll zum Datenaustausch zwischen Mischer und Flugsteuerung dar.

Da es zu dem MSP keine zusammenhängende Dokumentation gibt und der entsprechende Eintrag in dem MultiWii-Wiki (vgl. [Mul15]) lediglich eine Übersicht über die verfügbaren Datenfelder bietet, wurde an dieser Stelle auf die Umsetzung des MSP in Cleanflight zurückgegriffen, um sich anhand dieser ein Verständnis der Funktionsweise der Datenübertragung mittels MSP zu erarbeiten.

¹Cleanflight - <http://cleanflight.com/>

²MultiWii - <http://www.multiwii.com/>

Listing 5.8: MultiWiiSerial.h - Auszug

```

1 #define MSP_MAX_SUPPORTED_CHANNEL_COUNT 16
2 #define MSP_FRAME_SIZE 16*2 // two bytes per channel

4 class MultiWiiSerial
5 {
6     public:
7         MultiWiiSerial(rcSource *source);
8         void init(HardwareSerial *msp_Serial);
9         void SerialEncode();

11        private:
12            HardwareSerial *_Serial;
13            rcSource *_source;
14            uint8_t channelCount;
15            uint16_t rxRefreshRate;

17            uint8_t frame[MSP_FRAME_SIZE];

19            uint16_t channelData[MSP_MAX_SUPPORTED_CHANNEL_COUNT];
20 };

```

Die Klasse MultiWiiSerial wird analog zu den bereits vorgestellten Empfangs-Klassen instanziiert. Der `void init(HardwareSerial *msp_Serial)` Methode wird die Referenz auf eine serielle Schnittstelle übergeben. Die Methode `SerialEncode()` liest die Kanaldaten aus dem referenzierten rcSource-Objekt und schreibt sie als MSP-Pakete formatiert auf die serielle Schnittstelle. Der Aufbau eines solchen MSP-Paketes gestaltet sich wie folgt:

The general format of an MSP message is:
<preamble>,<direction>,<size>,<command>,<crc>

Where:

- preamble = the ASCII characters '\$M'
- direction = the ASCII character '<' if to the MWC or '>' if from the MWC
- size = number of data bytes, binary.
Can be zero as in the case of a data request to the MWC
- command = message_id as per the table below
- data = as per the table below. UINT16 values are LSB first.
- crc = XOR of <size>, <command> and each data byte into a zero'ed sum

[Mul15]

Die Implementierung dieses Protokolls gestaltet sich als verhältnismäßig einfach; die folgende Methode `SerialEncode()` ist speziell und ausschließlich für das Bilden und Senden sogenannter „MSP_SET_RAW_RC“-Pakete (message_id = 200, vgl. [Mul15]) geschrieben.

Listing 5.9: MultiWiiSerial::SerialEncode()

```

1 void MultiWiiSerial::SerialEncode()
2 {
3     uint8_t crc = 0; // checksum
4     uint8_t size = channelCount*2; // size in bytes
5     uint8_t opcode = 200; // opcode 200 is for Raw RC
6     // Data
7     uint8_t header[] = {'$', 'M', '<', size, opcode};

9     _Serial->write(header, 5);

11    // First relevant data for the checksum is the size-byte.

```

```
12     crc ^= size;
13     crc ^= opcode;

15     for(uint8_t i = 0; i < channelCount; i++)
16     {
17         _Serial->write(uint8_t(_source->data[i]%256)); //low byte
18         // first
19         _Serial->write(uint8_t(_source->data[i]/256)); //high byte
20         // last

22         crc ^= uint8_t(_source->data[i]%256);
23         crc ^= uint8_t(_source->data[i]/256);
24     }
25     _Serial->write(crc);
26 }
```

6 TESTS

6.1 Parametrierung des Mischers

Zur Beschreibung der Parametrisierung des Mischers ist eine Abgrenzung zwischen dem Gesamtsystem Mischer – LeonardoMixerIO – und den Komponenten dieses, zum einen dem Scheduler und zum anderen dem eigentlichen Mischer, nötig.

Letzterer ist statisch in Programmcode implementiert und kann in seinem Funktionsumfang lediglich durch Änderungen an diesem angepasst werden. Eine Parametrisierung im Sinne des Mischerverhaltens, genauer der Gewichtung der vom Mobiltelefon empfangenen Kanaldaten, kann zur Laufzeit und aus der Ferne über einen freien Proportional-Kanal der verwendeten Fernsteuerung erfolgen. Dies erwies sich während der Testflüge als ausgesprochen hilfreich.

Die Parametrisierung des Schedulers erfolgt innerhalb der `setup()` Methode der LeonardoMixerIO.cpp:

Listing 6.1: Parametrisierung des Schedulers

```
1  nRtTask42.priority = 3;
2  nRtTask42.activity = doSomething_Function;
3  nRtTask42.timestamp = 0;

5  rtTask0.activity = doSomethingReallyImportant_Function;
6  rtTask0.timestamp = 0;
7  rtTask0.cycleTime = 20000;

9  scheduler = new Scheduler(rtTasks, nRtTasks);
10 //scheduler->setDebugger(&Serial);
```

Die hierbei hervorzuhebenden Parameter sind die `cycleTime` der echtzeitkritischen Tasks sowie die `priority` der nicht-echtzeitkritischen Tasks. Diese wurden für den Flugbetrieb wie folgt gewählt:

```
1  readRawSerial.priority = 1;
2  readSpektrum.priority = 1;
3  mix.priority = 2;
4  printLCD.priority = 3;

6  send.cycleTime = 20000;
```

Die Priorisierung der nicht-echtzeitkritischen Tasks gestaltet sich wie folgt: Tasks welche Daten empfangen und das lokale Daten-Abbild aktualisieren besitzen die höchste Priorität. Es folgt der eigentlichen Mischer, welcher die durch diese Tasks empfangenen Daten verarbeitet. Niedrigste Priorität besitzt die LCD-Ausgabe, welche nur im stationären Laborbetrieb im Rahmen des Debugging genutzt wird. Das Senden der Daten an die Flugsteuerung ist echtzeit-kritisch und erfolgt zyklisch alle 20ms.

6.2 Testbedingungen

Getestet wurden sowohl die Applikation im Ganzen, also der LeonardoMixer, wie auch der Scheduler für sich alleine. Bei diesem wurde zum einen die Ausführungszeit (der sog. „Runtime-Over-head“) und zum anderen die Stabilität untersucht. Bzgl. der Ausführungszeit wurde wiederum zwischen Echtzeit- und Nicht-Echtzeit-Taskverwaltung unterschieden.

Im ersten Test wurde dazu die Funktion `schedule()` derart modifiziert, dass am Ende der Ausführung die Differenz zwischen der aktuellen Laufzeit und der Laufzeit zu Beginn der Funktion ausgegeben wird. Als Test-Tasks wurden lediglich zwei leere Echtzeit-Tasks mit unterschiedlichen Zykluszeiten angelegt. So wird zum einen gewährleistet, dass die Funktion `sortRtTasks(...)` zum Tragen kommt. Zum anderen wird `perform()` nicht aufgerufen, da die Betrachtung der Nicht-Echtzeit-Tasks einzeln erfolgen soll.

Die Messung des „Runtime-Overheads“ der Nicht-Echtzeit-Taskverwaltung gestaltete sich insofern einfacher, dass `perform()` nicht extra modifiziert werden musste. Es war ausreichend, lediglich einen Nicht-Echtzeit-Task anzulegen, der bei jeder Ausführung die Differenz zwischen dem Zeitpunkt seiner letzten Ausführung (d. h. `timestamp`) und der aktuellen Laufzeit ausgibt.

Die Stabilität des Gesamtsystem LeonardoMixer wurde in Form eines Dauertests auf der im Rahmen des Parallelprojekts „VBLs“¹ genutzten Flugplattform, einem Quadrokopter in X-Anordnung, getestet.

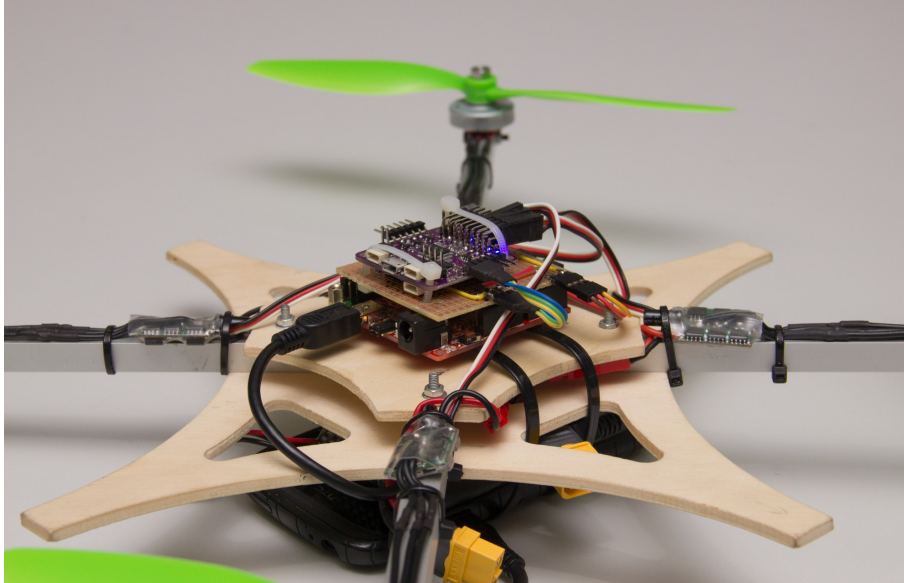


Abbildung 6.1: Nahansicht des Versuchsaufbaus im Profil

In Abbildung 6.1 gut zu erkennen ist der schichtweise Aufbau des Testaufbaus:

- Naze32-kompatible Flugsteuerung.
- Arduino-Leonardo-kompatibler Olimexino 32u4 mit eigens angefertigter Adapterplatine, um die Verdrahtung der Komponenten entsprechend den Anforderungen des vibrationsträchtigen Aufbaus sicher zu gestalten.
- Der ursprünglich von Weber gebaute Quadrokopter, mit neu ausgeführter Verdrahtung und Ergänzungen [vgl. Web16].
- Akku
- Mobiltelefon mit VBLs-Applikation

¹VBLs - <https://gitlab.cvh-server.de/lf.ps/vbls/tree/master/Visual-Based-Landing-System>

6.3 Ergebnisse

Nach mehreren Messungen des „Runtime-Overheads“ des Schedulers konnten folgende Ausführungszeiten auf dem verwendeten Atmega 32u4 ermittelt werden:

Echtzeit-Taskverwaltung:	85µs
Nicht-Echtzeit-Taskverwaltung:	20µs

Dies bedeutet, dass bei Verwendung dieses Schedulers mit zwei Echtzeit-Tasks eine Abweichung von der vorgegeben Zykluszeit von unter 85µs erreicht werden kann. Für die gegebene Anwendung mit einer Ziel-Zykluszeit von 20ms entspricht dies einer Abweichung von 0,425%.

Anmerkung: Es ist zu beachten, dass lediglich der statische Anteil des „Runtime-Overheads“ für zwei Echtzeit-Tasks gemessen wurde. Der dynamische Anteil, d. h. die für die Sortierung benötigte Zeit, steigt mit zunehmender Anzahl der jeweiligen Tasks um den Faktor n (für Echtzeit-Tasks) bzw. $n \cdot \log n$ (für Nicht-Echtzeit-Tasks). Die Ausführungszeit im Fall von Starvation erhöht sich durch das Ausführen von `setPriorities()` ebenfalls.

Bzgl. der Stabilität des Gesamt-Systems konnte nach einem Dauertest über den Zeitraum von 45 Minuten kein Verklemmen oder anderweitig unerwünschtes Verhalten festgestellt werden.

Im Verlaufe der abschließenden Flugversuche ist kein einziger Ausfall zu vermerken, welcher auf das in diesem Projekt geschaffene System zurückzuführen wäre. Unterschiede im Ansprechverhalten des Systems im Betrieb mit und ohne Mischer konnten nicht festgestellt werden. Dies bedeutet, dass der Mischer die gesetzten Anforderungen bezüglich der Echtzeitfähigkeit und der Stabilität der Anwendung gänzlich erfüllt. Dieser Erfolg ist insbesondere dem Scheduler sowie der konsequenten Auslegung der einzelnen Methoden auf kooperatives Multitasking zuzuschreiben.

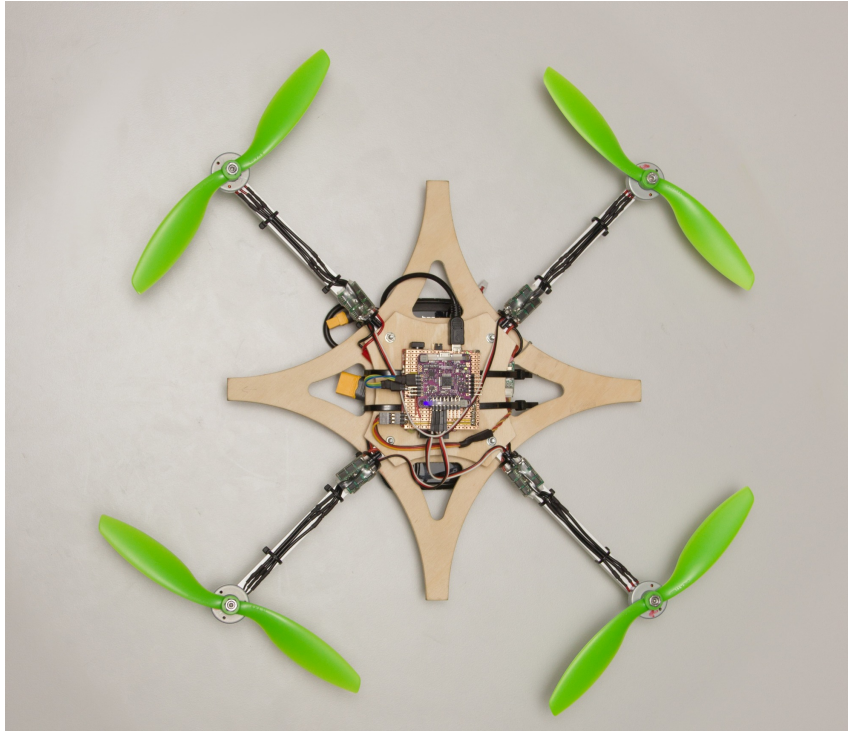


Abbildung 6.2: Draufsicht auf die Oberseite des Versuchsaufbaus. Die Hauptdiagonale zwischen zwei Rotoren beträgt 415mm.



Abbildung 6.3: Draufsicht auf die Unterseite des Versuchsaufbaus

7 ZUSAMMENFASSUNG

Abschließend lässt sich festhalten, dass alle Projektziele und -anforderungen erfüllt werden konnten. Nicht nur konnte eine Vernetzung der Einzelkomponenten RC-Empfänger, Mobiltelefon und Flugsteuerung hergestellt und im Testbetrieb aufrecht erhalten werden, diese konnte sich auch in mehreren Testflügen über einen Zeitraum von insgesamt mehr als 60 Minuten bewähren. Weiterhin weist der im Rahmen des Projekts erstellte Scheduler mit ca. 0.1ms eine für den gegebenen Anwendungsfall mehr als ausreichende Echtzeitfähigkeit auf und stellt eine gute Grundlage für weitere echtzeitkritische Projekte auf Arduino-Basis dar.

8 AUSBLICK

Wie in der Zusammenfassung bereits verdeutlicht, konnten alle Projektziele erreicht werden. Der modulare Aufbau dieses Projektes bzw. des diesem zugrunde liegenden Programms lädt jedoch zu Erweiterungen und Modifikationen ein:

- Implementierung eines präventiven (im Gegensatz zum vorhandenen reaktiven) Ansatz der Taskverwaltung, d. h. lediglich Ausführen eines Nicht-Echtzeit-Tasks, wenn dessen durchschnittlich benötigte Ausführungszeit geringer ist, als die Differenz zwischen aktuellem Zeitpunkt und dem Ausführungszeitpunkt des nächsten Echtzeit-Tasks. Somit könnte künstlich nahezu harte Echtzeit geschaffen werden
- Erweiterung um weitere Protokolle
- Erweiterung des eigentlichen Mischer-Codes, beispielsweise um eine Möglichkeit der Parametrisierung zur Laufzeit, etwa mittels eines Kommandozeileninterfaces. Denkbar wäre hier auch eine Konfiguration des Mischers über eine Android-Applikation, ganz im Sinne des Schwesterprojektes VBLS
- Portierung auf andere Plattformen, beispielsweise den ESP8266. Dieses SoC – System on Chip – stellt neben mehr Prozessor- und Speicherkapazitäten noch WLAN zur Verfügung, worüber eine GUI zur Konfiguration des Mischers oder Telemetriedaten bereitgestellt werden können.
- Integration in andere Plattformen, beispielsweise MultiWii oder Cleanflight.

9 LITERATUR

- [Atm16] Atmel Corporation. ATmega16U4/32U4 [DATASHEET SUMMARY]. Apr. 2016. URL: http://www.atmel.com/Images/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Summary.pdf (besucht am 10. 01. 2017).
- [Bau81] Rüdiger Baumann. „Informatik mit Pascal“. In: 1.Edition. Stuttgart, Baden-Württemberg: Ernst Klett Verlag, 1981. Kap. Komplexität.
- [Hor16] Horizon Hobby, LLC. Specification for Spektrum Remote Receiver Interfacing. Apr. 2016. URL: <https://www.spektrumrc.com/ProdInfo/Files/Remote%20Receiver%20Interfacing%20Rev%20A.pdf> (besucht am 02. 02. 2017).
- [Mul12] MultiWii Forum. New Multiwii Serial Protocol. Apr. 2012. URL: <http://www.multiwii.com/forum/viewtopic.php?f=8&t=1516> (besucht am 03. 10. 2016).
- [Mul13] Multiplex Modellsport GmbH & Co. KG. SRXL-MULTIPLEX V2. Feb. 2013. URL: <https://www.multiplex-rc.de/Downloads/Multiplex/Schnittstellenbeschreibungen/srxl-multiplex-v2.pdf> (besucht am 03. 10. 2016).
- [Mul15] MultiWii Wiki. Multiwii Serial Protocol. Jan. 2015. URL: http://www.multiwii.com/wiki/index.php?title=Multiwii_Serial_Protocol (besucht am 03. 10. 2016).
- [Pla17] PlatformIO. PlatformIO is an open source ecosystem for IoT development. 2017. URL: <http://docs.platformio.org/en/latest/> (besucht am 05. 02. 2017).
- [Pre+92] William H. Press u. a. „Numerical Recipes in C“. In: 2.Edition. Cambridge, UK: Press Syndicate of the University of Cambridge, 1992. Kap. Heapsort.
- [Sed92] Robert Sedgewick. „Algorithmen in C++“. In: 1.Edition. Bonn, Nordrhein-Westfalen: Addison Wesley GmbH, 1992. Kap. Prioritätswarteschlangen.
- [The01] The Model Electronics Company. How It Works – The PPM Radio Control System: Part 2. Jan. 2001. URL: <http://www.e-radiocontrol.com.ar/downloads/mectn004.pdf> (besucht am 03. 10. 2016).
- [Web16] Jan Weber. „Gespräch mit Herrn Weber“. In: (Sep. 2016).
- [Wika] Wikipedia. Wikipedia-Eintrag zur Flugdynamik von Quadrocoptern. URL: https://en.wikipedia.org/wiki/Quadcopter#Flight_dynamics (besucht am 07. 03. 2017).
- [Wikb] Wikipedia. Wikipedia-Eintrag zur Funkfernsteuerung. URL: <https://de.wikipedia.org/wiki/Funkfernsteuerung> (besucht am 07. 03. 2017).

10 ANHANG

10.1 Persönliches Fazit

Ergänzend zu dem vorangegangenen inhaltlichen Fazit möchten wir zum Ende dieser Dokumentation noch kurz unser persönliches Fazit ziehen.

Die Entwicklung dieses Systems hat uns als Autoren der hierfür nötigen Software und dieser Dokumentation einiges an Zeit gekostet. Zusammenfassend lässt sich jedoch sagen, dass diese Zeit ausgesprochen sinnvoll investiert ist und die gewonnenen Erkenntnisse und Erfahrungen den nötigen Aufwand mehr als rechtfertigen. Auch konnten wir als Team von einem regen Wissens- und Erfahrungsaustausch untereinander profitieren. So konnte jeder von uns sein eigenes Wissen und seine persönlichen Fähigkeiten voll in dieses Projekt einbringen. Dies freut uns sehr.

10.2 Lizenzen

Der im Rahmen dieses Projektes geschaffene Programmcode steht ebenso wie diese Dokumentation unter der Modifizierten BSD Lizenz (<https://gitlab.cvh-server.de/lf.ps/vb1s/blob/master/common/BSD-MODIFIED.txt>).

Eine Weiterverbreitung und Weiterverwendung ist ausdrücklich erwünscht. Die Software ist in der Hoffnung entstanden, nützlich zu sein und wird „wie sie ist“ zur Verfügung gestellt. Es kann keine Gewährleistung auf Fehlerfreiheit oder die Eignung für einen bestimmten Zweck übernommen werden. Darüber hinaus kann keinerlei Haftung für direkt oder indirekt aus der Nutzung dieser Software hervorgegangene Personen- oder Sachschäden übernommen werden.

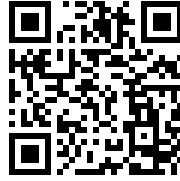
Jeder ist für sein eigenes Handeln verantwortlich. Wir raten zu einem sicheren und besonnenen Umgang mit Fluggerät jedweder Art.

Auflistung verwendeter Software und deren Lizenzen (ohne Anspruch auf Vollständigkeit):

- Atom
MIT License
<https://github.com/atom/atom/blob/master/LICENSE.md>
- PlatformIO
Apache License Version 2.0
<https://github.com/platformio/platformio-docs/blob/develop/LICENSE>
- Arduino
GNU General Public License Version 2
<https://github.com/arduino/Arduino/blob/master/license.txt>
- avr-gcc
GNU General Public License Version 2
<https://gcc.gnu.org/wiki/avr-gcc>
- AVRDUDE
GNU General Public License Version 2
<http://savannah.nongnu.org/projects/avrdude>
- L^AT_EX
LaTeX project public license
<https://www.latex-project.org/lppl.txt>
- GIMP
GNU General Public License Version 2
<https://www.gimp.org/about/COPYING>
- IrfanView
Freeware
<http://www.irfanview.com/eula.htm>
- Adobe Photoshop Lightroom
Proprietär & Kommerziell
<http://labs.adobe.com/technologies/eula/lightroom.html>

10.3 gitlab-Repository

Der im Rahmen dieses Projektes entstandene Quellcode sowie die hier vorliegende Dokumentation können über den gitlab-Server des Campus Velbert-Heiligenhaus der Hochschule Bochum bezogen werden.



<https://gitlab.cvh-server.de/lf.ps/vbls>

10.4 Eidesstattliche Erklärung

Die Autoren versichern durch ihre Unterschriften, dass diese Arbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel verwendet wurden. Alle Zitate und Übernahmen sind im Text der Hausarbeit kenntlich gemacht.

Lukas Friedrichsen
Solingen, den 20. März 2017

Philipp Stenkamp
Langenfeld, den 20. März 2017