Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Bachelorarbeit

Entwicklung einer WLAN Schnittstelle mit Google Protobuf für humanoide Roboter

Hauke Mönck

Gutachter: Prof. Dr. Raúl Rojas
Betreuer: Dr. Hamid Reza Moballegh

28. April 2014

Zusammenfassung

Damit ein autonomer Roboter mit seiner Umwelt kommunizieren kann, benötigt er ein entsprechendes Kommunikationsinterface. Der Anspruch dieser Kommunikation liegt in der Auswertung sensorischer Messdaten oder dem Austausch von Informationen mit einer Basisstation oder anderen Robotern. Im FUB-KIT Projekt geschieht diese Auswertung von Messdaten bislang kabelgebunden über eine serielle Schnittstelle. Das FUmanoids Team nutzt zu diesem Zweck eine drahtlose Kommunikation über WLAN.
Im Rahmen dieser Arbeit wird an diesen Entwicklungsstand angeknüpft, indem die Hardware des FUB-KIT Projektes mit einem seriell ansteuerbaren WLAN-Modul versehen und das Protokoll der FUmanoids adaptiert wird. So können bereits vorhandene Mechanismen zur Auswertung wiederverwendet und erweitert werden.
Da all dies im Rahmen eines eingebetteten Systems geschieht, wird ein besonderes Augenmerk auf Performanz und Redundanzfreiheit gelegt, wobei gleichermaßen die Robustheit des Protokolls und der Software zu beachten sind.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Berlin, den 28.04.2014

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1
1.2 Aufgaben und Zielsetzung . . . 1
1.3 Aufbau der Arbeit . . . 2
1.4 Definitionen . . . 2

2 Rahmenbedingungen **4**

2.1 Stand der Technik . . . 4
2.2 Regel der Kommunikation . . . 4

3 Komponenten **6**

3.1 Überblick über Hardwarekomponenten . . . 6
3.1.1 CM-530 . . . 6
3.1.2 Avisaro WLAN Module 1.0 . . . 7
3.1.3 GS1500M WLAN-Modul . . . 8
3.1.4 HaViMo2 . . . 9
3.2 Überblick über Softwarekomponenten . . . 10
3.2.1 'easy-functions' CM-530 Bibliothek . . . 10
3.2.2 Nanopb Protobuf Bibliothek . . . 10
3.2.3 FU-Remote . . . 11
3.2.4 Das FU-Remote Protokoll . . . 12

4 Inbetriebnahme **14**

4.1 Verschaltung der Hardware . . . 14
4.2 Installation der Firmware GS1500M . . . 15
4.3 Starten der Beispielsoftware . . . 15

5 Struktur der Software **17**

5.1 Das Frontend . . . 17
5.2 Kommunikation mit dem WLAN-Modul . . . 19
5.2.1 Problem der Nebenläufigkeit . . . 19
5.2.2 Empfangen von dem WLAN-Modul . . . 20
5.3 Versenden von Nachrichten . . . 22
5.4 Protobuf und Nanopb . . . 23
5.4.1 Das Protobuf Format . . . 23
5.4.2 Das Protobuf Format - Ein Beispiel . . . 24
5.4.3 Generierung von Protobuf Nachrichten . . . 26

5.4.4 Encodierung und Decodierung 26
5.5 Verarbeiten von eingehenden Nachrichten 28
5.6 Erstellen neuer Nachrichten 29
5.7 Erstellen neuer Debug-Optionen 29

6 Sichere Datenübertragung 31
6.1 Zweck des Protokolls 31
6.2 Beispiel eines Alternierenden-Bit Protokolls 32
6.3 Test des Beispielprotokolls 34

7 Footprint 36
7.1 Messverfahren und Testbedingungen 36
7.2 Analyse der Stacknutzung 37
7.3 Analyse der Rechenzeit 38
7.4 Analyse des Traffics 40

8 Ausblick und Fazit 42
8.1 Einschränkungen 42
8.2 Ausblick 43
8.3 Zusammenfassung 43

Abbildungsverzeichnis 45

Literaturverzeichnis 46

9 Abkürzungsverzeichnis 49

1 Einleitung

Dieses Kapitel soll den Leser auf den Inhalt der Arbeit einstimmen, mit den Rahmenbedingungen vertraut machen und ihn über Strukturierung und Zielsetzung der Arbeit informieren.

1.1 Motivation

Der RoboCup ist eine Wettbewerbsveranstaltung für Roboter, in dem Universitäten ihre Forschungsergebnisse in den Feldern der künstlichen Intelligenz und Motorik von Robotern vorstellen und vergleichen können. Je nach Forschungsschwerpunkt bietet der RoboCup unterschiedliche Ligen, wie beispielsweise das humanoide Fußball oder Rettungsroboter an.

Das Projekt FUB-KIT ist ein Projekt der FU Berlin in Kooperation mit dem Kyushu Institute of Technology (九州工業大学) und beschäftigt sich mit der Entwicklung humanoider Fußballroboter der 'TeenSize' Klasse. Beim humanoidem Fußball ist eine direkte Steuerung der Roboter nicht erlaubt. Das Team soll unabhängig agieren und nur untereinander kommunizieren können. Außerdem sollte es Nachrichten von einem Schiedsrichter beachten können. Der RoboCup bietet für humanoide Fußballroboter folgende Ligen an:

- KidSize, 30-60 cm groß
- TeenSize, 90-120 cm groß
- AdultSize, 130 cm und größer

Das FUmanoids Projekt beschäftigt sich mit der Entwicklung von Robotern der 'KidSize' Klasse. Zum Auslesen von Sensor- und Messdaten etc. wurde im Rahmen des Projektes eine Softwarelösung geschaffen, welche drahtlos mit den Robotern mit Hilfe des Google Protobuf Formats kommunizieren kann. Im Rahmen dieser Arbeit soll das FUB-KIT Projekt an diesen Entwicklungsstand anknüpfen, indem für die projekteigene Hard- und Software das Protokoll der FUmanoids implementiert und eine drahtlose Kommunikation realisiert wird. Die Einschränkungen durch die Hardware, wie etwa vorhandener Speicher, Prozessorgeschwindigkeit und Kommunikationsschnittstellen, stellen in diesem Zusammenhang eine große Herausforderung dar.

1.2 Aufgaben und Zielsetzung

Die Problemstellung ist, Sensor- und Messdaten des Microcontrollers mit Hilfe der FU-Remote Software des FUmanoids Projektes auswerten zu können. Dies bezieht sich auf vorverarbeitete Kamerabilder als Grafiken, Tabellenansichten, Liniendiagrammen, relativer Position und einfachem Text. Außerdem müssen dem Microcontroller sogenannte 'GameController' Nachrichten, welche ein eigenes, simples Protokoll verwenden, übermittelt werden können. Des Weiteren sollte die Implementierung über die Kommunikation mit FU-Remote hinaus

auch eine Kommunikation mehrerer Microcontroller untereinander ermöglichen.

Ziel dieser Arbeit ist es, ein Konzept zu entwickeln und umzusetzen, mit dem es einem Microcontroller möglich ist, unter Verwendung des Google Protobuf Formats drahtlos über TCP-IP Netzwerke zu kommunizieren. Diese Kommunikation soll über eine serielle 'Zigbee' Schnittstelle mit Hilfe eines geeignet WLAN-Moduls abgewickelt werden.

1.3 Aufbau der Arbeit

Das Kapitel 'Rahmenbedingungen' wird zunächst der Stand der Technik erläutert und die vom RoboCup gegebenen Regeln erörtert, welche für diese Arbeit relevant sind. Daraufhin wird in 'Komponenten' ein Überblick über alle verwendeten Hard- und Softwarekompenenten geben und außerdem erörtert, warum speziell diese Komponenten gewählt wurden. Das darauf folgende Kapitel 'Inbetriebnahme' soll eine Anleitung sein, wie die zuvor erwähnte Hardware und der Sourcecode zu einem lauffähigen Beispiel gebracht werden können. Daraufhin wird im Kapitel 'Struktur der Software' der Quellcode und seine Komponenten im Detail erörtert. Als Kernbestandteil dieser Arbeit wird dort auf alle funktionalen Details eingegangen. Dann wird im Kapitel 'Sichere Übertragung' ein Protokoll vorgestellt, mit wessen Hilfe sich Daten sicher übertragen lassen, wobei auch auf die Relevanz eines solchen Protokolls eingegangen wird. Daraufhin wird im Kapitel 'Footprint' der Footprint des erarbeiteten Sourcecodes im Detail analysiert. Zum Schluss wird im Kapitel 'Zusammenfassung' das Thema abgerundet und zusammengefasst, ein Ausblick gegeben, aber auch einige Einschränkungen des hier erarbeiteten Codes aufgezeigt.

1.4 Definitionen

Im Rahmen dieser Arbeit wird u.A. das Versenden von Nachrichten im Detail beschrieben. Damit es dabei nicht zu begrifflichen Verwirrungen kommt, folgen einige Definitionen. Diese dienen lediglich der Eindeutigkeit und Einheitlichkeit der Bezeichnung im Rahmen dieser Arbeit.

Begriffsdefinitionen

- Paket
  Ein Paket ist ein IP-Paket nach dem Standard RFC791.[35] Es ist somit unabhängig von Protokollen wie etwa TCP oder UDP.
- Nachricht
  Eine Nachricht ist ein Array, welches strukturierte Informationen im Sinne des FUmanoid-Protokolls enthält.[1] Dies schließt etwaige Header des FUmanoid-Protokolls

[1] Protokoll nach Quellcode 'FUmanoids Code Release 2012 (released: December 27, 2012) 2012.1'[8]

vor einer Protobuf Nachricht ein, schließt allerdings für das WLAN-Modul zum Versenden oder Empfangen verwendete Header aus.[2]

- Protobuf-Nachricht
  Eine Protobuf-Nachricht besteht ausschließlich aus encodierten Daten im Sinne des Google Protobuf Formats[10].
- FUmanoids-Header
  Der FUmanoids-Header ist genau der Satz an Informationen, welcher laut FUmanoid-Protokoll einer Protobuf-Nachricht im Rahmen der Kommunikation mit FU-Remote vorangestellt werden muss. Insofern nicht anders erwähnt, ist der bis 2014 verwendete und 12 Byte große Header gemeint.[3]
- WLAN-Header
  Ein WLAN-Header beinhaltet all diejenigen Informationen, welche dem WLAN-Modul zum Versenden[4] übergeben oder als Zusatzinformation beim Empfangen[5] erhalten werden. Erstere werden speziell als WLAN-Sendeheader und letzte als WLAN-Empfangsheader bezeichnet.
- WLAN-Nachricht
  Eine WLAN-Nachricht ist eine Nachricht inklusive vorangehendem WLAN-Header.

[2]Siehe Packetheader 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20] 6 'Appendix'
[3]Vgl. 'FUmanoids Code Release 2012 (released: December 27, 2012) 2012.1'[8]
[4]Siehe Dokumentation 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20] 3.4.1 'Bulk data Tx and Rx'
[5]Siehe Dokumentation 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20] 6 'Appendix'

2 Rahmenbedingungen

In diesem Kapitel soll dem Leser vermittelt werden, was der bisherige Stand der Technik ist und unter welchen Rahmenbedingungen diese Arbeit entstanden ist.

2.1 Stand der Technik

Das FUB-KIT Team benutzt bislang den CM-530 Microcontroller, um den Roboter vollständig zu steuern. Andere Teams des RoboCup verwenden meist mehrere eingebettete Systeme für jeweils unterschiedliche Zwecke, welche dann in der Regel über eine serielle Schnittstelle miteinander kommunizieren. Das FUmanoids-Team benutzt beispielsweise ein 'Raspberry PI' mit vollständigem Linux Betriebssystem als leistungsstärkere Recheneinheit. Dadurch können leicht für Desktop-PC's handelsübliche WLAN-Sticks für die Kommunikation verwendet werden. Die Steuerung der Motoren wird durch ein eigens entwickeltes Board 'EROLF'[6] abgewickelt, welches seriell angesprochen wird. Das 'NimbRo' Team der Universität Bonn, Sieger im TeenSize Cup 2013, verwendet ein ähnliches System, benutzt aber ein noch leistungsstärkeres System und einen CM-730 zum Auslesen einiger Sensoren.[7]

Allerdings gibt es abseits des RoboCups einige Projekte, in denen ein Microcontroller mit Serial-2-WiFi-Modul in Kombination mit Google Protobuf eingesetzt wurden. Bemerkenswert ist beispielsweise das Projekt des Entwicklers der Google Protobuf Library 'NanoPB', welche auch in diesem Projekt verwendet wurde.[8] Dieser hat NanoPB als effiziente Alternative zu 'protobuf-c' entwickelt. Google Protobuf wurde als bandbreitenschonende Alternative zu einem FTP-ähnlichen ASCII Protokoll gewählt. In diesem Hobby-Projekt ging es darum, ein über GPRS ferngesteuertes Boot zu bauen, welches durch einen STM32 Microprozessor gesteuert werden sollte.

Der Haupteinsatzzweck der Serial-2-Wifi-Module liegt allerdings bei Sensornetzwerken. In der Regel greift ein Microcontroller Daten eines oder mehrerer Sensoren ab, verarbeitet diese und sendet sie dann an eine Form von auswertender Software auf einem Desktoprechner weiter. Dabei werden die Serial-2-Wifi Module meistens als reine Brückenadapter von Seriell auf WLAN verwendet. Dabei spielt die Verarbeitungszeit keine wesentliche Rolle, da das Versenden der Daten nicht zeitkritisch ist.

2.2 Regel der Kommunikation

Im Kapitel Einleitung und Unterkapiteln wurden bereits einige Regeln des RoboCup angesprochen. In diesem Kapitel sollen einige aus dem Regelwerk der TeenSize Klasse hervorgehoben werden, welche im Rahmen dieser Arbeit eine besondere Relevanz haben.

[6] Vgl. 'Berlin United – FUmanoids, EROLF' [7]
[7] Vgl. 'NimbRo-OP Humanoid TeenSize Open Platform Robot, Concept'[9]
[8] Vgl. Projekthomepage 'GPRS Boat'[11]

Das Regelwerk für die humanoiden Klassen[9] schreibt vor, dass ein Team nicht mehr als 1 Mb/s, also 125 kB/s Datenverkehr erzeugen darf. Das WLAN selbst wird durch einen offiziellen Router bereitgestellt. Das Erstellen und Verwenden eigener Drahtlosnetze ist untersagt. Allerdings sollen die Roboter auch in der Lage sein zu spielen, falls das WLAN versagt oder von schlechter Qualität ist. Zur Kommunikation darf ausschließlich das UDP Protokoll verwendet werden. Im Gegensatz zum Regelwerk der 'Middle Size' Klasse[10] nicht-humanoider Fußballroboter sind alle Formen des Datenverkehrs, etwa Unicast oder Broadcast, erlaubt.

Während des Spiels ist es den Robotern erlaubt, miteinander zu kommunizieren und Informationen auszutauschen. Es dürfen auch Informationen an eine Basisstation gesendet werden, allerdings darf sie während des Spiels selbst keine Daten senden. Diese darf also lediglich zum Überwachen und Evaluieren des Spielablaufs verwendet werden und nicht, um in den Spielablauf einzugreifen.

Über den verwendeten Standard des WLANs, etwaige Verschlüsselung oder andere Details, wie etwa Adressstruktur des Netzes oder Beacon Intervall wird keine Auskunft gegeben. Deshalb war es im Rahmen der Arbeit wichtig, ein WLAN-Modul zu wählen, welches möglichst alle gängigen WLAN-Standards und Verschlüsselungen unterstützt. Außerdem müssen Einstellungen bezüglich des WLANs ad-hoc vorgenommen werden können, damit auf Vorgaben z.B. in Form von SSID oder Adressbereich umgehend reagiert werden kann.

[9] Vgl. 'RoboCup Soccer Humanoid League, Rules and Setup'[3], Kapitel 4.5
[10] Vgl. 'Middle Size Robot League, Rules and Regulations for 2013' [2], Kapitel RC-4.2.5

3 Komponenten

Das folgende Kapitel beschäftigt sich mit den verwendeten Hard- und Softwarekomponenten. Hierbei wird sowohl auf die Vorgaben aus dem Projekt und bereits genannte Rahmenbedingungen eingegangen, als auch auf die daraufhin ausgewählten Komponenten. Insbesondere wird dabei auf die Eignung der jeweiligen Komponente erörtert.

3.1 Überblick über Hardwarekomponenten

Das folgende Kapitel beschäftigt sich mit den verwendeten Hardwarekomponenten.

3.1.1 CM-530

Abbildung 1: Der CM-530 Microcontroller

Der CM-530 ist der Controller, mit dem die FUB-KIT-Roboter gesteuert werden. Er verfügt über einen 32-Bit ARM Prozessor mit 72 MHz, 512 KB Speicher und 64 KB RAM. Zur Ein- und Ausgabe stehen ein Zigbee[11] Port, fünf Dynamixel Ports und sechs 5-Pin Aux-Geräteschnittstellen zu Verfügung. Außerdem sind fünf auslesbare Taster, sechs LEDs und ein Reset-Taster zum Zurücksetzen der Software vorhanden. Bei dem Reset-Taster ist anzumerken, dass die Stromversorgung selbst bei dauerhafter Betätigung an den Ein- und Ausgabeports, wie etwa Zigbee und Dynamixel, nicht abbricht. Dadurch bleiben eventuell vorher angenommene Hard- und Softwarezustände der Peripheriegeräte erhalten.

Desweiterem ist ein Mini-USB Port vorhanden, über den die Firmware installiert wird und welcher auch zur seriellen Ein- und Ausgabe genutzt werden kann. Die bereits fertiggestellten Teile der Firmware arbeiten als Echtzeitsystem. Das heißt, es ist kein Scheduler vorhanden. Außerdem ist keine dynamische Speicherverwaltung implementiert.

[11] Laut 'ROBOTIS e-Manual v1.15.00: CM-530'[32] auch 'Communication Jack' genannt.

3.1.2 Avisaro WLAN Module 1.0

Abbildung 2: Das Avisaro WLAN Module 1.0

Das 'Avisaro WLAN Module 1.0'[4] ist das Vorgängermodul des in diesem Projektes verwendetem und im nächsten Kapitel vorgestelltem GS1500M WLAN-Modul. Obwohl es in diesem Projekt nicht verwendet wird, sollen hier einige Details und Eigenschaften aufgezeigt werden, welche zu der Entscheidung geführt haben, ein neues Modul anzuschaffen.

Es verfüget über einen 'Befehlsmodus' zum Entgegennehmen von AT-Befehlen und einen 'Datenmodus' für das Versenden von Daten über eine vorher konfigurierte Verbindung.[12] Die Konfiguration findet über AT-Befehle im Befehlsmodus oder über ein Webinterface statt. Für den Wechsel vom Befehlsmodus in den Datenmodus genügt ein AT-Befehl. Um vom Datenmodus in den Befehlsmodus zurückzukehren müssen drei '+' Zeichen im Abstand von 500 ms gesendet werden. Der Vorteil dieser Architektur ist, dass man es direkt als Brücke zwischen Seriell und WLAN verwenden kann. Das Problem, vor allem im Kontext des FUB-KIT Projektes ist, dass sich Anfang und Ende eines Paketes nicht zuverlässig steuern lassen. Tests haben ergeben, dass dies anhand sehr kleiner Timeouts steuerbar ist, aber geringste Abweichungen können zu unerwünschten Paketaufteilungen führen. Das genaue Vorgehen beim Umsetzen der seriellen Daten in Pakete ist nicht dokumentiert.[14]

Außerdem ist es aufgrund der hohen Timeouts für die Escapesequenz von drei '+' Zeichen nicht sinnvoll möglich, während des Betriebs mit Hilfe von AT-Befehlen das Modul neu zu konfigurieren. Dadurch ist eine Kommunikation mit mehreren Teilnehmern nur über Broadcasts sinnvoll möglich.

Das 'Avisaro WLAN Module 1.0' besteht aus zwei Komponenten: einer Steuerplatine und dem eigentlichen WLAN-Modul als Steckkarte. Tests mit 'Wireshark'[16] haben ergeben, dass die Steuerplatine einen Puffer von ca. 250 Byte und die Steckkarte ca. 1200 Byte haben. Die Transferierung von 250 Byte vom einem in den anderen Puffer dauert eine bis zwei Millisekunden. Wenn die Steckkarte den nächsten Satz Daten sofort darauf erhält, werden diese zu einem Paket akkumuliert. In Tests konnten so Pakete bis zu einer größe von ca. 1100 Byte verschickt werden. Allerdings kam es dabei sehr häufig vor, dass Pakete aufgrund kleinster Verschiebungen im Timing in zwei oder mehr Pakete aufgeteilt wurden.

[12] Details siehe Dokumentation 'Avisaro WLAN Protokoll Adapter'[14]

Die noch vorzustellende Software FU-Remote erlaubt nur Nachrichten in einem Paket und erwartet eine Längenangabe der Nachricht. Da in diesem Projekt hauptsächlich UDP verwendet und im Rahmen des RoboCups ausschließlich erlaubt ist, ist eine Erweiterung des Protokolls auf gesplittete Nachrichten nicht sinnvoll, da die Reihenfolge der Pakete ist bei UDP nicht garantiert.

Aufgrund dieser Faktoren wurde entschieden, ein neues WLAN-Modul ohne die o.g. Einschräunkungen anzuschaffen.

3.1.3 GS1500M WLAN-Modul

Abbildung 3: Das GS1500M WLAN-Modul

Das GS1500M ist ein WLAN-Modul, welches über die Seriellen Schnittstellen SPI und UART mittels AT-Befehlen gesteuert werden kann. Der Benutzer kommt dabei, wie beim 'Avisaro WLAN Module 1.0' auch, nicht mit der Implementierung von Sockets oder Treibern in Kontakt, sondern es wird eine auf AT-Befehlen basierende Schnittstelle angeboten. Zur Kommunikation werden die WLAN-Standards IEEE 802.11b/g/n[37] unterstützt. Außerdem muss eine externe Antenne über eine MC-Card Buchse angeschlossen werden.

Dieses Modul weist im Vergleich zu den meisten anderen WLAN-Modulen, wie auch zum Vorgängermodul, einen signifikanten Unterschied im Befehlsinterface auf. Der typische Verwendungszweck seriell angesprochener WLAN-Module ist als Brückenadapter für ältere Hardware, die netzwerkfähig gemacht werden soll. Ein weiterer typischer Anwendungsfall ist das Auslesen und versenden von Sensordaten an eine i.d.R. fest konfigurierte Basisstation. Sie werden einmal für ein bestimmtes WLAN und einen bestimmten Kommunikationspartner konfiguriert und kommunizieren über WLAN ohne Änderungen am alten System vornehmen zu müssen.

Das GS1500M hat ausschließlich einen Befehlsmodus.[13] Es können mehrere Verbindungen über AT-Befehle geöffnet werden. Diese erhalten als Verbindungsnummer eine ID. Um ein Paket zu senden, wird vor der zu versendenden Nachricht der WLAN-Sendeheader an das

[13] Vgl. Dokumentation 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20]

WLAN-Modul geschickt. Dieser besteht aus einem Escape-Zeichen, gefolgt von der Verbindungsnummer, Pakettyp und der Länge der zu schickenden Nachricht. Direkt danach wir die Nachricht selbst an das WLAN-Modul geschickt. Das Empfangen von Nachrichten geschieht mit einem ähnlichen, vorangehenden Header.[14] Dadurch lassen sich Statusmeldungen und Nachrichten präzise voneinander trennen. Wie zuvor beschrieben, macht dieses Befehlsinterface das WLAN-Modul als reines Brückenmodul, ungeeignet. Des Weiteren ist zu beachten, dass ein- und ausgehende Pakete des GS1500M maximal 1400Bytes groß sein dürfen.[15]

3.1.4 HaViMo2

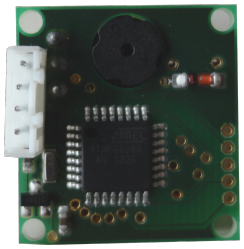

Abbildung 4: Die HaViMo2 Kamera

Das HaViMo2 ist eine Kamera, welche über Dynamixel angesprochen werden kann. Sie macht bis zu 19 Bilder pro Sekunde bei einer Auflösung von 160x120 Pixeln.[16] Zusätzlich verfügt sie über Hardware, welche die aufgenommenen Bilder in der Form vorverarbeiten kann, dass Objekte anhand von Farben erkannt werden.

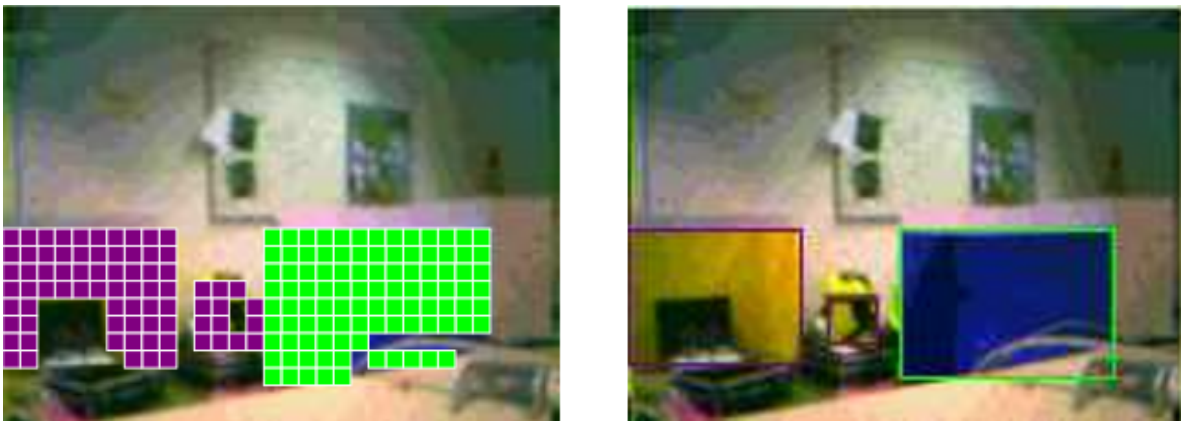

Abbildung 5: Links: Beispiel 'Gridding' Algorithmus, Rechts: Beispiel 'Region Growing' Algorithmus [17]

Dazu stehen ein 'Region Growing' Algorithmus und ein 'Gridding' Algorithmus zur Verfü-

[14] Packetheader Beschreibung siehe Dokumentation 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20] 6 'Appendix'
[15] Vgl. Dokumentation 'ROBOTIS e-Manual v1.15.00: CM-530'[32], 3.4.1 'Bulk data Tx and Rx'
[16] Siehe Spezifikationen 'HaViMo2 Image Processing Module'[23]
[17] Abbildung Vgl. 'HaViMo2 Image Processing Module'[23]

gung. Der 'Region Growing' Algorithmus legt um Flächen einer bestimmten Farbe ein Rechteck. Dazu wird einer Farbe zu solch einer rechteckigen Fläche auf dem Kamerabild zugeordnet. Die Koordinaten zweier relevanter Eckpunkte werden dann auf Anfrage über den Dynamixel Port verschickt. Der 'Gridding' Algorithmus erstellt ein Raster, in welches erkannte Farbflächen eingetragen werden. Ein Beispiel für die Arbeitsweise der beiden Algorithmen ist in Abbildung 5 dargestellt. Die Daten dieser beiden Algorithmen sollen im Laufe dieser Arbeit ausgelesen und zur Darstellung per WLAN an die noch vorzustellende Software FU-Remote gesendet werden.

3.2 Überblick über Softwarekomponenten

Das folgende Kapitel beschäftigt sich speziell mit den verwendeten Softwarekomponenten.

3.2.1 'easy-functions' CM-530 Bibliothek

'easy-functions'[29] ist eine Bibliothek für den CM-530, welche die von ROBOTIS[34] bereitgestellten Funktionen verbessert und um einige Wrapperfunktionen erweitert. Dabei wurden auch einige Bugs aus dem Code von ROBOTIS behoben. Sie ist außerdem mit dem HaViMo2 kompatibel.

3.2.2 Nanopb Protobuf Bibliothek

Nanopb [12] ist eine C Implementierung für einen Encoder bzw. Decoder des Google Protobuf Formats. Kennzeichnend für diese Bibliothek ist ihre größe von ca. 2 bis 10 kB kompiliert. Der Speicherverbrauch beläuft sich auf ca. 300 Bytes Größe, Protobuf-Nachrichten selbst nicht eingerechnet. Sie verwendet außerdem keine dynamische Speicherverwaltung und kommt ohne weitere Bibliotheken aus. Sie unterliegt einigen Einschränkungen, welche für dieses Projekt allerdings nicht relevant sind. Der interessierte Leser kann diese auf der Homepage nachschlagen. [12]

Nanopb wurde zum einen gewählt, weil es eine reine C Implementierung ohne Abhängigkeiten ist und zum anderen, weil die Implementierung sehr minimalistisch ist. Die Speicherverwaltung ist zwar statisch, lässt sich aber mit einer Implementierung von 'Malloc' ohne größere Schwierigkeiten zu einer dynamischen machen.

Sollte eine dynamische Speicherverwaltung nötig werden, existiert ein Branch von NanoPB, welcher eine dynamische Speicherverwaltung implementiert.[18] Dabei ist anzumerken, dass bei Verwendung dieses Branch etwaige Footprint-Analysen, welche im Rahmen dieser Arbeit getroffen werden, nicht mehr stimmen.

[18] Code repository online unter 'Nanopb - protocol buffers with small code size, dynamic alloc dev (Branch)'[30]

3.2.3 FU-Remote

FU-Remote ist eine in Java geschriebene Software, welche u.A. Debug-Informationen von den Robotern anfordert, entgegen nimmt und diese datenspezifisch visualisieren kann. FU-Remote wird seit einigen Jahren vom 'FUmanoids' Team der FU-Berlin entwickelt und eingesetzt. Jährliche Codereleases sind auf der Projekthomepage zu finden.[6]

Für die Implementierung des Protokolls benutzt FU-Remote das Protobuf Format.[10] Diesem werden noch einige projektspezifische Headerinformationen hinzugefügt. Außerdem bettet es den GameController ein.[35] Der Gamecontroller ist eine eigenständige Java-Anwendung, welche in Robocup-Tournieren als Schiedsrichter dient. Dieser teilt den Spielern z.B. mit, dass das Spiel beginnt, endet, oder wer auf welches Tor spielt. Der GameController verwendet allerdings ein eigenes, simples, binäres Protokoll.

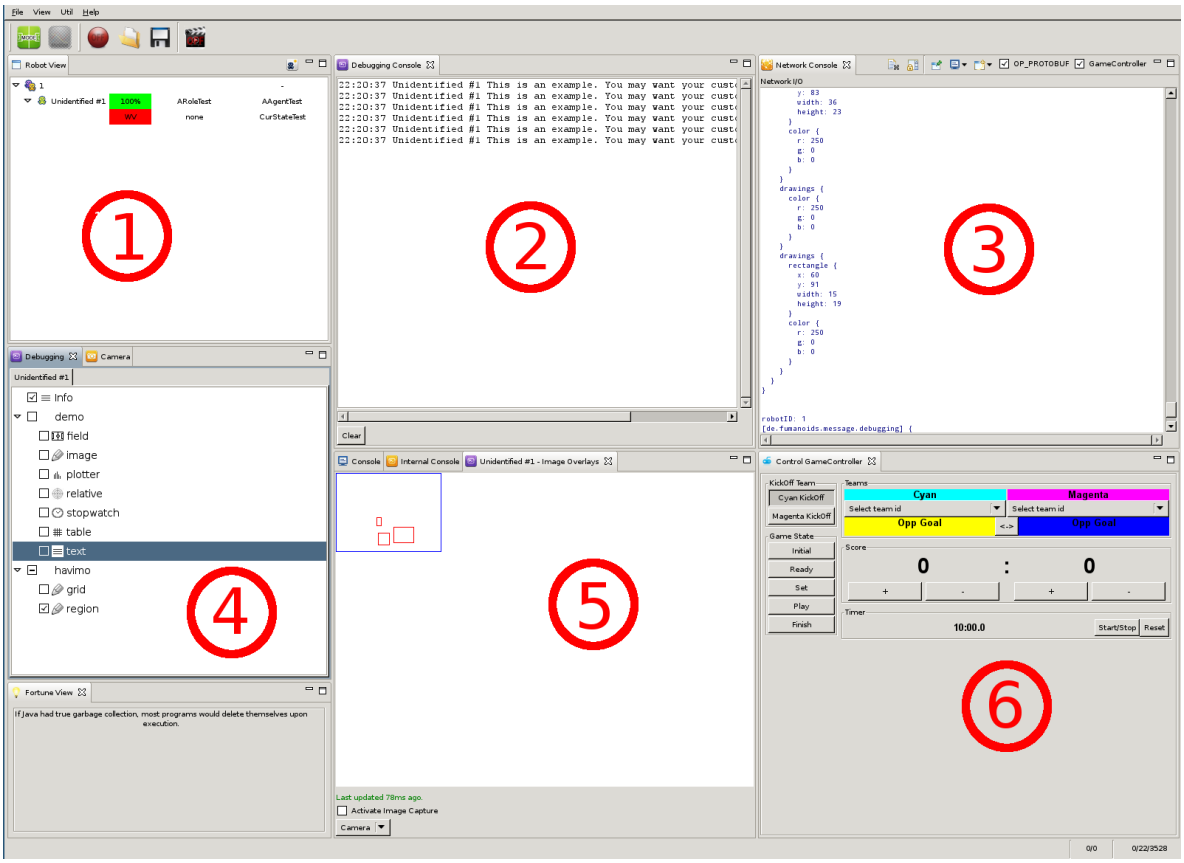

Abbildung 6: Übersicht einiger FU-Remote Funktionen

In Abbildung 6 sind die für diese Arbeit relevanten Funktionen bzw. Ansichten von FU-Remote exemplarisch dargestellt. FU-Remote implementiert frei verschiebbare Karteikarten, wodurch sich die Ansicht nach Belieben umgestalten lässt. Der 'Robot View' (1) zeigt alle einmal 'gesehenen' Roboter an. 'Gesehen' bedeutet in diesem Fall, dass mindestens einmal ein Paket von ihm erhalten wurde. Zu jedem Roboter werden einige Kenndaten, wie beispielswei-

se dessen ID, Name, Akkuladestand oder Rolle im Spiel, etwa 'Stürmer', angezeigt.

Die 'Debugging Console' (2) gibt nur Text aus und versieht diesen mit einem Zeitstempel. Die 'Network Console' (3) bietet die Möglichkeit ein- und ausgehende Netzwerkdaten anzeigen zu lassen. Protobuf-Nachrichten werden dazu in menschenlesbarer Textform serialisiert. In der 'Debugging' Kategorie lässt sich für jeden Roboter individuell auswählen, welche Debug-Daten dieser schicken soll. Es wird daraufhin eine Anfrage an den Roboter geschickt. Bei Doppelklick auf einen Listeneintrag wird die dem Datentyp zugeordnete Visualisierung geöffnet. In Karteikarte (5) 'Image Overlay' ist ein Beispiel für Debug-Daten geöffnet. In diesem Fall handelt es sich um die Visualisierung des Ergebnisses des 'Region Growing' Algorithmuses der HaViMo2 Kamera. Das blaue Rechteck stellt das gesamte Kamerabild dar, die roten Rechtecke einzelne erkannte Objekte. Das letzte Fenster 'Control Game Controller' (6) bettet die gesamte GameController Software ein. Hier lassen sich wie eingangs erwähnt alle Schiedsrichter-Kommandos erteilen.

3.2.4 Das FU-Remote Protokoll

Im Kapitel'Struktur der Software wird das Senden, Empfangen und Verarbeiten von Nachrichten detailliert besprochen. Dafür ist eine grundlegende Kenntnis des FU-Remote Protokolls notwendig. Zu diesem Zweck wird das Protokoll im Folgenden kurz vorgestellt. Die GameController Nachrichten werden dabei übergangen. Sie lassen sich sehr einfach auf ein Struct casten und ggf. die Byte-Reihenfolge anpassen.

Der FUmanoids-Header besteht aus folgenden Daten:

- 16 Bit Nachrichtentyp, immer '42'
- 8 Bit Flags
- 8 Bit Alignment
- 32 Bit reserviert (je nach Version)
- 32 Bit Nachrichtenlänge

Alle Zahlen sind in 'Big-Endian' anzugeben.
In den neueren Codereleases ab 2014 wird nur noch die Nachrichtenlänge vorangestellt. Alle anderen Headerinformationen entfallen. Diese Neuerung wurde im Rahmen dieser Arbeit nicht berücksichtigt, da die neueren Versionen immer noch das alte Format unterstützen.[19]

Die Protobuf-Nachricht folgt immer einem grundlegendem Schema. Sie sind eine Anwendung des 'Extension Objects Pattern'.[21] Alle 'Messages' erweitern 'Message' mit Hilfe des

[19]Siehe 'Berlin United Framework Coderelease'[5] 2013 bzw. 2014, comm.cpp

'extends' Schlüsselwortes. 'Message' beinhaltet nur diese 'Extension' und die ID des absendenden Roboters. FU-Remote hat für dieses Feld eine gesonderte ID. Die 'Messages', welche als Erweiterung auftreten, können beliebige Inhalte haben, welche im Einzelnen zu definieren sind.

Der Vorteil an diesem Pattern, speziell in diesem Fall ist, dass die möglichen Erweiterungen nicht bekannt sein müssen. So können einzelne Projekte ihre eigenen Erweiterungen implementieren und das Protokoll erweitern, ohne mit anderen zu interferieren.

4 Inbetriebnahme

Das folgende Kapitel beschreibt, wie man die aus dieser Arbeit hervorgehende Software installiert und in Betrieb nimmt. Dies umfasst alle Arbeitsschritte, bis hin zu den sichtbaren Debug-Informationen in FU-Remote.

4.1 Verschaltung der Hardware

Das HaViMo2 kann mit einem geeigneten Kabel an einen beliebigen Dynamixel-Port des CM-530 angeschlossen werden.

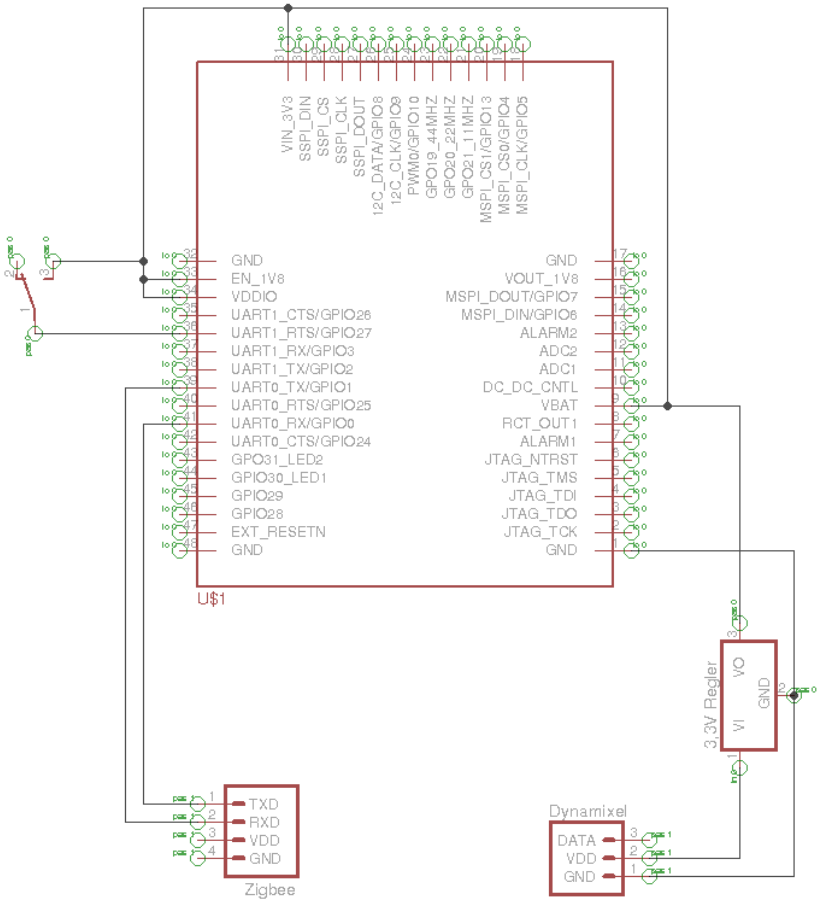

Abbildung 7: Schaltplan zur Inbetriebnahme des GS1500M[20]

In Abbildung 7 ist die Verschaltung des GS1500M WLAN-Moduls dargestellt. Da die Leistungsaufnahme des WLAN-Moduls für den Zigbee-Port zu hoch sein kann, wird die Spannungsversorgung von einem der Dynamixel-Ports bezogen. Da die Dynamixel-Ports eine Spannung von 12 V liefern, das WLAN-Modul jedoch eine Versorgungsspannung von 3,3 V benötigt, ist hier ein 3,3 V Regler nötig. Die serielle Kommunikation findet über den Zigbee-

[20] Abbildung Vgl. 'Eine WLAN-Platine zum selber bauen'[25]

Port statt.
Mit dem Schalter kann eine logische 1 auf den Port 'GPIO27' gelegt werden. Dadurch geht das Modul in den sog. 'Flash Modus', in welchem die Firmware auf das Modul geschrieben werden kann.

4.2 Installation der Firmware GS1500M

Da keine Brückenfirmware für den CM-530 vorhanden ist, muss zur Installation der Firmware auf den GS1500M ein beliebiger Rechner mit Windows®Betriebssystem über einen USB nach UART Adapter direkt an die UART Schnittstelle des WLAN-Moduls angeschlossen werden. Damit das WLAN-Modul in den 'Flash Modus' geht, ist es notwendig, auf den Pin 'GPIO 27' während des Einschalten des Moduls eine logische 1 zu legen. Im vorherigen Kapitel wurde zu diesem Zweck ein Schalter integriert.

Die Software zum Flashen der Firmware und die Firmware selbst sind vom Hersteller erhältlich. Die Software zum Flashen ist allerdings nur für Windows erhältlich, daher die Betriebssystemeinschränkung. Die Firmware selbst besteht aus drei Binärdateien. Eine für die WLAN Software und zwei für die Anwendungssoftware. Als Anwendungssoftware wurde für dieses Projekt eine Software mit 'Web Provisioning' gewählt. Diese ist in der Lage selbstständig einen Webserver zu starten, über den sich die Netzwerkkonfiguration des WLAN-Moduls ändern lässt. Sonstige Features und Informationen zu alternativen Firmwares kann der interessierte Leser im Dokument 'Provisioning Methods with S2W'[18] nachschlagen.

Die Verwendung der Flash Software ist ausreichend in mitgelieferten Dateien dokumentiert, es sollte jedoch angemerkt werden, dass diese einige Fehler enthält.[21] Im Testfall kam es vor, dass diese Probleme hatte, den Speicher des WLAN-Moduls zu löschen. Um dieses Problem zu umgehen, kann die Flash Software von WizNet®verwendet werden. Die WLAN-Module von WizNet beinhalten auch einen von Gainspan®gefertigten Chip und verwenden deshalb ein Derivat von dessen Software. Sie ist auf der Internetseite von WizNet nicht mehr verfügbar. Ggf. muss hier eine neuere Version von Gainspan verwendet werden oder der Hersteller kontaktiert werden. Das Schreiben der Binärdateien sollte in jedem Fall mit der Flash Software von Gainspan erledigt werden.

4.3 Starten der Beispielsoftware

Vor dem Übersetzen des Quellcodes sollten ggf. die Netzwerkeinstellungen angepasst werden. In der Datei 'comm.c' befindet sich die Funktion 'configure_wlan_fubkit', welche eine Beispielkonfiguration für ein WPK-PSK Netzwerk vornimmt. Außerdem konfiguriert sie das WLAN-Modul darauf, beim Start automatisch eine Verbindung herzustellen. Es ist beispielsweise auch möglich, das WLAN-Modul ein Ad-hoc Netzwerk erstellen zu lassen, so dass das Netzwerk über das Webinterface konfiguriert werden kann.

[21] Erfahrungen aus Tests mit Version 'DOS_ver_7_151'

Die Software zum Schreiben der Firmware auf dem CM-530 ist unter Linux in dem Projekt 'DARwIn-OP' zu finden. Dies ist ein Open-Source Projekt direkt vom Hersteller Robotis.[22] Für Windows-Systeme ist es in dem Softwarepaket 'RoboPlus' enthalten. Anleitungen zu Installation sind in der Robotis E-Manual zu finden.[23] Im Verzeichnis 'fubkit' befindet sich eine Makefile, dessen Ausführung im selben Verzeichnis die Datei 'fubkit.hex' erstellt. Diese ist nach Anleitung auf dem CM-530 zu installieren.

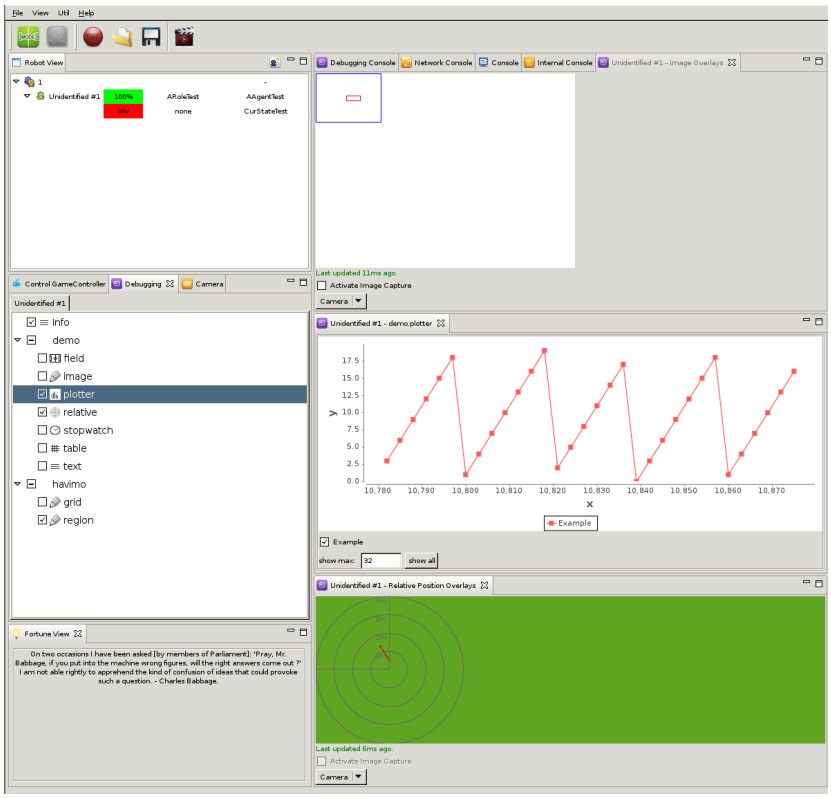

Abbildung 8: Beispielkommunikation mit FU-Remote und ausgewählten Testdaten

Nachdem die Firmware installiert und das Netzwerk konfiguriert ist, beginnt die Beispielsoftware automatisch eine UDP-Kommunikation auf Port 11011 ausgehend und auf Port 3838 eingehend. Dabei werden im 200ms-Takt 'Keep-Alives' gesendet. Das heißt, insofern das Netzwerk richtig konfiguriert wurde, wird die Beispielsoftware automatisch als Roboter in FU-Remote angezeigt. Dies setzt die richtige Konfiguration der Ports in FU-Remote voraus.

[22] Download unter 'DARwIn-OP downloads page'[26]
[23] Anleitung unter 'ROBOTIS e-Manual v1.16.00: Firmware Installer'[33]

5 Struktur der Software

Das folgende Kapitel beschäftigt sich mit der Struktur der Software. Zunächst wird dazu die Handhabung des Frontends erklärt, um darzulegen, was die Software tut bzw. tun kann. Danach wird auf die Interna der einzelnen Funktionen und Programmbibliotheken eingegangen und erklärt, wie die Software bestimmte Dinge erledigt. Da es sich um ein Echtzeit-System handelt, wurden für alle Aufgaben, welche sehr lange dauern oder eine unbekannte Länge haben, Funktionen zum Pollen entwickelt.

5.1 Das Frontend

Das Frontend stellt im Wesentlichen vier Funktionen bereit, welche für das Senden und Empfangen von Daten über das WLAN-Modul interessant sind. Diese sind 'pollInitNetwork', 'recvWlanData', 'sendMsgToCon' und 'processData'.

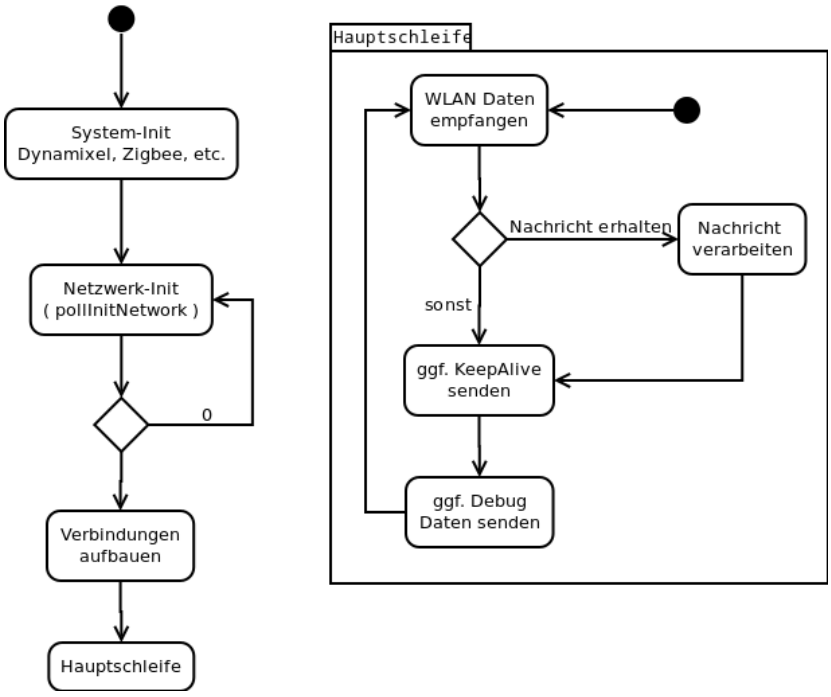

Abbildung 9: Aktivitätsdiagramm für Beispielnutzung des Frameworks

In Abbildung 9 ist ein Aktivitätsdiagramm dargestellt, welches einen beispielhaften Ablauf zum Verbindungsaufbau und zur Kommunikation über das Netzwerk darstellt. Die Fehlerbehandlung ist in diesem Diagramm nicht berücksichtigt. Das Aufbauen einer Verbindung muss manuell über einen AT-Befehl geschehen. Im Rahmen dieses Projektes ist dafür nur 'AT+NCUDP' relevant.[24] Der Erfolg und die Verbindungsnummer, welche diese Verbindung

[24]Spezifikation von 'AT+NCUDP' unter 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20] 4.10.2 'UDP Clients'

identifiziert, werden von der noch vorzustellenden Funktion 'recvWlanData' im Framework notiert und sind abfragbar. Es können bis zu 16 unidirektionale Verbindungen geöffnet werden.[25]

Zunächst zu 'pollInitNetwork'. Beim Einschalten des WLAN-Moduls ist es nicht direkt betriebsbereit. Davon ausgehend, dass automatisches Verbinden[26] konfiguriert wurde, muss das WLAN-Modul, nachdem die Software hochgefahren wurde, zunächst die Verbindung aufbauen. Dies kann unterschiedlich lange dauern und ggf. fehlschlagen. 'pollInitNetwork' kann so oft aufgerufen werden, bis es 'TRUE' zurückgibt. Danach ist das WLAN-Modul sende- und empfangsbereit. Die Funktion pollt intern die serielle Ausgabe des WLAN-Moduls und extrahiert wesentliche Informationen, wie etwa Verbindungserfolg, Fehlschlag etc.. Sollte der Verbindungsaufbau fehlschlagen, so wird sie es automatisch erneut versuchen. Dieser Arbeitsschritt muss ausgeführt werden, bevor mit dem WLAN-Modul gearbeitet werden kann.

'recvWlanData' erwartet ein Byte-Array und einen 32Bit Integer Pointer als Parameter und liefert ein Byte als Antwort. Das Array sollte vom Benutzer erstellt und mit Nullen initiiert werden. Der Integer sollte auf 0 gesetzt werden. Nun kann die Funktion solange gepollt werden, bis sie 'TRUE' zurückgibt. Sollte die Funktion dabei andere Meldungen als Pakete aus dem WLAN lesen, werden diese verarbeitet und nach PCU ausgegeben. Wenn die Funktion 'TRUE' zurück gibt, bedeutet dies, dass ein WLAN-Paket vollständig empfangen wurde und in dem Byte-Array inklusive WLAN-Empfangsheader bereit steht.[27] Für den Fall, dass die WLAN-Nachricht manuell verarbeitet werden soll, stehen die Funktionen 'extractHeaderSize' und 'extractDataSize' bereit, um die Größe des WLAN-Empfangsheader bzw. der Nachricht auszulesen.

Mit der Funktion 'sendMsgToCon' lässt sich eine Nachricht als Paket an eine anzugebende Verbindungsnummer senden. Der WLAN-Sendeheader wird automatisch vorangestellt. Die Verbindung muss zuvor mit einem entsprechenden AT-Befehl aufgebaut worden sein.[28] Nach dem Verbindungsaufbau wird das WLAN-Modul eine Verbindungsnummer als String ausgeben. Diese wird von 'recvWlanData' automatisch geparst und in ein Array übertragen.

Des Weiteren gibt es noch die Funktion 'processData', welche zum Verarbeiten von WLAN-Nachrichten des FU-Remote Protokolls genutzt werden kann. Diese nimmt den Byte-Array mit der WLAN-Nachricht und dessen Größe entgegen. Sie decodiert ggf. Protobuf-Nachrichten und ruft dann ein Callback auf. Zu diesem Zweck steht ein globales Array bereit, in dem Protobuf-Nachrichtentypen einem Callback zugeordnet werden können. Der Quellcode, welcher die Daten aus der Nachricht sinnvoll verarbeitet, muss also in diesem Callback implemen-

[25] Vgl. 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20] 2 'Interface Architecture'
[26] Siehe Dokumentation 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20], Kapitel 4.14.3 und 4.14.4
[27] Packetheader nach 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20] 6 'Appendix'
[28] Siehe Dokumentation 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20], Kapitel 4.10.2 und 4.10.3

tiert werden. Der gesamte Vorgang wird im Kapitel 'Verarbeiten von eingehenden Nachrichten' detailliert besprochen.

5.2 Kommunikation mit dem WLAN-Modul

Im folgenden Abschnitt wird das Senden und Empfangen von Daten an das WLAN-Modul behandelt.

5.2.1 Problem der Nebenläufigkeit

Das Problem bei der Kommunikation mit dem WLAN-Modul ist, dass direkt auf einen Befehl nicht die Antwort erwartet werden kann, da die Kommunikation asynchron ist und dass deshalb beim Parsen auf den Typ der Meldung geachtet werden muss. Gesendete Befehle werden vom WLAN-Modul aber in der Reihenfolge Abgearbeitet, in der sie empfangen wurden. Dadurch bleiben auch die mit den Befehlen zusammenhängenden Antworten in der richtigen Reihenfolge. Also z.B. 'Befehl A, Befehl B' wird immer eine Ausgabe erzeugen, wie '(...), Antwort auf A, (...), Antwort auf B, (...)'. Das Versenden von Nachrichten bildet allerdings eine Ausnahme. Hierbei sei angemerkt, dass dieses Verhalten durch Tests nachgewiesen wurde und nicht dokumentiert ist.

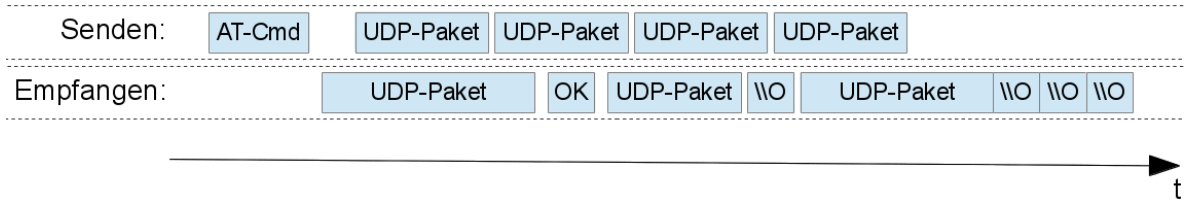

Abbildung 10: Beispielsequenz für Ein- und Ausgehende Daten

Ein Beispiel für eine solche Kommunikation ist in Abbildung 10 dargestellt. Es können auch WLAN-Nachrichten oder Statusmeldungen vom WLAN-Modul unerwartet dazwischen auftauchen. '\\O' Steht hierbei für für die Meldung des WLAN-Moduls, dass ein Paket erfolgreich gesendet wurde. 'OK' sei eine Antwort im Textformat auf einen AT-Befehl.

Daraus wird deutlich, dass sich die Befehle, welche an das WLAN-Modul gesendet werden, nicht auf einfache Funktionen abbilden lassen. Zum einen ist die Rückgabe verzögert und ggf. hinter anderen Meldungen vom WLAN-Modul im Puffer, zum Anderen ist die Bearbeitungsdauer nicht mit Sicherheit zu bestimmen. Das Aufbauen einer TCP-Verbindung kann beispielsweise aufgrund von Latenzen mehrere Sekunden oder nur Millisekunden dauern, wohingegen eine UDP Verbindung nicht aufgebaut werden braucht. Man konfiguriert nur eine neue Verbindung im WLAN-Modul, wobei keine Kommunikation statt findet.
Lösungsansätze für eine Abbildung auf funktionsartiges Verhalten existieren zwar, sind aber nicht praktikabel. Beispielsweise könnte man alle Antworten vor der Erwarteten puffern, jedoch ist der Speicher des Microcontrollers sehr begrenzt und es bleibt das Problem der mög-

lichen großen Verzögerung. Ggf. kommt auch nie eine Antwort und es müsste ein Timeout eingebaut werden.

Aus diesen Gründen wurde die bereits vorgestellte asynchrone Lösung mit Polling implementiert.

5.2.2 Empfangen von dem WLAN-Modul

Das Empfangen von Daten vom WLAN-Modul über die serielle Schnittstelle geschieht wie bereits im Kapitel 'Das Frontend' erwähnt über die Funktion 'recvWlanData'. Diese ist konzipiert, um gepollt zu werden und wird zurückkehren, sobald sich kein Zeichen mehr im Eingangspuffer befindet. Außerdem muss es ihr möglich sein, verschiedene WLAN-Header zu erkennen und von Textmeldungen unterscheiden zu können.

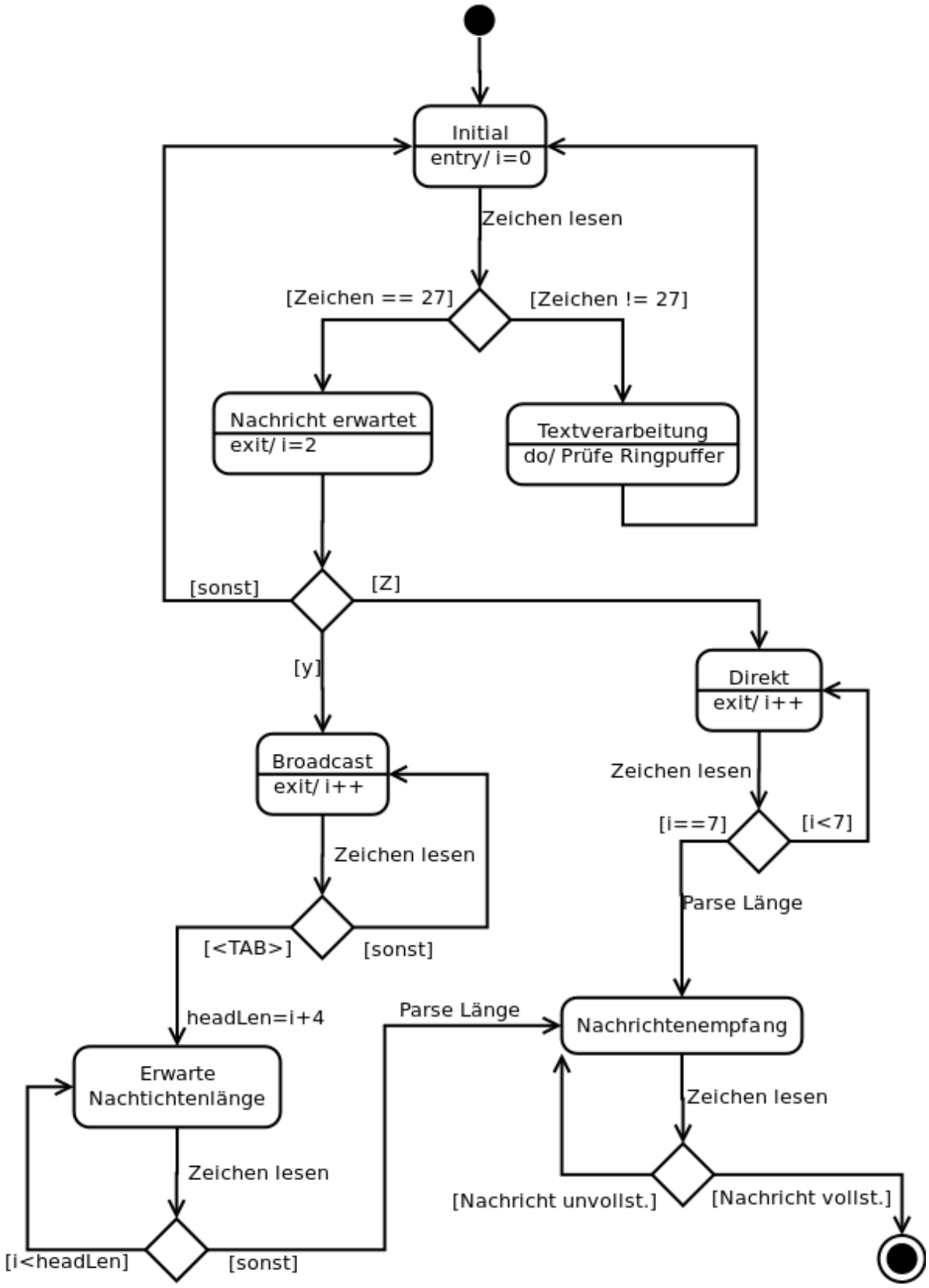

Abbildung 11: Zustandsdiagramm zum Parsen der seriell eingehenden Daten

In Abbildung 11 ist ein Zustandsdiagramm dargestellt, welches das Parsen der seriell eingehenden Daten darstellt. Zur Übersichtlichkeit des Diagramms wurden die Polling-Eigenschaft weggelassen. Bei jedem Vorgang 'Zeichen lesen' müsste es zur vollständigen Korrektheit die Möglichkeit geben, dass kein Zeichen mehr vorhanden ist. In diesem Fall würde der Automat in einen Wartezustand gehen und beim nächsten Poll an selber Stelle erneut versuchen ein Zeichen zu lesen.

Nun zur Arbeitsweise des Automaten. Die Variable 'i' besagt, wie viele Zeichen bereits gelesen wurden. Alle WLAN-Nachrichten beginnen mit dem Dezimalwert 27 als Escape-Zeichen. Solange dieser nicht gelesen wird, werden gelesene Zeichen in einem Ringpuffer abgelegt und dieser auf Schlüsselwörter, wie etwa für einen Verbindungsabbruch, überprüft. Sollte eine 27 gelesen werden, gibt es 3 mögliche Folgezeichen, insofern sich das WLAN-Modul im 'Bulk Data Modus' befindet. Headerinformationen für andere Modi sind nicht vorgesehen, aber vergleichsweise einfach auf Basis der bisherigen implementierbar. Sollte es sich beim zweiten Zeichen um ein anderes handeln als ASCII 'y' oder ASCII 'Z', so wird es verworfen und der Automat geht in den Ausgangszustand, denn bei der Nachricht muss es sich um eine Erfolgs- oder Fehlermeldung über ein versendetes Paket handeln. Diese sind nur 2 Zeichen lang und zu ignorieren, da es sich bei der Kommunikation im Wesentlichen um UDP-Datenverkehr handelt. Eine Erfolgsmeldung kann beispielsweise keine Aussage darüber treffen, ob das Paket tatsächlich beim Empfänger angekommen ist, sondern nur, dass es versendet wurde.

Ein 'Z' ist ein Indikator für ein direkt an die IP des WLAN-Moduls adressiertes Paket. Der WLAN-Header hat in diesem Fall eine konstante Länge und beinhaltet die Verbindungsnummer, sowie die Länge. Nachdem die Länge gelesen wurde, müssen nur noch exakt so viele Zeichen wie im WLAN Header angegeben gelesen werden. Danach ist die Nachricht vollständig und muss vom Benutzer verarbeitet werden.

Wird als zweites Zeichen der Nachricht hingegen ein ASCII 'y' gelesen handelt es sich um einen Broadcast. Diese Unterscheidung ist aufgrund der unterschiedlichen Header notwendig. Da IP und Ports, welche im WLAN-Header enthalten sind, eine variable Länge haben, muss hier auf das 'Tab' Zeichen gewartet werden, welches vor dem Nachrichtenlänge im WLAN Header steht. Sobald dieser gelesen und geparst wurde, verfährt der Automat analog zur direkten Nachricht.

Es ist zu beachten, dass es am Ende der Nachricht im 'Bulk Data Modus' kein Escape-Zeichen oder ähnliches gibt. In anderen Modi gibt es dieses jedoch. Der interessierte Leser findet im Programmierhandbuch[20] im Kapitel 4.10.16 und 6 mehr zu diesem Thema.

5.3 Versenden von Nachrichten

Es bietet sich an, das WLAN-Modul auf den 'Bulk Modus' zu konfigurieren. Dies ermöglicht das Versenden und Empfangen von Binärdaten. Ein 'ASCII Modus' sowie einen 'Raw Modus' sind ebenfalls verfügbar, aber ungeeignet um in Google Protobuf codierte Nachrichten

zu senden und zu empfangen. Im ASCII Modus werden gesendete Nachrichten nicht über ihre Länge definiert, sondern mit einer Escapesequenz beendet.[29] Der Raw Modus ist nur für den BACnet Support gedacht,[30] welcher für dieses Projekt nicht relevant ist.

Mit jeder Nachricht, welche vom WLAN-Modul zu einem TCP- bzw. UDP-Paket gemacht wird, muss mit einer Kopfzeile versehen werden. Diese wurde bereits in vorherigen Kapiteln erwähnt und soll hier zum besseren Verständnis einmal schematisch dargestellt werden:

<ESC>Z<Verbindungsnr.><Datenmenge><Daten>

Das Escape-Zeichen ist Dezimal 27. Das 'Z' ist eine Kennung für den Bulk Modus. Die Verbindungsnummer gibt den Socket an, über den gesendet werden soll. Diese Nummer teilt das WLAN-Modul nach erfolgreicher Verbindung über die serielle Schnittstelle durch 'CONNECT <Verbindungsnr.>' mit. Diese Zahl wird, wie im Kapitel Das Frontend erwähnt, automatisch vom Framework notiert. Die Datenmenge ist in Byte als vierstelliger ASCII String, ggf. mit führenden Nullen, anzugeben. Diese darf die Puffergröße von 1400 Bytes nicht überschreiten. Nachdem die gegebene Anzahl an Zeichen gelesen wurde, wartet das WLAN-Modul wieder auf neue Eingaben.

Zum Versenden von Daten wird vom Framework folgende Funktion angeboten:

sendWlanDataToCon(uint8_t *msg, uint32_t len, uint8_t con)

5.4 Protobuf und Nanopb

In diesem Kapitel soll dem Leser zum einen das Protobuf Format näher gebracht werden, und zum anderen erläutert werden, wie Nanopb dieses encodiert bzw. decodiert.

5.4.1 Das Protobuf Format

Das Protobuf Format ist im Vergleich zu anderen Serialisierungsformaten wie z.B. XML und SOAP[31] sehr minimalistisch. Ein grundlegendes Verständnis des Formats ist notwendig, um die Arbeitsweise von Nanopb nachvolziehen zu können.

Protobuf serialisiert eine Nachricht immer nach einem einfachen Muster. Zum Encodieren wird eine Nachricht Feld für Feld durchgegangen. Ein Datum erhält immer einen Header, den sog. Tag. Dieser Tag besteht aus der Feld Nr. und dem 'Wire Type' (i.F. Typ), also der Datentyp und ist als Varint formatiert.[32] Die Feld Nr. entspricht der in der *.proto vergebenen ID. Die drei niederwertigsten Bits enthalten den Typ, alle höherwertigen Bits die ID. Mögliche Typen sind:

[29] Details unter 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20] 6 'Appendix'
[30] Details unter 'Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3'[20] 3.4.2 'Raw Data Handling'
[31] Siehe 'SOAP Version 1.2'[28]
[32] Analog zu 'Protocol Buffers, Encoding'[22]

Typ	Bedeutung	genutzt für
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	längenbegrenzt	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

Tabelle 1: 'Wire Types' des Protobuf Formats

Für Strings folgt dann ein Varint für die Länge des Strings, gefolgt von dem String selbst. Submessages werden ebenfalls als Strings behandelt. Alle nummerischen Datentypen haben keine Längenangabe. Datentypen mit fest definierter Länge, wie beispielsweise 'int32', werden einfach in voller Länge in Little-Endian Ordnung serialisiert.

Interessant ist hierbei vor Allem die Serialisierung der Varint Datentypen, da diese eine variable Länge haben, aber keine Längenangabe. Bei einem Varint wird bei jedem Byte das höchstwertigste Bit dafür genutzt, um anzuzeigen, ob die Zahl nach dem aktuellen Byte weitergeht. Die restlichen sieben Bits sind Nutzlast. Die Zahl ist dabei in Little-Endian Ordnung. 'Signed' Datentypen wird eine zickzack artige Bijektion verwendet. Diese entspricht einer Bijektion zwischen ganzen und natürlichen Zahlen. Die Abbildungen lauten wie folgt:
Encodierung:

$$f(x) = \begin{cases} -(2x+1), & \text{x mod } 2 = 0 \\ 2x & \text{x mod } 2 = 1 \end{cases}$$

Für die Encodierung gilt also beispielsweise: 0 -> 0, -1 -> 1, 1 -> 2, -2 -> 3, usw..

Nachdem ein Feld serialisiert wurde, folgt direkt das nächste. Die Feld-Nummern müssen dabei nicht in aufsteigender Ordnung sein. Für wiederholte Felder kommt ein Datum doppelt vor. Diese doppelten Felder müssen nicht unbedingt nebeneinander liegen. Eine Protobuf-Nachricht ist also nur eine Menge von Feldern. Information über diese Menge wie etwa die Größe, ein Bezeichner oder eine ID werden nicht serialisiert.

5.4.2 Das Protobuf Format - Ein Beispiel

Da jetzt alle Bestandteile einer Nachricht eingehend besprochen wurden, soll dies nun an einem Beispiel verdeutlicht werden. Dazu wurde die 'message Status' gewählt, welche ca. alle 200ms im Zuge eines KeepAlive an FU-Remote gesendet werden soll.

```
1  message Message {
2          required int32 robotID = 1 [ default = -1 ];
3          extensions 2 to 99;
```

```
4           extensions 101 to max;
5   }
6
7   message Status {
8           // timestamp of this message
9           optional uint64          timestamp           = 1;
10          // ID of team the robot belongs to
11          optional uint32          teamID              = 2;
12          //...
13          optional uint32          timeOnline          = 3;
14          optional Position        robotPosition       = 7;
15          optional double          robotPositionBelief = 8;
16          optional string          activeAgent         = 12;
17          optional string          activeRole          = 15;
18          optional string          currentState        = 16;
19  }
20
21  extend Message {
22          optional Status          status              = 23;
23  }
```

Abbildung 12: Definition von 'message Message' und 'message Status'

In Abbildung 12 sind die Definitionen von 'message Message' und 'message Status' dargestellt.

Dezimal:	27	90	48	48	48	51	48	0	42	0	0	0	0	0	0	0	0	18	...
ASCII:	*	Z	0	0	0	3	0	*	*	*	*	*	*	*	*	*	*	*	...
Dezimal:	...	8	1	186	1	14	16	1	98	9	65	103	101	110	116	84	101	115	116
ASCII:	...	*	*	*	*	*	*	*	*	*	A	g	e	n	t	T	e	s	t

Abbildung 13: Dezimal- und ASCII-Darstellung einer 'message Status'

In Abbildung 13 ist eine vollständige WLAN-Nachricht sowohl in Dezimalform als auch in ASCII dargestellt. Die * stellen dabei nicht druckbare Zeichen dar. Am Anfang der Nachricht ist der Tag zu sehen, wie er im Kapitel 'Versenden von Nachrichten' beschrieben wurde. Die Nachricht ist also 30 Zeichen lang und wird auf Kanal 0 versendet. Danach folgt '0 42 0 0', welches ein Indikator für eine FUmanoids Nachricht nach altem Format ist. Von den vier möglichen Bytes für die Größe der Protobuf-Nachricht wird nur eines verwendet, in diesem Fall die 18. Im Folgenden ist zu sehen, wie anonym und redundanzarm

die Protobuf-Nachrichten aussehen. Das Erste Datum hat eine 8 als Tag. Hierbei ist die binäre Betrachtung hilfreich:

$8_{10} = 0000'1000_2$,

ergibt Feld Nr. 1 und Typ 0 (Varint). Der Wert ist eine schlichte 1. Es folgt eine 186:

$186_{10} = 1011'1010_2$,

ergibt Feld Nr. 23 und Typ 2 (längenbegrenzt). 23 ist die ID der Nachricht 'message Status' und Typ 2 ein längenbegrenztes Datum. Dies stimmt nicht genau. Es wurde bereits angesprochen, dass das hochwertigste Bit immer anzeigt, dass ein Datum weitergeht. Der Tag eines Datums selbst ist auch als Varint formatiert. Es folgt, dass das nächste Byte auch zum Tag dazugehört. Da hier Little-Endian verwendet wird, muss die 1 für Menschenlesbarkeit nach vorn gesetzt werden. Es folgt:

$442_{10} = 0000'0001'\mathbf{1}011'1010_2$,

wobei die rote **1** nur der Fortsetzungsmarker ist. Wie man sieht, stimmen oben beschriebene Feld Nr. und Typ immer noch. Nach diesem Feld folgt die Feld Nr. 2, die TeamID, als Varint mit dem Wert 1. Danach die Feld Nr. 15, ActiveRole, als längenbegranztes Feld. Dies hat die Länge 9 und da es sich um einen String handelt, ist die ASCII Repräsentation 'AgentTest'. Alle weiteren optionalen Felder der 'message Status' wurden in diesem Beispiel weggelassen.

5.4.3 Generierung von Protobuf Nachrichten

Die Nanopb Bibliothek enthält ein Python-Script 'nanopb_generator.py', welches *.pb Dateien in C-Code übersetzt. Diese *.pb Dateien sind sogenannte 'File Descriptor Sets' und sind vorkompilierte Exemplare der *.proto Nachrichtenbeschreibungen. Sie lassen sich mit dem Programm 'protoc -oFile' erstellen. Protoc ist der Protobuf-Compiler, welcher in jedem Protobuf Bundle enthalten ist.

Sobald erstellt, kann 'nanopb_generator.py' aus diesen Dateien also den C-Code erzeugen. Dabei wird für jedes 'File Descriptor Set' eine Header- und eine Quellcodedatei erzeugt. Jede einzelne Protobuf Nachricht wird dabei in ein Struct übersetzt. Die Felder werden soweit möglich in ihre in C äquivalenten Datentypen übersetzt.
Die Datentypen mit variabler Länge können jedoch nicht direkt übersetzt werden. Mehr dazu im nächsten Kapitel.

5.4.4 Encodierung und Decodierung

Zunächst wird das Verfahren des Encodierens betrachtet. Dabei wird zuerst aus einem Puffer ein Stream erstellt. Am Ende des Encodierungsprozesses steht die encodierte Nachricht

in diesem Puffer. Nach dem Erstellen des Streams wird die Encodierungsfunktion mit dem Stream, dem Nachrichtentyp und der zu kodierenden Nachricht in Form eines Structs aufgerufen. Diese wird dann wie im Kapitel 'Das Protobuf Format' beschrieben das Struct in die Protobuf-Nachricht encodieren. Beim Erstellen und Füllen des Structs mit Daten sind einige Dinge zu beachten, auf die im Folgenden näher eingegangen wird.

Einfache Felder statischer Länge lassen sich direkt zuweisen. Bei optionalen Feldern muss zusätzlich noch eine boolsche Variable gesetzt werden, die dem Variablennamen mit vorangehendem 'has_' entspricht. Auch verschachtelte Nachrichten können direkt zugewiesen werden.

Etwas mehr Aufwand muss bei Datentypen variabler Länge betrieben werden. Beispiele für Typen variabler Länge sind 'String' oder mit 'Repeated' gekennzeichnete Felder, wie erwähnt aber nicht verschachtelte Nachrichten. Für solche Felder wird jeweils ein Struct eingesetzt, welches einen Pointer für jeweils eine Encodierungs- und Decodierungsfunktion enthält und einen weiteren für Argumente an besagte Funktionen. Diese Felder müssen dann manuell en- bzw. decodiert werden. Nach dem Protobuf-Format werden verschachtelte Nachrichten zwar genau wie Strings als längenbegrenzt behandelt, aber Nanopb kennt zur Zeit der Übersetzung dessen Größe, was bei Strings nicht der Fall ist. Die Signatur einer Encodierungsfunktion sieht folgendermaßen aus:

```
bool (*encode)(pb_ostream_t *stream, const pb_field_t *field, void * const *arg);
```

Diese Funktion soll einzelne Felder kodieren, wie im Kapitel 'Das Protobuf Format' erklärt wurde. Zunächst muss mit der Funktion 'pb_encode_tag_for_field' den 'Tag', also den Header, encodieren und dann das Datum selbst encodieren. Zum Encodieren des Datums stehen verschiedene Funktionen bereit, welche jeweils mit 'pb_encode_' beginnen und den Datentyp als Suffix haben. Es wurde im vorherigen Kapiteln auch erwähnt, dass die Reihenfolge von Feldern prinzipiell egal ist. Das bedeutet, ein Callback kann nicht nur ein einzelnes Feld encodieren, sondern mehrere verschiedene. Für wiederholte Felder ist es sehr sinnvoll, jedes Vorkommen des Feldes in einem einzigen Callback zu encodieren. Im Beispielcode wurde etwa zum encodieren der Ergebnisse des 'Region Growing' Algorithmus des HaViMo2 ein einziges Callback verwendet, welches über ein Array aller gefundener Objekte iteriert. Zu beachten ist, dass die Encodierungsfunktionen verschachtelter Nachrichten mehrfach aufgerufen werden und deshalb immer im Kontext einer Nachricht das selbe Ergebnis erzeugen sollten. Ansonsten wird der Encodierungsvorgang mit dem Fehler 'submsg size changed' abgebrochen.
Im Quellcode sind in der Datei 'debug.c' einige weitere Beispiele zu diesem Thema zu finden.

Es besteht außerdem die Möglichkeit, den Feldern variabler Länge eine statische Länge zuzuweisen. Dies würde beispielsweise bei Strings bedeuten, dass diese in char Arrays konstanter Länge übersetzt werden. Dies ist eine spezielle Notation, welche der Nanopb Compiler akzeptiert. Auf die tatsächlich encodierte Nachricht hat dies keinen Einfluss, was bedeutet, dass andere Kommunikationsteilnehmer, welche beispielsweise eine Java-Implementierung von

Protobuf verwenden, wissen müssen, dass sie sich an etwaige Längenvorgaben halten müssen. Der interessierte Leser findet mehr dazu in der Nanopb Dokumentation[13] unter 'Data types'.

Es ist Nanopb außerdem nicht möglich, 'Extenions' automatisch zu decodieren. Der wesentliche Unterschied zu Submessages ist, das Submessages zu dem Zeitpunkt, zu dem die eigentliche Nachricht in Quellcode übersetzt wird, bereits bekannt ist. Diese wird also als Struct im Struct übersetzt. Bei einer Extension ist dies nicht der Fall. An Stelle der Extension wird ein standard Struct eingesetzt. Dieses enthält einen Pointer auf den Typ der Extension, einen Pointer auf den Speicher, wohin die Extension decodiert werden soll und einen Pointer auf das nächste Extension-Struct. Letzterer wird benötigt, falls die Protobuf-Nachricht mehrere Extensions beinhaltet. Diese Pointer zeigen zunächst auf NULL. In diesem Fall überspringt der Decoder das Feld. Es müssen nun manuell Werte eingetragen werden. Mehr dazu im nächsten Kapitel.

5.5 Verarbeiten von eingehenden Nachrichten

Im Kapitel 'Das Frontend' wurde bereits das Verarbeiten eingehender Nachrichten angesprochen. Da keine dynamische Speicherverwaltung vorhanden ist, wurde hier auf eine Lösung zurückgegriffen, welche sich lediglich des Funktionsstacks bedient. Die Funktion 'processData' wird also mit einem Pointer auf die Nachricht und der Größe des Datenpuffers aufgerufen. Der WLAN-Header kann in der Regel übersprungen werden, da er für die Verarbeitung keine Relevanz hat. Er gibt je nach Verbindungstyp nur Aufschluss über Verbindungsnummer und IP-Adresse des Absenders. Die Nachrichten des FUmanoid-Protokolls enthalten selbst bereits eine Information in Form einer ID, von wem die Nachricht stammt. Dies macht den Informationsgehalt des WLAN-Headers überflüssig.

Der nächste Schritt, das Erkennen der Extension, ist ein wenig aufwändiger. Im vorherigen Kapitel wurde beschrieben, wie Nanopb automatisch die einzelnen Felder der Reihe nach kodiert, aber dass es keine Extensions automatisch decodieren kann. Dies bedarf manueller Unterstützung, indem in dem Nanopb-Stream das Extension-Feld der 'message Message' herausgesucht und dessen Tag ausgelesen wird. Anhand des Tags kann dann in einem globalen Array anhand der Feld Nr. nachgeschlagen werden, um welche Extension es sich handelt und dementsprechend ein Exemplar des dazugehörigen Structs erstellt werden. Diese werden auf dem Stack erstellt. Der Tag ist, wie in vorherigen Kapiteln erwähnt, identisch mit dem in der *.proto Datei vergebenen ID.

Nachdem die Extension gesetzt wurde, müssen noch weitere Decodierfunktionen gesetzt werden. Also Funktionen, welche die Felder dynamischer Länge decodieren. Diese werden, wie im Kapitel Encodierung und Decodierung bereits vorgestellt, auch als Funktionspointer auf Callbacks angegeben. Auch hier ließe sich, falls die Anzahl der Funktionen größer wird, ein Verzeichnis anlegen. Nachdem alle Felder gesetzt wurden, kann die Nachricht decodiert

und das Callback zum Verarbeiten der Nachricht aufgerufen werden.

5.6 Erstellen neuer Nachrichten

In diesem Kapitel wird vorgestellt, wie dem FUmanoids-Protokoll eine Nachricht hinzugefügt werden kann und wie diese zur Kommunikation z.B. zwischen zwei Microcontrollern verwendet werden kann. Einige der hier besprochen Schritte wurden bereits in vorherigen Kapiteln besprochen, so dass deren Hintergrund nicht nochmals besprochen werden braucht.

Zunächst muss die Nachricht als *.proto Datei definiert werden. Die neu erstellte Nachricht sollte eine Extension der Nachricht 'Message' sein und eine entsprechende ID tragen. Bei der ID ist zu beachten, dass diese nicht mit anderen Nachrichten kollidiert. Darum sollte für neue Projekte ggf. ein Wertebereich an ID's für neue Protobuf-Nachrichten reserviert werden.
Danach muss die *.proto Datei analog zum Kapitel 'Generierung von Protobuf Nachrichten' in Quellcode übersetzt werden. Damit die Nachricht empfangen werden kann, muss die Nachricht in das Nachrichtenverzeichnis eingetragen werden. Dieses Verzeichnis ist in der Software als einfaches Array gegeben. In dieses Array wird auch der Callback eingetragen, der die Nachricht später verarbeitet. Dieser muss ebenfalls noch erstellt werden.

Sollte die Nachricht Datenfelder dynamischer Länge enthalten, müssen für diese Felder ebenfalls Callbacks zum Decodieren erstellt werden und diese Decodierfunktionen ebenfalls der Nachrichtenverarbeitung mittels der 'registerDecodingFunctions' Funktion bekannt gemacht werden. Da diese Datenfelder mangels dynamischer Speicherverwaltung nur mit Hilfe globaler Variablen an das verarbeitende Callback weitergegeben werden können, empfiehlt es sich, den Inhalt der Datenfelder direkt in der Decodierfunktion zu verarbeiten.

Zum Senden der Nachricht muss nichts weiter erstellt werden. Beispiele, wie Nachrichten erstellt und versendet werden können, sind in der Datei 'Debug.c' zu finden.

5.7 Erstellen neuer Debug-Optionen

Wie bereits vorgestellt wurde, wird in FU-Remote eine Liste an Debug-Optionen für jeden Roboter individuell angezeigt. Auf Knopfdruck können damit bestimmte Optionen aktiviert oder deaktiviert werden. FU-Remote fordert diese Liste vom jeweiligen Roboter an, wenn es ihn zum ersten Mal sieht und wird die Anfrage immer wiederholen, bis eine Liste erhalten wurde. Die Liste wird komplett in einer Nachricht versendet. Wird eine zweite solche Liste geschickt, betrachtet FU-Remote diese als Update und verwirft die alte. Sollte der Roboter offline gegangen sein und FU-Remote nach einem Neustart erneut kontaktieren, so wird FU-Remote die vorher bereits aktivierten Debug-Optionen automatisch erneut setzen. Das bedeutet für den Anwender des Microcontrollers, dass es möglichst kurze Namen und Beschreibungstexte für die Debug-Optionen verwenden, oder ganz auf Beschreibungen verzichten sollte, da die Nachrichten dürfen maximal 1400 Byte lang sein.

In der Datei 'Debug.c' gibt es eine 'optionList', in welche alle Debug-Optionen eingetragen werden. Es existiert bereits ein Callback, welcher das Aktivieren und Deaktivieren automatisch handhabt. Im Beispielcode wird diese Liste linear durchsucht, welche Debug-Optionen FU-Remote aktiviert hat und dementsprechend die angeforderten Daten verschickt.

6 Sichere Datenübertragung

In den vorherigen Kapitel wurden bereits alle Algorithmen und die dazugehörige Software zum Versenden und Empfangen von Paketen über das WLAN-Modul vorgestellt. Dabei wurde ein Verlust von Nachrichten in Kauf genommen, da das Fumanoids-Protokoll mit verloren gegangenen Nachrichten umgehen kann. Sollte eine sichere Datenübertragung, beispielsweise für eine drahtlose Kalibrierung des HaViMo2, notwendig sein, reichen die bislang vorgestellten Mechanismen nicht. In diesem Kapitel wird zunächst aufgezeigt, dass selbst eine Übetragung über TCP nicht sicher ist. Dann wird demonstriert, wie eine Übertragung ohne Nachrichtenverluste unter Einbehaltung der Nachrichtenreihenfolge realisiert werden kann. Zu diesem Zweck wird hier ein einfaches Protokoll implementiert und demonstriert.

6.1 Zweck des Protokolls

Bisher wurde verschwiegen, dass es bei ausreichend hoher Belastung des Netzwerks unvermeidbar ist, dass Daten verloren gehen können.

Messungen haben ergeben, dass WLAN-Nachrichten manchmal mit unvollständigem WLAN-Empfangsheader über die serielle Schnittstelle gelesen werden. Dies ist nicht auf fehlerhaftes Parsen der eingehenden Daten zurückzuführen. Es ist zu vermuten, dass der Puffer des WLAN-Moduls ein Ringpuffer ist, welcher überläuft. Ein Indikator dafür ist, dass immer nur der Anfang der Nachricht fehlt. Weiterhin ist zu vermuten, dass eingehende und ausgehende serielle Daten sich im WLAN-Modul einen Puffer teilen. Denn je mehr und je schneller Daten versendet und empfangen werden, desto mehr defekte WLAN-Nachrichten sind in der Ausgabe.

Dies liegt daran, dass das WLAN-Modul, je nach Implementierung, nur über einen 1400 Byte großen Puffer verfügt. Für Anwendungssoftware und WLAN-Funktionen des Moduls agieren zwei Prozessoren.[33] Ein handelsüblicher PC schafft es in jedem Fall, mehrere MB/s per WLAN zu übertragen. Das WLAN-Modul unterstützt zwar die Kommunikation über WLAN, verfügt aber nicht über genug Kapazitäten, um eine so große Datenmenge zu speichern. Die serielle Schnittstelle zum Lesen des Puffers ist nicht schnell genug, um den Puffer entsprechend schnell zu leeren, ganz unabhängig davon, ob die Firmware des WLAN-Moduls diese große Menge eingehender Daten überhaupt verarbeiten kann.

Im Fall von UDP geht das mit einem Verlust von Paketen und teilweise überschriebenen Paketen im Puffer einher. Im Falle von TCP kann nur der Puffer überlaufen. Um sicherzustellen, dass diese Annahmen stimmen, wurden sie in einer Beispielanwendung getestet.

Die Beispielanwendung ist in Java geschrieben und startet einen TCP-Server. Sobald der Roboter sich verbindet, sendet der Server 100 Pakete von 1 kB Größe mit willkürlichem Inhalt im Abstand von 50 ms. Nur das erste Byte ist nicht willkürlich. Dieses wird verwendet,

[33] Spezifikation siehe 'GS1500M, 802.11 b/g/n, Low-Power Wi-Fi Module, Data Sheet'[19] Seite 7 und 18

um die Pakete zu nummerieren. Die größte Paketnummer ist also 99. Um zu testen, ob Pakete verloren gehen, soll der CM-530 nun alle Pakete empfangen und die Summe der ersten Bytes berechnen. Diese sollte wie folgt lauten:

$$\sum_{i=1}^{100}(i-1) = 4950 \tag{1}$$

Das Ergebnis, welches der CM-530 berechnet hat, war wie erwartet inkorrekt. In mehreren Testläufen lag die Summe jeweils zwischen ca. 900 und 1000. Auch bei einem zeitlichen Abstand von 500 ms zwischen den Paketen wurde das exakte Ergebnis selten erreicht.

Sollte auf dem CM-530 noch zusätzlich ein Scheduler installiert werden und weitere Tasks laufen, oder werden die Netzwerkfunktionen aus einem anderen Grund sehr langsam aufgerufen, so erhöht sich die Anzahl der verlorenen Pakete. Dieses Problem ist allerdings nicht für das GS1500M modulspezifisch, andere Module haben das gleiche Problem.

In jedem Fall ist nicht garantiert, dass alle Pakete ankommen, oder im Fall von UDP ihre ursprüngliche Reihenfolge behalten. Bei den Debug Daten, welche an FU-Remote gesendet werden, oder den GameController-Nachrichten ist dies nicht wichtig. Bei den Kamerabildern ist es beispielsweise nicht wichtig, wie der Frame vor mehreren Sekunden aussah. Nur der Aktuellste zählt. Dementsprechend ist ein erneutes Anfordern bei Paketverlusten nicht sinnvoll. Selbiges gilt bzgl. des aktuellen Spielstatus, welcher vom GameController gesendet wird. Für andere, zukünftige Anwendungen ist ein sicheres Versenden aber sinnvoll und wichtig, wie beispielsweise für die Kalibrierung der HaViMo2 Kamera über WLAN, würde das WLAN-Modul einfach als Brücke verwendet werden. Bei fehlenden Paketen oder Paketen in falscher Reihenfolge würde eine solche Kalibrierung fehlschlagen.

Um eine sichere Übertragung zu gewährleisten, muss also ein TCP-ähnliches Protokoll implementiert werden. Dies wurde im Rahmen dieser Arbeit auch getan, zum einen um zu zeigen, dass es realisierbar ist und zum anderen, um die Möglichkeiten und Fähigkeiten des Frameworks zu demonstrieren.

Bei richtiger Nutzung wird das im Folgenden vorgestellte Protokoll eine sehr hohe Zuverlässigkeit haben. Eine hundertprozentige Sicherheit ist aber nicht gewährleistet. Dies wird im Kapitel Einschränkungen eingehend besprochen.

6.2 Beispiel eines Alternierenden-Bit Protokolls

Wie im vorherigen Abschnitt wurde bereits die Notwendigkeit eines sicheren Protokolls erläutert. Für diese Arbeit wurde ein kleines Protokoll implementiert, nämlich ein Derivat des 'Alternierenden Bit' Protokolls.[34] Dieses ist im Verzeichnis 'network' in der Datei

[34] Nach 'Computernetze, Ein Top-Down-Ansatz'[24]

'secure_stream.c' zu finden. Dieses Protokoll wurde aufgrund seiner Einfachheit und geringer notwendiger Puffergrößen gewählt.

Beim Alternierenden-Bit Protokoll enthält jedes Paket ein zusätzliches Bit. Dieses Bit ist die Paketnummer. Ein Kommunikationsteilnehmer sendet ein Paket mit diesem Bit auf 0 gesetzt und wartet darauf, dass der andere Teilnehmer dies mit einem ACK bestätigt. Sobald das ACK erhalten wurde, kann das nächste Paket mit dem Bit auf 1 gesendet werden. Hier wird genau so verfahren. Sobald das ACK für das Paket mit Bit 1 erhalten wurde, kann wieder mit 0 begonnen werden. Sollte ein ACK nach angemessener Zeit nicht erhalten worden sein, wird das Paket als verloren angenommen und erneut gesendet. Diese Form der Absicherung der Kommunikation funktioniert offensichtlich bidirektional.

Das hier implementierte Derivat unterscheidet sich in zwei wesentlichen Aspekten. Anstatt nur ein Bit zu verwenden, wird ein ganzes Byte verwendet. Zum einen erleichtert es die Implementierung des Protokolls in höheren Programmiersprachen und zum anderen verleiht es dem Protokoll gewisse Header-Eigenschaften, auf die später noch eingegangen wird. Der zweite Unterschied ist, dass dieses Byte nicht zwischen 1 und 0 alterniert, sondern wie ein Ringpuffer bis zu einer gewissen Zahl hoch zählt und dann wieder bon vorne beginnt. Das erste Byte eines Pakets ist diese Sequenznummer, das zweite ist ein ACK für das zuletzt vom Kommunikationspartner erhaltene Paket. Reine ACK-Nachrichten unterscheiden sich von Nachrichten mit Nutzlast nur darin, dass sie keine Nutzlast haben. Somit kann jede Nachricht mit Nutzlast gleichzeitig ein ACK sein. Dies wird auch 'Piggibacking' genannt.[35]

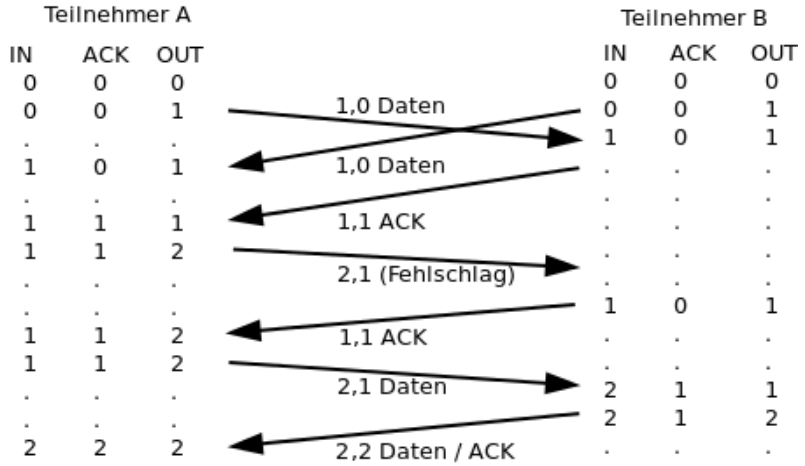

Abbildung 14: Beispielsequenz des Derivats des Alternierenden-Bit Protokolls

In Abbildung 14 ist eine Beispielkommunikation abgebildet, welche alle Spezialfälle, bis auf den Überlauf abdeckt. 'IN' ist hierbei der Zähler für eingehende Daten und 'OUT' der Zähler für ausgehende Daten. 'ACK' ist die letzte ausgehende Nachricht, welche bestätigt wurde. Dementsprechend kann OUT n erst gesendet werden, wenn ACK (n-1) ist. Bei Fehlverhalten sollten solche Nachrichten ignoriert werden. Eine Zeile mit drei Punkten

[35] Vgl. 'Computernetze, Ein Top-Down-Ansatz'[24], Seite 280

bedeutet keine Änderung zum vorherigen Zustand. In den ersten beiden Nachrichten senden Teilnehmer A und B vom logischen Ablauf her gleichzeitig eine Nachricht. Teilnehmer A versäumt es, sofort ein ACK zu schicken, sondern sendet es mit seiner nächsten Nachricht. Dies ist generell Protokoll-konform, aber ohne explizite ACK's kann es bei verlorenen Nachrichten zu Verklemmungen kommen. Teilnehmer B sendet direkt ein ACK, auf dessen Erhalt hin Teilnehmer A seine zweite Nachricht schickt. Diese sollte auch die erste Nachricht von Teilnehmer B bestätigen, geht aber verloren. Bei Teilnehmer B läuft dementsprechend ein Timeout ab und er sendet sein erstes ACK erneut. Dieses kommt wiederum an und Teilnehmer A bemerkt in diesem Moment, dass seine vorherige Nachricht nicht ankam und sendet sie erneut. Nach Erhalt kann Teilnehmer B dann seine zweite Nachricht senden.

Bei stagnierender Kommunikation, also wenn beide Teilnehmer keine Daten zu senden haben, ist es sinnvoll in regelmäßigen Abständen ACK's zu senden. Zum einen als KeepAlive und zum anderen, falls eine Nachricht verloren gegangen sein sollte.

Wenn Nachrichten erst einmal vom WLAN-Modul empfangen wurden, muss die Anwendersoftware entscheiden, was für ein Paket dies ist. Sollte in der Kommunikation nur dieses Protokoll verwendet werden, so dass man sicher gehen kann, dass jede Nachricht diesem Protokoll entspricht, wäre ein solches Parsen unerheblich. Sollen etwa FU-Remote-Nachrichten, GameController-Nachrichten und Nachrichten dieses Protokolls gemischt werden, so sollten diese Nachrichten anhand eines Headers klar unterscheidbar sein.

FU-Remote Header beginnen immer mit den Bytes Dezimal 0 und darauf folgend Dezimal 42. GameController Nachrichten beginnen mit ASCII 'RG'.[36] Aus diesem Grund ist es sinnvoll, die Sequenznummern bis maximal 41 umlaufen zu lassen. Dadurch ist die Nachricht immer klar unterscheidbar. Im Beispielcode wurden nur Sequenznummern von 0 bis 20 verwendet. Die Sequenznummern als Bytes wurden anstatt einzelner Bits aus zwei Gründen gewählt. Erstens wird ohnehin mindestens ein ganzes Byte Aufgrund des Alignments benötigt, zwei Bytes als Header zu betrachten macht die Identifizierung allerdings sicherer. Zweitens kann dieses Protokoll dadurch relativ einfach durch Puffer und selektive ACK's auf ein 'Selective-Repeat' Protokoll erweitert werden. Dieses bietet sich eher an, da bei einer hohen Verlustrate der Datendurchsatz höher ist, als z.B. bei dem 'Go-Back-N' Protokoll. Der interessierte Leser findet in 'Computernetze, Ein Top-Down-Ansatz'[24] auf Seite 261 und 265 einige Informationen und Details zu diesen Protokollen.

6.3 Test des Beispielprotokolls

Im vorherigen Kapitel wurde die Implementierung des Protokolls erwähnt. Eine Java-Implementierung mit zwei Testfällen liegt ebenfalls bei. Auf einen formellen Beweis der Korrektheit sei hier verzichtet, da es sich um ein Derivat eines Protokolls nach Literaturvorgaben

[36] Vgl. Quellcode aus 'RoboCup GameController: RoboCupGameControlData.h'[36]

handelt. Die Illustration des vorherigen Kapitels stellt ausreichend dar, dass die Korrektheit des Protokolls nicht beeinträchtigt wird.

Der erste Testfall testet eine unidirektionale Kommunikation und ist mit 'Lorem Ipsum Test' benannt. Ein Kommunikationsteilnehmer, in diesem Fall die JAVA-Anwendung, sendet einen Text, welcher vom CM-530 ohne Verlust und in richtiger Reihenfolge ausgegeben werden soll. Der CM-530 muss in diesem Fall nur Nachrichten Empfangen und ACK's senden.

Der zweite Testfall testet die bidirektionale Kommunikation und ist mit 'Mathe Test' benannt. Hier senden beide Kommunikationsteilnehmer gleichzeitig Informationen. Teilnehmer A beginnt mit einer Zufallszahl von 1 bis 7 und addiert sie auf sein Eigenergebnis. Dann sendet er diese Zahl an Teilnehmer B, welcher diese Zahl auf sein Fremdergebnis addiert, ebenfalls bei 0 beginnend. Die zweite Zahl wird zum Eigenergebnis von Teilnehmer A multipliziert und versendet. Teilnehmer B multipliziert diese Zahl ebenfalls mit seinem Fremdergebnis. Dies läuft bis zu einer vorher festgelegten Menge an Zahlen. Dieses Verfahren wird parallel betrieben, so dass das Eigenergebnis mit dem Fremdergebnis des jeweils anderen identisch sein sollte. Am Ende wird das Eigenergebnis an den jeweils anderen versendet, um das Ergebnis zu überprüfen. Zusätzlich wird die Anzahl der Operanden ausgegeben.

Dieser Test eignet sich aus mehreren Gründen besonders gut. Der Testumfang ist leicht skalierbar, der Test testet bidirektionale Kommunikation, durch Grundrechenregeln wird geprüft, dass Pakete in der richtigen Reihenfolge empfangen werden und das Fehlen von Nachrichten wird ebenfalls entdeckt.

Es ist natürlich möglich, dass trotz fehlerhafter Übertragung das Resultat des Tests stimmt, aber dennoch Werte fehlen. Dies ist unerheblich, da in diesem Fall die Anzahl der Operanden nicht korrekt ist.

7 Footprint

In diesem Kapitel werden verschiedene Aspekte des Footprints der Netzwerkfunktionen betrachtet. Dabei wurden durch verschiedene Messverfahren Anforderungen und Verbrauch von Hardwareressourcen und Zeit festgestellt. Diese Messergebnisse bilden die Grundlage für Gedanken zur Optimierung der Kommunikationsprozesse und für Erwägung von Nutzbarkeit in bestimmten Anwendungsfällen.

7.1 Messverfahren und Testbedingungen

Der relevante Netzwerkcode wurde auf drei Metriken hin überprüft. Zum Ersten der Speicherverbrauch auf dem Stack, zum Zweiten die Programmlaufzeit und zum Dritten auf den Traffic.

Die Laufzeit wurde mit Hilfe einer zu diesem Zweck erstellten Timerfunktion gemessen. Die Genauigkeit des Timers ist dabei architekturbedingt bis auf $\frac{1}{100000}$Sekunde genau. Bedingt durch das Auftreten von Interrupts und dem Overhead der Messung selbst, unterliegt das Messergebnis einer minimalen Ungenauigkeit. Da das System Single-Threaded ist, können andere Faktoren die Laufzeit nicht beeinflussen.

Die Messung der Nutzung des Stacks geschieht in Anlehnung an 'Stack Bounds Analysis for Microcontroller Assembly Code'[15] mittels einem für Embedded-Systeme typischen Verfahren. Vor dem zu messenden Aufruf wird eine große Menge Speicher auf dem Stack mit einem Bitmuster belegt. Dieses Bitmuster sollte mit möglichst geringer Wahrscheinlichkeit zufällig vorkommen. Deshalb eignen sich etwa die Werte Dezimal '0' oder Dezimal '1' für ein Byte Speicher nicht. Die Software 'IAR IDE' verwendet für solche Analysen eine Folge von Bytes mit dem Dezimalwert '205'.[37] Dieser wurde auch für die hier vorgestellten Messungen verwendet. Nachdem der Speicher initialisiert wurde, wird die zu messende Funktion aufgerufen. Diese Funktion und all ihre Unterfunktionen werden im Laufe ihrer Lebenszeit das Bitmuster mit ihren eigenen Daten überschreiben. Im letzten Schritt wird, nachdem die zu messende Funktion abgearbeitet wurde, der Stack solange durchlaufen, bis das zuvor geschriebene Bitmuster wieder auftaucht. In diesem Falle wird solange gesucht, bis der Dezimalwert '205' vier Mal hintereinander im Stack steht, denn ein viermaliges sequentielles Auftreten sollte bei einer gleichverteilten Wahrscheinlichkeit über alle Werte nur mit einer Wahrscheinlichkeit von 1 zu 256^4 auftreten. Dies kann als unwahrscheinlich genug betrachtet werden.

Dieses Verfahren nutzt aus, dass das System keinen Scheduler betreibt, welcher das Messverfahren stören könnte. Sowohl der Scheduler als auch der neu aktiv geschaltete Prozess würden den Stack für ihre Variablen nutzen und das Messergebnis verfälschen.

Der Traffic wurde mit Hilfe des Tools 'NTop' gemessen.[38] Dieses Tool ermöglicht eine sehr differenzierte Auswertung nach Hosts, Protokoll, Paketgrößen, usw..

[37] Siehe Dokumentation 'Controlling the stack size'[27], Seite 5
[38] Version ntop v.4.1.0 (64 bit), aktuelle Version unter 'ntopng'[17]

Die ersten beiden Messverfahren wurden gewählt, da sie leicht zu implementieren sind und dank der Single-Threaded Eigenschaft des Systems trotzdem eine sehr hohe Genauigkeit besitzen. Die Baudrate der seriellen Schnittstelle zwischen CM-530 und WLAN-Modul beträgt bei allen Tests 460800.

7.2 Analyse der Stacknutzung

Nachrichtentyp	Minimum	Maximum	Durchschnitt
demo.field	2904	2912	2907.12
demo.image	2912	2928	2918.96
demo.plotter	2152	2192	2173.2
demo.relative	2912	2920	2917.04
demo.table	2136	2144	2136.96
havimo.region	2936	2984	2950
havimo.grid	2944	2976	2946.6
demo.text[39]	2104	2136	2121.52
demo.text max[40]	3520	3560	3546.8

Tabelle 2: Stacknutzung der Debug-Funktion, jeweils in Byte

In Tabelle 2 ist die Stacknutzung der einzelnen Debug-Optionen aufgelistet. Die minimale Stacknutzung liegt bei ca. 1900 Byte. Das liegt daran, dass die Encodierungsfunktion nicht weiß, wie groß die Protobuf-Nachricht sein wird. Deshalb erstellt sie direkt ein Array von 1400 Byte Größe. Dies kann als globale Variable ausgelagert oder durch dynamische Speicherverwaltung reduziert werden. Weil die Größe bei Verwendung einer dynamischen Speicherverwaltung nicht bekannt sein muss, würde daraus eine minimale Größe von ca. 500 Byte + Minimale Nachrichtengröße' folgen. Es wäre auch möglich, die Daten zwei Mal zu encodieren. Beim ersten Mal würden nur die Bytes gezählt werden, so dass ein entsprechend großes Array erzeugt werden kann. Dies würde aber wiederum zu einer erhöhten Rechenzeit führen.

Nachrichtentyp	Minimum	Maximum	Durchschnitt
Empfangen von WLAN Daten, 'recvWlanData'	72	152	98.37
Senden des KeepAlive	1912	1936	1920.5

Tabelle 3: Stacknutzung anderer Funktionen, jeweils in Byte

Das Empfangen von Daten benötigt vergleichsweise wenig Ressourcen. Ein Grund dafür ist, dass die Empfangsfunktion durch das Pollen immer nur sehr kleine Arbeitsschritte erledigt. Das Ergebnis, also z.B. eine empfangene WLAN-Nachricht, wird in

[39] 65 Zeichen langer Beispieltext
[40] 1365 Zeichen langer Beispieltext (Maximale Nachrichtenlänge)

einem vom Benutzer übergebenen Speicherbereich akkumuliert. Hierfür wären wiederum ca. 1439 Byte hinzuzurechnen. 1400 Byte für die Nachricht und bis zu 39 Byte für den WLAN-Empfangsheader.

7.3 Analyse der Rechenzeit

Zunächst wird das Empfangen von Daten über die serielle Schnittstelle betrachtet. Beim lesen von Bytes von der seriellen Schnittstelle kommt es im Wesentlichen darauf an, was für Daten gelesen werden. Statusmeldungen vom WLAN-Modul müssen geparst werden, da sie wichtige Informationen, wie etwa Verbindungsaufbau oder Abbruch beinhalten können. Da dies über einfaches Stringparsing geschieht und eine serielle Ausgabe nach PCU geschieht, ist es vergleichsweise langsam. Das Empfangen von WLAN-Nachrichten hingegen geht um einiges schneller, da hier nur einige wenige Headerinformationen geparst werden müssen.

Funktion	Minimum	Maximum	Durchschnitt
Empfangen (Statusmeldungen)	0.20	24.12	-
Empfangen (WLAN-Nachrichten)	0.02	0.19	0.05
Empfangen (Nichts bzw. OK)	0	0.03	0.01
Verarbeiten (Vorbereiten)	0.03	0.04	0.04
Verarbeiten (Decodieren alg.)	0.06	10.11	4.16
Verarbeiten (Decodieren klein)	0.06	0.07	0.06
Verarbeiten (Decodieren groß)	3.36	10.11	6.8

Tabelle 4: Dauer verschiedener Netzwerkfunktionen, jeweils in Millisekunden

Tabelle 4 ist zu entnehmen, dass das Pollen von WLAN-Nachrichten und OK-Meldungen einen vernachlässigbar geringen Zeitaufwand von weniger als $\frac{1}{5}$ Millisekunde haben. Das Pollen von Statusmeldungen hat mit etwas mehr als 24 Millisekunden einen nennenswerten Zeitaufwand. Da das Stringparsing selbst kaum optimierbar ist, könnte zur Reduzierung nur die Anzahl der Erkennbaren Schlüsselwörter reduziert werden. Ggf. ließe sich hier auch eine harte Obergrenze durch Zeitmessung einbauen.

Zum Zweck der differenzierten Betrachtung wurde das Verarbeiten von Nachrichten in mehrere Teilaufgaben zerlegt. Das 'Vorbereiten' beinhaltet alle Arbeitsschritte vor dem Parsen, also z.B. erstellen und initiieren der Datenstrukturen, Auslesen des Extension Type usw.. Diese Arbeitsschritte laufen in vernachlässigbarer Zeit ab. Das Parsen ist stark abhängig von der Empfangenen Nachricht. Einfache Anfragen nach Daten enthalten oft nur einen Integer und werden dementsprechend sehr schnell verarbeitet. Diese sind in der Tabelle mit 'klein' bezeichnet. Größere Nachrichten, welche Daten variabler Länge enthalten, werden entsprechend langsamer decodiert und in der Tabelle mit 'groß' gekennzeichnet. Dies liegt hauptsächlich an den Decodierungs-Callbacks. Die Callbacks zu den Funktionen wurden nicht betrachtet. Im Beispielcode beträgt die durchschnittliche Dauer etwa 5ms bis

10ms aufgrund von Ausgaben nach PCU. Die Laufzeit im Anwendungsfall hängt also vom Anwendercode ab.

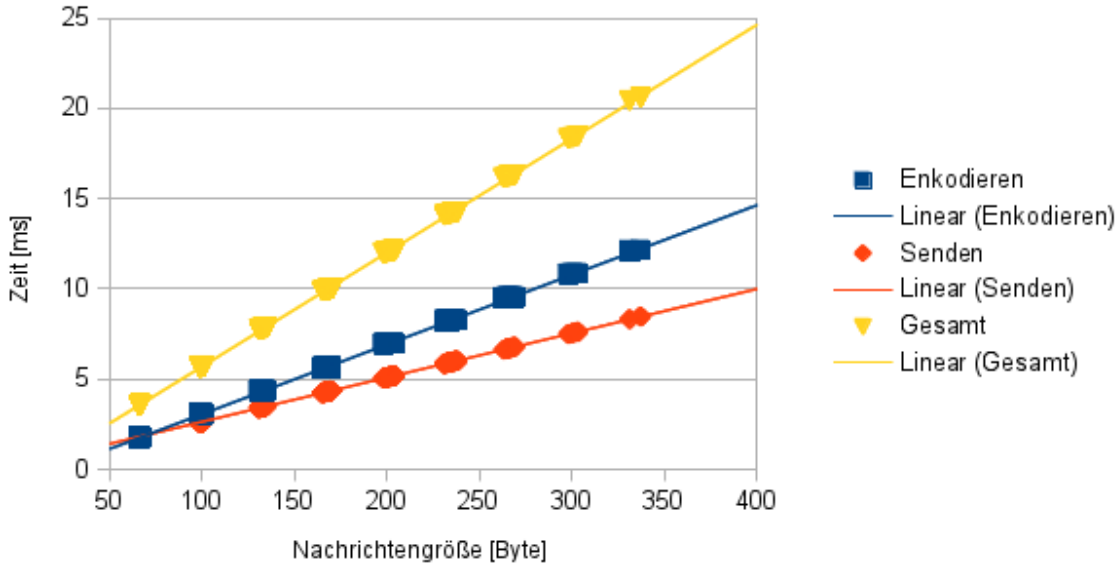

Abbildung 15: Zeitaufwand zum Encodieren und Versenden von HaViMo2-Daten

In Abbildung 15 ist der benötigte Zeitaufwand zum Encodieren und Versenden am Beispiel von HaViMo2 Region-Growing-Daten dargestellt. Dabei ist die benötigte Zeit, um das Kamerabild anzufordern und zu speichern, nicht mit eingerechnet. Dies kann laut Dokumentation bis zu 60 ms dauern. Die Encodierungslinie enthält den Aufwand, der für den gesamten Encodierungsprozess benötigt wird. Die Linie 'Senden' stellt den Zeitaufwand für das Senden der Daten über die serielle Schnittstelle dar. Damit das WLAN-Modul die eingegangen Daten verarbeiten und den Puffer leeren kann, ist eine zusätzliche Wartezeit von 2 bis 5 Millisekunden empfehlenswert.

Nach Betrachtung des Protobuf Formats ist es nicht weiter verwunderlich, dass der Zeitaufwand zur Encodierung linear ist. Selbiges gilt für das Versenden von Daten über die serielle Schnittstelle. Durchaus relevant ist allerdings der Zeitaufwand, der sich mit Hilfe der aus den Messdaten resultierenden Geradengleichungen berechnen lässt.

Dauer zum Encodieren: 0.0386x-0.8057

Dauer zum Versenden: 0.0245x+0.184

Gesamtdauer (encodieren und versenden): 0.0632x-0.6217

Daraus ergibt sich, dass die maximal zu erwartende Dauer, nämlich bei einer Nachrichtengrö-

ße von 1400 Byte, zum Versenden von Nachrichten insgesamt ca. 88 ms beträgt. Theoretisch lassen sich größere Nachrichten erstellen und erfolglos versenden, aber sollte eine Nachricht den Nanopb vorgegebenen Puffer übersteigen, so wird das Versenden mit einer Fehlermeldung abbrechen. Darum gilt für Nachrichten größer als 1400 Byte folgende Gleichung.

0.0386*1400-0.8057 ≈ 53 ms

Die Rate, mit der Nachrichten versendet werden, liegt deutlich unter der Baudrate, welche hierfür die obere Grenze ist. Der Messpunkt war hier die Funktion 'sendWlanDataToCon'. Setzen wir die Baudrate in die entsprechende Formel ein, so erhalten wir ca. das 11-Fache der Idealdauer.

0.0245*460800+0.184 = 11289.784

Dies spricht für einen Overhead auf Implementierungsseite und einen hohen Verlust bei der Übertragung aufgrund der hohen Baudrate. Des weiterem sind die WLAN-Header nicht im Netzwerkverkehr eingerechnet, müssen aber seriell übertragen werden. Außerdem werden etwaige Interrupts ebenfalls eingerechnet.

7.4 Analyse des Traffics

Wie im Kapitel 'Regel der Kommunikation' bereits erwähnt wurde, gibt es laut Regelwerk des RobotCups eine Obergrenze für den Traffic. Dieser beträgt 1 Mb/s bzw. 125 kB/s pro Team. Da nicht genau bekannt ist, wie schnell seriell ausgehende Daten tatsächlich in Pakete umgesetzt werden können, wurde dieses mit Hilfe des Tools 'NTop' über kurze und mittelfristige Zeiträume ausgemessen.

Nutzungsfall	Stichprobe	Ø1 Minute	Ø5 Minuten	Peak
Nur KeepAlives	2.5	2.5	2.5	2.6
Simulierte Last, HaViMo2	13.4	12.8	12.7	14.3
Vollast, Protobuf	36.2	42.8	41.8	44.4
Vollast, Rohdaten	221.3	298.3	306.4	319.8

Tabelle 5: Verursachter Traffic in verschiedenen Anwendungsfällen, jeweils in Kbit/s

Wie in Tabelle 5 dargestellt, wurde der verursachte Traffic in vier verschiedenen Anwendungsfällen gemessen. Dabei wurde jeweils eine Stichprobe, der Durchschnitt über eine bzw. fünf Minuten und der Spitzentraffic betrachtet. Zunächst wurde der Traffic von KeepAlives gemessen. Dies sind 24 Byte große WLAN Nachrichten, welche ca. alle 200 ms geschickt werden. Der Messwert entspricht in etwa dem rechnerisch erwarteten Wert. Dies soll am Beispiel kurz durch gerechnet werden. Der IP Header hat eine Größe von 6 * 32

Bit.[41] Der UDP Header kommt mit 64 Bit hinzu.[42] Es folgt:

(24*8 Bit + 64 Bit + 6*32 Bit) * 5 /s = 2240 Bit/s

Die restlichen 250 Bit/s entfallen auf Messtoleranz.

Bei 'Simulierte Last' wurde ein normales Nutzungsverhalten simuliert. Zu diesem Zweck wurden nur HaViMo2 Debug-Daten verschiedenen Umfangs versendet. Die Paketgrößen lagen im folgenden Bereich.

- 64 < Größe <= 128 bytes, Anteil 74.4%, 3,249 Pakete
- 128 < Größe <= 256 bytes, Anteil 24.7%, 1,078 Pakete
- 256 < Größe <= 512 bytes, Anteil 1.0%, 42 Pakete

Diese wurden im 100 ms Takt verschickt. Hinzu kommen die KeepAlives. Im darauf folgenden Test wurden pausenlos Protobuf-Nachrichten encodiert und versendet. Die Nachrichten waren Text-Debug-Nachrichten von ca. 250 Byte Länge. Dieser Test stellt den maximalen Durchsatz bei der Kommunikation mit FU-Remote dar. Der Fall 'Simulierte Last' erzeugt also eine Last, welche etwa 30% der Maximallast darstellt. Im letzten Test wurden rohe Texte versendet, welche von FU-Remote nicht interpretiert werden. Die Nachrichtengröße betrug dabei 1 KB. Dieser Fall testet die potentiell mögliche Maximalgeschwindigkeit unter diesen Testbedingungen. Das Erzeugen und Versenden der Protobuf Nachrichten geschieht mit nur ca. 14% der Maximalgeschwindigkeit. Das lässt darauf schließen, dass ca. 86% der Kommunikationszeit auf interne Aufgaben abfällt. Der Flaschenhals liegt also bei der CPU, bzw. dem Encodierungsalgorithmus, dem Protobuf Format, etc..

[41] Spezifikation siehe 'RFC 791, Internet Header Format'[1] Seite 10, Kapitel 3.1. 'Internet Header Format'
[42] Spezifikation siehe 'RFC 768, User Datagram Protocol'[31]

8 Ausblick und Fazit

In diesem Kapitel werden einige Einschränkungen des WLAN-Moduls aufgezeigt, ein Ausblick gegeben und das Thema in einer Zusammenfassung abgerundet.

8.1 Einschränkungen

Hardwarebedingt unterliegt die Netzwerkkommunikation einigen Einschränkungen. Diese sollen in diesem Kapitel kurz erläutert werden.

Laut Datenblatt dauert es nach Anlegen der Spannung bis zu ca. 4 Sekunden, bis das WLAN-Modul betriebsbereit ist.[43] Wenn automatisches Verbinden aktiviert ist, dauert es dementsprechend länger, bis eine Kommunikation mit dem WLAN-Modul möglich ist. Dies liegt daran, dass das WLAN-Modul auf neue Befehle nicht antworten wird, solange es versucht, eine Verbindung aufzubauen. Wie lange dies dauert, hängt von vielen Faktoren ab, wie beispielsweise der Latenz der Kommunikation. Es kann also nicht vorhergesagt werden, wie lange es dauert, bis das WLAN-Modul mit dem Netzwerk verbunden und bereit zur Kommunikation ist.

Der Auslastungsgrad der CPU und die Auslastung des Puffers des WLAN-Moduls sind zu keiner Zeit bekannt und nicht auslesbar. So kann es sein, dass die CPU nach dem Verbindungsaufbau noch beschäftigt ist und eingehende Befehle nicht sofort verarbeitet. Die Verarbeitungsgeschwindigkeit von WLAN-Nachrichten zu Paketen, inkl. dem Versenden ist ebenfalls nicht bekannt oder dokumentiert.

Dies legt indirekt eine obere Schranke für die Übertragungsgeschwindigkeit fest. Im Kapitel 'Analyse des Traffics' wurde diese ermessen und analysiert. Dies mag als gute Abschätzung dienen, aber nicht als Garantie. Allerdings haben die Analysen ergeben, dass die technischen Möglichkeiten des WLAN-Moduls und der seriellen Anbindung weit über dem liegen, was der Microcontroller wahrnehmen kann. Lediglich eine Anhäufung eingehender Daten kann den Puffer leicht zum überlaufen bringen.

Im Kapitel Zweck des Protokolls wurde ein vermeintlich sicheres Protokoll vorgestellt. Tatsächlich ist es im schlechtesten Fall unvorhersehbar und völlig willkürlich, was als nächstes Byte aus dem Puffer gelesen wird. Angenommen es werden in einem sehr langsamen Intervall (z.B. 1 B/s) Daten von der seriellen Schnittstelle gelesen. Außerdem werden ausreichend schnell neue Pakete vom WLAN-Modul empfangen und in den Puffer geschrieben, beispielsweise zwischen 700 und 1300 Byte alle 10 ms. Wenn das vermeintlich zweite Zeichen einer Nachricht gelesen wird, könnte es sich tatsächlich um ein willkürliches Datum aus einer anderen Nachricht handeln. Auch wenn dieses Beispiel sehr konstruiert wirkt, widerlegt es, dass eine hundertprozentig sichere Kommunikationen möglich ist. Um diesen Fall zu vermeiden, müssen den Netzwerkfunktionen eine gewisse Laufzeit, beispielsweise durch einen

[43] Zusammenrechnung aus 'AN039, Provisioning Methods with S2W'[18], Kapitel 2.4 'System States'

Scheduler, zugesichert werden und die Bandbreitennutzung des Kommunikationspartner entsprechend eingeschränkt werden. Diese Größen lassen sich relativ leicht berechnen und somit eine fehlerfreie Kommunikation nahezu garantieren.

8.2 Ausblick

In diesem Kapitel sollen mögliche Erweiterungen und neue Verwendungszwecke jenseits des FUB-KIT Projektes des im Rahmen dieser Arbeit entstandene Quellcodes und der dazugehörigen Hardware aufgezeigt werden.

Bislang schöpft der Code nur einen Großteil der Debug-Optionen aus, welche FU-Remote bereits mitbringt und verwendet ausschließlich dessen bereits definierte Nachrichten. Darüber hinaus bietet das hier geschaffene Rahmenwerk die Möglichkeit neue Nachrichten schnell und einfach zu definieren und einzubinden. Nach Implementierung neuer Visualisierungsoptionen in FU-Remote ließen sich diese also mit geringem Aufwand in die Software implementieren. Auch die Nutzung einer Fernsteuerung für motorische Tests wäre gut implementierbar. Ebenfalls denkbar wären neue Nachrichtentypen zur Kommunikation zwischen zwei Robotern. Diese könnten sich gegenseitig ihre vermutete Position oder die Sichtung des Balls oder eines Gegenspielers übermitteln.

Durch die Implementierung des zuverlässigen Protokolls, welches im Abschnitt 'Sichere Datenübertragung' vorgestellt wurde, lässt sich eine große Menge neuer Verwendungszwecke erschließen.

Ein möglicher Einsatzzweck für dieses sichere Protokoll wäre die drahtlose Kalibrierung eines HaViMo2 Moduls. Die Kommunikation, die bislang seriell abgewickelt wurde, müsste nur über erwähntes Protokoll per WLAN abgewickelt werden. Des weiteren ließen sich Debug- und Messdaten auf dem internen Speicher des CM-530 lagern und dieser Speicher per WLAN auslesen und verwalten. Dies ließe sich beispielsweise über ein FTP-ähnliches Protokoll oder ebenfalls über Google Protobuf abwickeln, denn das vorgestellte sichere Protokoll fügt nur 2 Byte Header hinzu, so dass sich beliebige Daten darin kapseln lassen.

8.3 Zusammenfassung

In dieser Arbeit wurde der Aufbau einer drahtlosen Kommunikation mit Hilfe eines seriell angesteuerten WLAN-Moduls erörtert und Anweisungen zum Aufbau bis hin zur Betriebsfähigkeit gegeben. Darüber hinaus wurden Protokolle zur Kommunikation, im Wesentlichen Google Protobuf respektive Header, aber auch GameController-Nachrichten, vorgestellt und dessen Funktionsweise erläutert.

Aufbauend auf diesen Kenntnissen wurde ein Rahmenwerk erarbeitet und vorgestellt, welches Nachrichten über gegebene Hardware und Protokolle versenden und empfangen kann. Bei diesem Rahmenwerk wurde ein besonderes Augenmerk auf Performanz und Redundanz-

freiheit gelegt, um den Hardware-Rahmenbedingungen gerecht zu werden. Aus diesem Grund wurde in einem auswertendem Kapitel eingehend der Footprint der Anwendung analysiert, um sicher zu stellen, dass diesen Ansprüchen gerecht geworden ist.

In einem weiteren Kapitel wurde ein Protokoll zur sicheren Übertragung vorgestellt, um die Möglichkeiten des Frameworks zu demonstrieren.

Zum Schluss wurden einige Einschränkungen der Hard- und Softwarelösung aufgezeigt und ein Ausblick auf zukünftige Nutzungsmöglichkeiten gegeben.

Abbildungsverzeichnis

1 Der CM-530 Microcontroller . . . 6
2 Das Avisaro WLAN Module 1.0 . . . 7
3 Das GS1500M WLAN-Modul . . . 8
4 Die HaViMo2 Kamera . . . 9
5 Beispiel 'Gridding' Algorithmus und 'Region Growing' Algorithmus . . . 9
6 Übersicht einiger FU-Remote Funktionen . . . 11
7 Schaltplan zur Inbetriebnahme des GS1500M . . . 14
8 Beispielkommunikation mit FU-Remote und ausgewählten Testdaten . . . 16
9 Aktivitätsdiagramm für Beispielnutzung des Frameworks . . . 17
10 Beispielsequenz für Ein- und Ausgehende Daten . . . 19
11 Zustandsdiagramm zum Parsen der seriell eingehenden Daten . . . 21
12 Definition von 'message Message' und 'message Status' . . . 25
13 Dezimal- und ASCII-Darstellung einer 'message Status' . . . 25
14 Beispielsequenz des Derivats des Alternierenden-Bit Protokolls . . . 33
15 Zeitaufwand zum Encodieren und Versenden von HaViMo2-Daten . . . 39

Literaturverzeichnis

[1] IETF Network Working Group: *RFC 791, Internet Header Format*. Website, September 1981. – Available online at http://tools.ietf.org/html/rfc791#section-3.1

[2] MSL Technical Committee: *Middle Size Robot League, Rules and Regulations for 2013*. Website, January 2013. – Version - 16.1 20121208, Available online at http://wiki.robocup.org/images/9/98/Msl_rules_2013.pdf

[3] RoboCup Humanoid League: *RoboCup Soccer Humanoid League, Rules and Setup*. Website, May 2013. – Available online at http://www.informatik.uni-bremen.de/humanoid/pub/Website/Downloads/HumanoidLeagueRules2013-05-28.pdf

[4] Avisaro AG: *Avisaro WLAN Modul 1.0 (Produktansicht)*. Vahrenwalderstr. 7 (tch), 30165 Hannover : Website, March 2014. – Available online at http://www.avisaro.com/tl/wlan-modul-serie1.html

[5] AG Intelligente Systeme und Robotik, Institut für Informatik, Freie Universität Berlin: *Berlin United Framework Coderelease*. Arnimallee 7, 14195 Berlin : Website, March 2014. – Project Homepage http://www.fumanoids.de/de/code/framework/

[6] AG Intelligente Systeme und Robotik, Institut für Informatik, Freie Universität Berlin: *Berlin United – FUmanoids, Code Release*. Arnimallee 7, 14195 Berlin : Website, March 2014. – Available online at http://www.fumanoids.de/code/coderelease/

[7] AG Intelligente Systeme und Robotik, Institut für Informatik, Freie Universität Berlin: *Berlin United – FUmanoids, EROLF*. Arnimallee 7, 14195 Berlin : Website, March 2014. – Available online at http://www.fumanoids.de/de/hardware/erolf/

[8] AG Intelligente Systeme und Robotik, Institut für Informatik, Freie Universität Berlin: *FUmanoids Code Release 2012 (released: December 27, 2012) 2012.1*. Arnimallee 7, 14195 Berlin : Website, March 2014. – Available online at http://maserati.mi.fu-berlin.de/fumanoids/wp-content/plugins/download-monitor/download.php?id=2

[9] Universität Bonn, Institute for Computer Science, Departments: I, II, III, IV, V, VI: *NimbRo-OP Humanoid TeenSize Open Platform Robot, Concept*. Römerstr. 164, 53117 Bonn : Website, March 2014. – Available online at http://www.nimbro.net/OP/

[10] Google Inc: *Protocol Buffers. Google's Data Interchange Format*. 1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA : Website, March 2014. – Project Homepage http://code.google.com/p/protobuf

[11] Aimonen, Petteri: *GPRS Boat*. Website, March 2014. – Available online http://essentialscrap.com/boat/

[12] Aimonen, Petteri: *Nanopb - protocol buffers with small code size*. Website, March 2014. – Project Homepage at http://koti.kapsi.fi/jpa/nanopb/

[13] Aimonen, Petteri: *Nanopb: Basic Concepts*. Website, March 2014. – Available online in project Documentation http://koti.kapsi.fi/jpa/nanopb/docs/concepts.html

[14] Avisaro AG (Hrsg.): *Avisaro WLAN Protokoll Adapter*. 2.1. Vahrenwalderstr. 7 (tch), 30165 Hannover: Avisaro AG, 2012

[15] Brauer, Jörg ; Schlich, Bastian ; Reinbacher, Thomas ; Kowalewski, Stefan: Stack Bounds Analysis for Microcontroller Assembly Code. 4 (2009)

[16] Combs, Gerald ; contributors: *Wireshark 1.6.7*. Website, March 2014. – Available for Download online at http://www.wireshark.org/download.html

[17] Deri, Luca: *ntopng*. Website, March 2014. – Available online at http://www.ntop.org/get-started/download/

[18] GainSpan Corporation (Hrsg.): *AN039, Provisioning Methods with S2W*. 3590 N. First Street, Suite 300, San Jose in CA 95134: GainSpan Corporation

[19] GainSpan Corporation (Hrsg.): *GS1500M, 802.11 b/g/n, Low-Power Wi-Fi Module, Data Sheet*. 3590 N. First Street, Suite 300, San Jose in CA 95134: GainSpan Corporation

[20] GainSpan Corporation (Hrsg.): *Serial-To-WiFi Adapter, Application Programming Guide, 2.4.3/3.4.3*. 3590 N. First Street, Suite 300, San Jose in CA 95134: GainSpan Corporation, 12 2012

[21] Gamma, Erich (Hrsg.) ; IFA Consulting (Veranst.): *The Extension Objects Pattern*. Ceresstrasse 27, CH-8034 Zurich, . – Available online at http://www.mif.vu.lt/ plukas/resources/Extension

[22] Google: *Protocol Buffers, Encoding*. Website, April 2012. – https://developers.google.com/protocol-buffers/docs/encoding?hl=de

[23] HaViSys UG (Hrsg.): *HaViMo2 Image Processing Module*. Ferdinandstr. 3A, 12209-Berlin: HaViSys UG, March 2010. – Available online at http://robosavvy.com/RoboSavvyPages/Support/Hamid/HaViMo2.pdf

[24] James F. Kurose, Keith W. R.: *Computernetze, Ein Top-Down-Ansatz*. Pearson Studium, 2008. – ISBN 978–3–8273–7330–4. – 4. aktualisierte Auflage, Deutsche Übersetzung von 'Computer Networking, A Top-Down Approach'

[25] Jann, Christian: *Eine WLAN-Platine zum selber bauen*. Website, March 2014. – Available online at http://www.jann.cc/2012/08/07/eine_wlan_platine_zum_selber_bauen.html

[26] Kim, ROBOTIS Hee-il Jaehong: *DARwIn-OP downloads page*. Website, March 2014. – Available online at http://sourceforge.net/projects/darwinop/

[27] Lindblom, Nicolas: *Controlling the stack size*. Messe Campus, Werner-Eckert-Strasse 9, D-81829 Muenchen: IAR Systems GmbH. – Available online at https://www.iar.com/Global/Resources/Developers_Toolbox/Building_and_debugging/Controlling the stack size.pdf

[28] Organization, W3C S.: *SOAP Version 1.2*. Website, March 2014. – http://www.w3.org/TR/soap/

[29] Paulishen, Matthew: *CM-530 C/C++ 'easy-functions'*. Website, March 2014. – Available online at https://github.com/tician/cm530/

[30] Petteri Aimonen, Michael P.: *Nanopb - protocol buffers with small code size, dynamic alloc dev (Branch)*. Website, March 2014. – Available online at http://code.google.com/p/nanopb/source/browse/?name=dynamic_alloc_dev

[31] Postel, J.: *RFC 768, User Datagram Protocol*. Website, August 1980. – Available online at https://www.ietf.org/rfc/rfc768.txt

[32] ROBOTIS: *ROBOTIS e-Manual v1.15.00: CM-530*. Website, March 2014. – Available online at http://support.robotis.com/en/product/auxdevice/controller/cm530.htm

[33] ROBOTIS: *ROBOTIS e-Manual v1.16.00: Firmware Installer*. Website, March 2014. – Available online at http://support.robotis.com/en/product/darwinop/development/tools/firmware_installer.htm

[34] ROBOTIS: *ROBOTIS Firmenhomepage*. Website, March 2014. – Available online at http://www.robotis.com/xe/

[35] Seifert, Daniel ; Thomas, Dirk ; Scholz, Dorian ; L., Thomas ; Röfer, Thomas: *RoboCup GameController*. Website, March 2014. – Project Homepage http://sourceforge.net/projects/robocupgc/

[36] Seifert, Daniel ; Thomas, Dirk ; Scholz, Dorian ; L., Thomas ; Röfer, Thomas: *RoboCup GameController: RoboCupGameControlData.h*. Website, March 2014. – Project Homepage http://sourceforge.net/p/robocupgc/code/HEAD/tree/trunk/doc/RoboCupGameControlData.h

[37] Society, IEEE C.: IEEE 802.11™: Wireless LANs. (2012), March. – Available online at http://standards.ieee.org/about/get/802/802.11.html

9 Abkürzungsverzeichnis

HaViMo2 Hamid Vision Module 2

SPI Serial Peripheral Interface

UART Universal Asynchronous Receiver Transmitter

PCU PC UART, Serielle Schnittstelle auf Mini-USB Ausgang

Protobuf Google Protocol Buffers

BACnet Building Automation and Control Networks

SSID Service Set Identification

GPIO General Purpose Input Output