Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Arne Wischer

Design und Implementierung eines verteilten linuxbasierten Entwicklungssystems für Nahbereichsfunkanwendungen mit ARM-Architektur

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Arne Wischer

Design und Implementierung eines verteilten linuxbasierten Entwicklungssystems für Nahbereichsfunkanwendungen mit ARM-Architektur

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. rer. nat. Hans Heinrich Heitmann
Zweitgutachter: Prof. Dr.-Ing. Franz Korf

Eingereicht am: 17. April 2013

Arne Wischer

Thema der Arbeit
Design und Implementierung eines verteilten linuxbasierten Entwicklungssystems für Nahbereichsfunkanwendungen mit ARM-Architektur

Stichworte
Entwicklungssystem, IEEE 802.15.4, OpenWRT, OpenOCD, Carambola, JTAG, GDB, MIPS, ARM, UART

Kurzzusammenfassung
Dieses Arbeit beschreibt die Entwicklung eines verteilten Systems, mit dessen Hilfe eingebettete Funksysteme entworfen, implementiert und analysiert werden können. Das System soll es einem Entwickler dabei ermöglichen, viele dieser Entwicklungsschritte netzwerkgestützt durchzuführen. Da beispielsweise auch die Funkdaten analysiert werden sollen, ist eine hohe Genauigkeit der Datenerfassung notwendig. Realisiert wurde das Entwicklungssystem mithilfe mehrerer MIPS-Entwicklungsboards, die als „Datenbrücken“ fungieren und JTAG-Steuerbefehle über GDB zur Verfügung stellen sowie über UART gesammelte Daten an eine Clientanwendung durchreichen. Diese Anwendung sortiert die eingehenden Daten nach ihrem Auftreten und zeigt sie dem Entwickler an.

Arne Wischer

Title of the paper
Design and implementation of a distributed Linux-based development platform for near-field radio systems using the ARM architecture

Keywords
development platform, IEEE 802.15.4, OpenWRT, OpenOCD, Carambola, JTAG, GDB, MIPS, ARM, UART

Abstract
This thesis describes the development of a distributed system to help design, implement and analyze embedded radio systems. The system allows a developer to perform many of these development steps with network support. Since for example, the radio data needs to be analyzed, a high accuracy while collecting data is necessary. The development system was implemented using multiple MIPS development boards, working as "data bridges" to provide JTAG-commands using GDB as well as provide data collected via UART to a client application. This application sorts the incoming data according to their occurrence and shows them to the developer.

Inhaltsverzeichnis

Abkürzungsverzeichnis

CCA Clear Channel Assessment. 30–32

DAP Debug access port. 10, 11

GDB GNU Debugger. 25, 28, 37, 43, 44, 55, 62

GPIO General purpose input/output. 41

IC Integrated Circuit. 9

IDE Integrated Development Environment. 6, 7, 15, 23, 28

ISP In-System Programmer. 11

JTAG Joint Test Action Group. 6, 9–11, 17, 22, 24, 27, 37, 40, 42, 43, 55, 56, 62, 64

MAC Medium Access Control. 22, 23, 29, 30, 32

MVP Model View Presenter. 54

NTP Network Time Protocol. 33, 58

ROM Read-Only Memory. 9, 11, 37

RTOS Real-time operating system. 44, 56

SPI Serial Peripheral Interface Bus. 19, 20, 22, 23, 36, 61

SWD Serial Wire Debug. 10, 11, 17, 24

TAP Test access port. 9–11, 37

UART Universal Asynchronous Receiver Transmitter. 19, 20, 22, 23, 29, 32, 35, 36, 40–42, 45, 48–51, 56, 61, 62, 64

UCI Unified Configuration Interface. 40, 41

UML Unified Modeling Language. 6

1. Einführung

Inhaltsangabe

1.1. Motivation

Eingebettete Systeme sind aus unseren heutigen Welt nicht mehr wegzudenken. Sie dienen zur Heimautomatisierung, Prozesssteuerung und vielem Anderen. Diese Systeme kommunizieren häufig über Funkprotokolle, die auf eine hohe Energieeffizienz ausgelegt sind und sich in ihrem Netzwerkaufbau oftmals dynamisch organisieren können. Als Beispiel für derartige Funksysteme lassen sich *ZigBee*, *ANT*, *Z-Wave*, *Bluetooth low energy* oder *MiWi* nennen.

Gleichzeitig sind diese Systeme stark in bestehende Umgebungen wie Anlagen oder Bausubstanz integriert beziehungsweise müssen in eine solche Umgebung integriert werden.

Die Problematik liegt hierbei in der Entwicklung dieser vernetzten Systeme. Wird ein einzelnes System entwickelt, sind die Randbedingungen meist recht klar definiert, Anwendungen zur Softwareentwicklung sind weit verbreitet und gängige Testverfahren können die Einhaltung der Vorgaben überprüfen.

Soll jedoch ein Funknetzwerk, bestehend aus mehreren einzelnen Systemen, entwickelt werden, offenbaren sich manche Schwierigkeiten erst im Zusammenspiel der einzelnen Komponenten. Auch können störende Umwelteinflüsse in der Konzeption nicht immer vollständig erfasst und bedacht werden.

Dazu kommt, dass meist eine Vielzahl identischer Systeme miteinander kommuniziert und so auch oft die gleiche Software auf diesen Systemen zum Einsatz kommt.

Und auch eine fortlaufende Wartung dieser Systeme ist, aufgrund der teilweise hohen räumlichen Abstände, nicht immer sehr einfach zu handhaben.

1.2. Existierende Lösungen

Für stark in ihren Ressourcen eingeschränkte, verteilte Funksysteme existieren eine Anzahl hierfür optimierter Betriebssysteme. Das TinyOS-Projekt[1] zum Beispiel ist ein an der Universität Berkeley entstandenes und spezifisch für den Betrieb drahtloser Sensornetzwerke entwickeltes Betriebssystem. Weitere Beispiele wären Contiki[2] oder FreeRTOS[3].

Für Änderung oder Aktualisierung der meisten dieser Betriebssysteme wird eine direkte Verbindung zu dem Rechner des Entwicklers benötigt. Dies kann, insbesondere mit einer hohen Anzahl Sensorknoten, durchaus zu einer logistischen Herausforderung werden.

Definition 1.1 (Zielsystem) *Das* Zielsystem *ist das System, welches mit Hilfe des* Entwicklungssystems *realisiert werden soll.*

Um dieses Problem zu umgehen, verfolgt TinyOS einen anderen Ansatz. Will man auf mehrere TinyOS-basierte Zielsysteme beziehungsweise Sensorknoten eine neue Softwareversion aufspielen, kann man sich die in das System integrierte Software „Deluge 2“[4] zunutze machen. Diese Software ermöglicht es, eine neue Softwareversion mittels eines Bootloaders und über das bereits bestehende Funknetzwerk zu verteilen. Hierbei funktioniert die Verteilung „epidemisch“ und wird damit größtenteils durch die Sensorknoten selbst verwaltet.

Dieses Vorgehen hat jedoch wiederum den Nachteil, dass für eine Verteilung der Software überhaupt erst einmal ein Netzwerk unter den Sensorknoten existieren muss. So wird dieses Verfahren unmöglich, sobald ein komplett neues Funkprotokoll entwickelt werden soll.

Wünschenswert wäre es also, die Verteilung neuer Softwareversionen möglichst kabellos und gleichzeitig ohne starke Abhängigkeit vom Zielsystem oder dem verwendeten Funksystem durchführen zu können.

Während der Entwicklung eines solchen Funksystems spielen auch andere Aspekte eine Rolle. So ist es ab einem gewissen Punkt in der Entwicklung sicher wichtig, diese Systeme und die Abläufe im Funknetz im laufenden Betrieb analysieren zu können.

1.3. Ziele

Ziel dieser Bachelorarbeit ist es, ein Entwicklungssystem zu schaffen, mit dessen Hilfe sich integrierte und mit einem Funkmodul ausgestattete Zielsysteme entwickeln und in ihrem Betrieb untersuchen lassen.

Hierzu werden verschiedene Hilfsmittel benötigt, deren einzelne Bestandteile herausgearbeitet, entworfen und implementiert werden sollen.

1.4. Gliederung

In Kapitel 2 „Analyse eines Entwicklungssystems“ werden die Elemente eines allgemeinen Entwicklungssystems analysiert. Anschließend sollen Szenarien vorgestellt werden, die bei der Entwicklung eines verteilten, eingebetteten Systems auftreten können. Daraus erwachsend sollen dann die zu erfüllenden Anforderungen an das zu entwerfende Entwicklungssystem gestellt werden.

In Kapitel 3 „Konzeption des verteilten Entwicklungssystems“ wird zuerst die grundlegende Idee eines verteilten Entwicklungssystems erläutert, bevor anschließend Vorentscheidungen über die zu verwendende Hardware getroffen werden. Hierbei müssen auch die für die Realisierung wichtigen Randbedingungen ermittelt werden.

Kapitel 4 „Realisierung und Analyse“ beschreibt dann die Umsetzung des konzipierten Systems, erläutert die Funktionen der einzelnen Bestandteile und analysiert abschließend das System hinsichtlich der Umsetzung der Vorgaben.

Kapitel 5 „Ausblick und Fazit“ bietet zum Schluss einen Ausblick auf mögliche Verbesserungen und Fortentwicklungen des Systems.

2. Analyse eines Entwicklungssystems

Inhaltsangabe

In diesem Kapitel soll aufgezeigt werden, woraus ein Entwicklungssystem im Allgemeinen besteht und welche Anforderungen sich also für ein zu entwerfendes Entwicklungssystem ergeben.

Definition 2.1 (Entwicklungssystem) *Zu einem* Entwicklungssystem *gehören alle Hard- und Softwareelemente, die nötig sind, um die Umsetzung eines bestimmten Softwareprojektes zu erreichen. Dies ist oft eine Vielzahl unterschiedlichster, miteinander interagierender Bestandteile.*

Da die grundlegenden Abläufe während der Entwicklung eines eingebetteten Systems oft gleich sind, wird in diesem Kapitel hauptsächlich eine allgemeine Sicht dargestellt. Die Umsetzung auf ein Entwicklungssystem für verteilte Funknetzwerke erfolgt in Kapitel 3.

2.1. Elemente eines Entwicklungssystems

Die Elemente eines Entwicklungssystems lassen sich grob in drei Kategorien einordnen.

Zu bedenken ist, dass bei der Entwicklung von Soft- und Hardware viele der einzelnen Elemente, auch kategorieübergreifend, miteinander interagieren.

2.1.1. Vorgehensmodell

Zu jedem Entwicklungsprozesses gehört auch immer die Entscheidung über eine Vorgehensweise. Viele Vorgehensweisen basieren dabei auf Vorgehensmodellen, die in Industrie und Wirtschaft weit verbreitet sind. Als Beispiele lassen sich hier das *Wasserfallmodell*, das *V-Modell*, *Testgetriebene Entwicklung*, *Scrum* oder auch *Extreme Programming* nennen.

All diesen Modellen ist gemein, dass einige ihrer Schritte unbedingt Unterstützung durch Soft- und Hardware bedürfen oder diese die Entwicklung unter Umständen beschleunigen kann.

Da es eine klare Strukturierung zeigt, soll hier exemplarisch für alle Vorgehensmodelle das **Wasserfallmodell** aufgeführt werden.

Andere Vorgehensmodelle strukturieren ihren Ablauf unterschiedlich, jedoch sind die zentralen Entwicklungsschritte meist identisch.

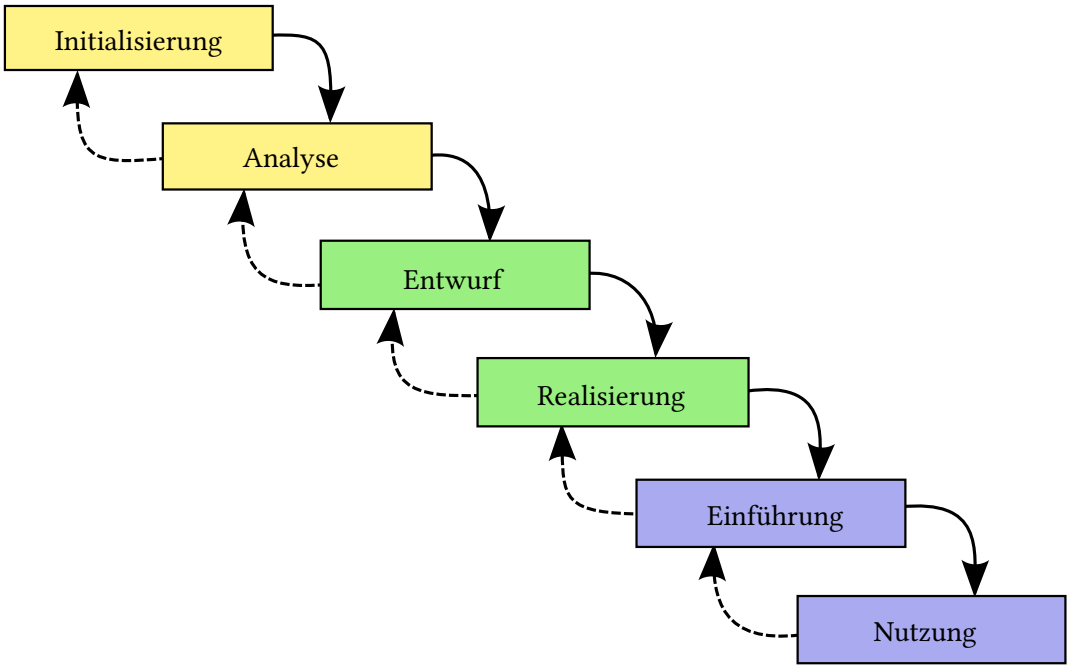
Durch die Wahl des Vorgehensmodells lässt sich allerdings festlegen, welche Priorität die einzelnen Bestandteile des Entwicklungssystems einnehmen. So legt zum Beispiel eine Testgetrieben Entwicklung ihren Schwerpunkt auf die Möglichkeiten zum Testen der Soft- und Hardware, während das Wasserfallmodell relativ ausgewogene Anforderung an alle Bestandteile hat.

Wasserfallmodell

Das *Wasserfallmodell* ist ein klassisches, nicht-iteratives Vorgehensmodell, dessen Ablauf man sich anhand eines Wasserfalls gut visualisieren kann.

In Abbildung 2.1 ist der mögliche Ablauf eines *Wasserfallsmodells* dargestellt.

In den **Initialisierungs- und Analysephasen** wird oft der Kunde in die Entwicklung eingebunden. Typischerweise werden hier deshalb keine oder nur wenige spezialisierte Tools verwendet. Zu der verwendeten Software zählt hier unter anderem *Projektmanagementsoftware*.



Jedes Element in diesem Diagramm stellt einen der Schritte und damit eine Phase des *Wasserfallmodells* dar. Die Übergänge zwischen den einzelnen Phasen sind oft fest definiert.

Abbildung 2.1. Ablauf eines Softwareprojektes nach dem Wasserfallmodell. Erstellt nach [5]

Die **Entwurfs- und Realisierungsphasen** jedoch benötigen oft eine hohe Softwareunterstützung. Während in der Entwurfsphase meist Tools zur Visualisierung und Strukturierung von Softwarearchitektur eingesetzt werden *(Unified Modeling Language (UML), Struktogramme, etc.)*, werden in der Realisierungsphase meist Werkzeuge eingesetzt, die von der konkreten Anforderung abhängen *(Compiler, Integrated Development Environments (IDEs), Debugger).*

Um einen Übergang in die **Einführungs- und Nutzungsphasen** zu ermöglichen, muss die Software beziehungsweise das Gesamtsystem oft Tests durchlaufen, die das System auf eine Erfüllung der Anforderungen überprüfen. Hierfür werden häufig Testpraktiken benutzt, die auch soft- und hardwaregestützt ablaufen *(Joint Test Action Group (JTAG), Unittests, In-circuit Tests).*

Um ein Ausliefern der Software zu ermöglichen, werden Werkzeuge zum Deployment benötigt. Dies erfordert oft ein Zusammenspiel aus Hardware- und Softwarekomponenten *(Installer, Softwarepakete, Flasher).*

Oft durchläuft ein System in seiner Nutzungsphase auch **Wartungszyklen**. Diese Zyklen umfassen oft das Aufspielen neuer Softwareversionen auf das Zielsystem und setzen eine möglichst einfache Wartbarkeit des Systems voraus.

2.1.2. Software

Software unterstützt Softwareprojekte in allen Phasen der Durchführung.

Obwohl viele dieser einzelnen Elemente auf Hardwareunterstützung angewiesen sind, sollen in diesem Abschnitt nur die Softwarebestandteile beschrieben werden. Soweit es nötig ist, werden die zugehörigen Gegenstücke in Unterabschnitt 2.1.3 beschrieben.

Im Gegensatz zu den Hardwareelementen wird die hier erwähnte Software oft nicht oder nur geringfügig modifiziert, soll sie zur Entwicklung verschiedener eingebetteter Systeme verwendet werden.

- **IDEs**

  IDEs sind **zentrale Entwicklungsoberflächen** der Softwareentwickler. In dieser Umgebung werden oftmals viele Tools zusammengefasst, die für den Entwicklungszyklus notwendig sind. Sie dienen dazu, Quelltext zu verwalten und zu schreiben, Versionsverwaltung zu betreiben, Code zu kompilieren, die entwickelte Software zu testen, zu deployen und zu debuggen.

  Oft werden die in den folgenden Punkten aufgeführten Tools in eine *IDE* integriert, um Entwicklungsabläufe zu vereinfachen und zu beschleunigen.

  Beispiele: Eclipse, NetBeans, CodeWarrior, Microsoft Visual Studio, Qt Creator

- **Compiler**

  Compiler sind für viele Softwareprojekte von essentieller Bedeutung. Speziell im eingebetteten Bereich werden überwiegend kompilierte Programmiersprachen wie C oder C++ eingesetzt, da sie nach dem Kompiliervorgang in **nativem Maschinencode** vorliegen und so oft eine höhere Ausführungsgeschwindigkeit erzielen. Außerdem können sie meist besser an das Zielsystem angepasst werden, da sie keine weiteren Frameworks oder

Interpreter mit ungenutzte Funktionen benötigen, die auf dem System unter Umständen wertvollen Speicherplatz „verbrauchen“.

Soll eine kompilierte Software auf einem anderen als dem Entwicklungssystem eingesetzt werden, so wird ein sogenannter **Cross-Compiler** benötigt. Diese speziellen Compiler sind in der Lage, Software zu kompilieren, die später auf einer andere Rechnerarchitektur (z.B. ARM statt Intel) und/oder auf einem anderen Betriebssystem (z.B. Linux statt Windows) betrieben werden soll.

Beispiele: GCC, IAR C/C++ Compiler, ARM RVCT Compiler, Intel C++ Compiler, CodeWarrior Compiler

- **Debugger**

  Der *Debugger* ist ein Tool, mithilfe dessen ein Entwickler eine Software während ihrer Laufzeit untersuchen, steuern und, in einem gewissen Rahmen, modifizieren kann. Er dient dem Entwickler hauptsächlich zur **Fehlersuche**, da es mit seiner Hilfe sehr leicht ist, die Vorgänge in einer Software nachzuvollziehen.

  Der Entwickler ist durch das Setzen von Breakpoints[1] mittels des *Debuggers* in der Lage, das System bis zu einem festgelegten Systemzustand laufen zu lassen. Anschließend kann er den weiteren Programmablauf schrittweise ausführen (single-stepping) oder mitunter auch geringfügige Änderung am Speicher des Systems vornehmen. Außerdem sind die meisten *Debugger* in der Lage, das Zielsystem zurückzusetzen und so einen definierten Ausgangszustand herzustellen.

  Debugger sind in integrierten Systemen oft auf eine hohe Hardwareunterstützung angewiesen. In Abschnitt 2.1.3 wird dies näher beschrieben.

  Beispiele: GDB, IDB (Intel Debugger), Valgrind

- **Testtools**

  In die Kategorie der *Testtools* fallen alle Elemente die benötigt werden, um die Funktionsfähigkeit eines Systems **zu prüfen**. Außerdem dienen diese auch oft dazu, ein Gesamtsystem sowie dessen Teilsysteme auf die Erfüllung der Anforderungen zu prüfen.

[1] Willkürliche, vom Entwickler vor dem Start der Software zu setzende, „Haltepunkte“ im Programmablauf.

Um die einzelnen Bestandteile eines Systems zu überprüfen, existieren viele verschiedene Ansätze. Meist werden Tests grob in *Modultests*, *Integrationstests*, *Systemtests* und *Abnahmetests* unterteilt.

Alle diese Testarten setzen jedoch eine grundlegende Testbarkeit des Systems voraus.

Beispiele: CUnit, hitex Tessy, Boost Test Library

- **Deploymenttools**

  Um ein System ausliefern zu können, muss es für den Endanwender vorbereitet werden.

  Im Embeddedbereich erfolgt dies meist durch ein **Übertragen der Firmware** in den Read-Only Memory (ROM) des Zielsystems. Dieser Vorgang wird jedoch, zu Testzwecken, auch in der Entwicklungsphase eines Projektes oft durchgeführt und muss daher entsprechend einfach und möglichst unkompliziert sein.

  Beispiele: Installer, Paketverwaltungssysteme, Skripte

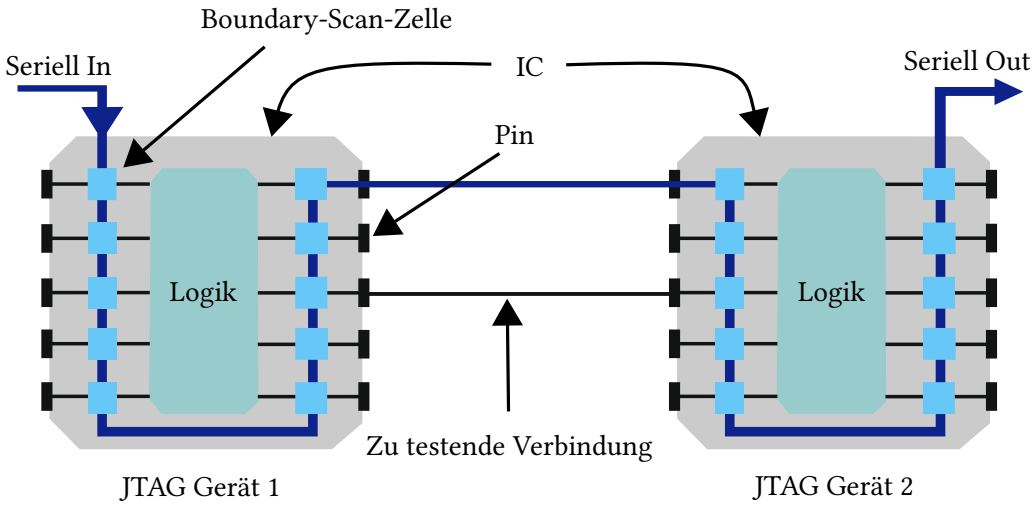
2.1.3. Hardware

Welche Art von Unterstützung in welchem Umfang von Hardware geleistet werden muss, ergibt sich oft aus den Projektvorgaben. Im Speziellen ist hierbei von Bedeutung, welche Hardware als Zielsystem zum Einsatz kommt.

Insbesondere im Bereich der eingebetteten Systeme ist die Hardware eines Entwicklungssystems von hohere Bedeutung, da sie oftmals eine Art Schnittstelle zwischen dem Entwicklersystem und dem Zielsystem darstellen muss.

- **Testtools**

  Um Tests durchführen zu können, wird oft auch eine Unterstützung durch Hardware benötigt. Der JTAG-Port ist ein System, welches ursprünglich für genau diese Zwecke konzipiert wurde. Er ermöglicht es, einen sogenannten „boundary-scan“ durchzuführen. Jedes in Abbildung 2.2 mit „Integrated Circuit (IC)“ markierte Bauteil muss dafür über einen, in der Abbildung nicht eingezeichneten, sogenannten Test access port (TAP) oder



Diese Abbildung zeigt den vereinfachten Aufbau einer Verkettung mehrerer JTAG-Komponenten zu einer JTAG-Chain. Ein JTAG-Adapter liefert über „Seriell In“ Testdaten und überprüft die Ausgabe der zugehörigen Daten von „Seriell Out“.

Abbildung 2.2. Funktionsweise von JTAG. Erstellt nach [6]

Debug access port (DAP)[2] verfügen. Diese befinden sich *nicht* in der Logik des Bauteils, sondern stellen eine separate Steuerschaltung dar.

Die TAPs/DAPs werden zu einer sogenannten JTAG-Chain verkettet und ermöglichen es, die Ein- und Ausgänge der einzelnen Bauteile über die zugehörigen „Boundary-Scan-Zellen“ mit Prüfdaten zu betreiben, Bauteile komplett zu überbrücken oder auch sie in ihrem Betrieb anzuhalten. Die in Abbildung 2.2 als „Zu testende Verbindung“ markierte Leitung kann so auf Unterbrechungen oder Überbrückungen geprüft werden.

Einen darauf aufbauenden Funktionsumfang bietet das, vorwiegend in ARM Prozessoren zum Einsatz kommende, Serial Wire Debug (SWD)[7] an.

Um (auch analoge) Bauteile und Platinen in größerem Umfang zu testen, dient der *in-circuit Test*. Hierbei wird ein speziell angefertigtes „Nagelbett“ auf eine Platine gesenkt. Dies erlaubt zusätzlich zum *boundary-scan* das Testen von Widerständen, Kapazitäten und anderen elektrischen Kenngrößen.

Beispiele: JTAG-Debugger, SWD-Debugger, Spy-Bi-Wire, In-circuit Testanlagen

[2] Wird von neueren ARM Kernen benutzt. Ermöglicht auch Tracing.

- **Debugger**

  Um Softwaredebugging, wie in Abschnitt 2.1.2 beschrieben, im Umfeld eines Mikroprozessors zu erleichtern beziehungsweise überhaupt zu ermöglichen, besitzen diese oft verschiedene Eigenschaften. Um eine Software Schritt-für-Schritt auszuführen, wird eine Unterstützung von der CPU vorausgesetzt. Prozessorkerne der ARM-Architektur lassen sich zum Beispiel mittels des DAP/TAP in einen *Debug-Modus* bringen. Dieser *Debug-Modus* ist Voraussetzung dafür, Single-Step-Debugging duchführen oder Breakpoints setzen zu können.

  Da ein JTAG-Port einen direkt Zugriff auf alle Eingänge eines Mikrocontrollers bietet und der DAP/TAP tief in die Hardware integriert ist, lassen sich über diese Verbindung oft auch Register, Speicherbereiche und andere Komponenten untersuchen und modifizieren.

  Beispiele: DebugWIRE, JTAG-Debugger, SWD-Debugger

- **Flasher**

  Um, wie in Abschnitt 2.1.2 beschrieben, ein Zielsystem mit einer Firmware beschreiben zu können, wird oft ein Hardwarebauteil benötigt.

  Abhängig vom verwendeten Zielsystem erfolgt ein „Flashen“ dabei häufig über extra dafür bestimmte Hardware. Eine mögliche Variante ist hierbei zum Beispiel die Verwendung eines In-System Programmers (ISPs). Ein ISP ist ein System, das speziell dafür ausgelegt ist, Firmware auf ein bereits eingebautes Bauteil zu übertragen.

  Es ist jedoch oft auch möglich, einen JTAG-Debugger als *Flasher* einzusetzen. Dafür muss er lediglich in der Lage sein, den ROM modifizieren zu können. Dies bieten die meisten Zielsysteme an.

  Beispiele: AVRISP mkII, Waveshare LPC ISP(mini), JTAG-Debugger

2.2. Szenarien eines Funksystems

Für das Design eines Entwicklungssystems ist außerdem entscheidend, dass die Zielsysteme über eine Funkschnittstelle verfügen.

Soll nun also ein neues Funkprotokoll entworfen werden, ist es sicher nützlich, die Abläufe innerhalb des Funknetzwerkes auf Korrektheit zu überprüfen.

So sind mehrere Szenarien denkbar, für die eine Analyse der genauen Abläufe wünschenswert wäre.

2.2.1. Reichweite der Zielsysteme

Da ein zu entwerfendes Netzwerk mitunter eine hohe räumliche Verteilung aufweist, kann nicht garantiert werden, dass jeder Netzwerkknoten zu jedem anderen Knoten eine Verbindung aufbauen kann.

Um die Empfangbarkeit der verschiedenen Netzwerkknoten untereinander zu überprüfen, muss von jedem Knoten bekannt sein, welche Knoten er empfangen kann und welche Knoten ihn empfangen können.

Hierfür ließe sich folgender simpler Ablauf konstruieren:

Jeder Netzwerkknoten sendet eine Broadcastmitteilung an das gesamte Netzwerk. Jeder Knoten der diese Nachricht empfängt, befindet sich nun also in Reichweite des Absenders.

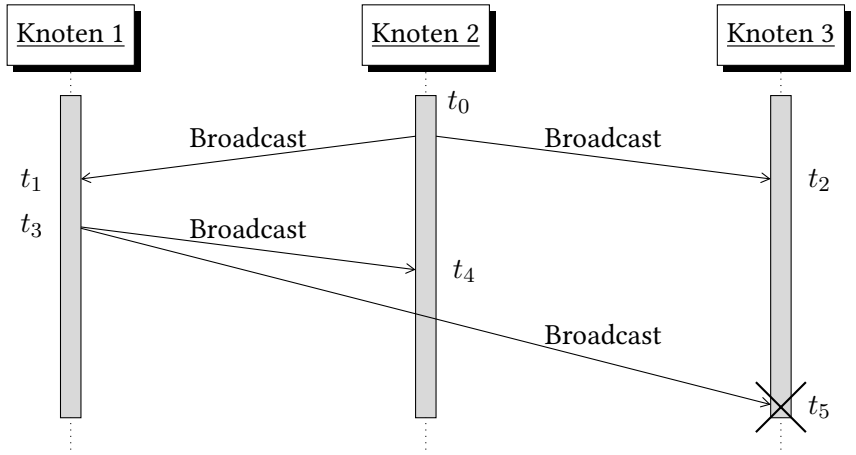
Für die Auswertung dieses Vorgangs müssen nun jedoch von allen Netzwerkknoten die erfassten Daten gesammelt und analysiert werden. Erst aus diesen gesammelten Daten lassen sich Rückschlüsse auf die Verbindungsqualität ziehen.

Falls sich das Funkprotokoll jedoch noch in seiner Entwicklungsphase befindet, benötigt man eine **physische Verbindung** zu den jeweiligen Netzwerkknoten, um an diese Informationen zu gelangen. Anderenfalls könnte man das Funknetzwerk der Netzwerkknoten selbst nutzen, um diese Informationen drahtlos abzufragen.

Abbildung 2.3 stellt einen beispielhaften Ablauf eines solchen Broadcasting-Vorganges dar. Während die zu Zeitpunkt t_0 von *Knoten 2* abgesetzte Nachricht sowohl von *Knoten 1* als auch von *Knoten 3* empfangen wurde, erreicht die zu Zeitpunkt t_4 gesendete Broadcast-Nachricht von *Knoten1* bei Zeitpunkt t_5 nur *Knoten 2*.

Sammelt man die Daten dieser Abläufe nun und sortiert sie nach den Zeitpunkten ihres Auftretens, ergibt sich ein Ablauf wie in Abbildung 2.4 dargestellt.

Führte man diese Testreihe fort, ließe sich sehr leicht eine Topologie der Empfangsstärken bilden.



Hier können zu vier verschiedenen Zeitpunkten Daten aufgezeichnet werden, um einen zeitlichen Verlauf untersuchen und verdeutlichen zu können.

Abbildung 2.3. Möglicher Ablauf eines fiktiven Funkprotokolls

Zielsystem	Zeitstempel	Nachricht
<Knoten 2>	t_0	Broadcast gesendet
<Knoten 1>	t_1	Broadcast von *Knoten 2* erhalten
<Knoten 3>	t_2	Broadcast von *Knoten 2* erhalten
<Knoten 1>	t_3	Broadcst gesendet
<Knoten 2>	t_4	Broadcast von *Knoten 1* erhalten
Keine Daten für Zeitpunkt t_5		

Ein Ablauf wie in Abbildung 2.3 dargestellt, könnte diese Reihenfolge von Nachrichten erzeugen.

Abbildung 2.4. Erfassung der Daten des Ablaufs aus Abbildung 2.3

2.2.2. Kollisionen

Auch die Kollisionserkennung ist für ein Funksystem von hoher Bedeutung. Senden zwei Funkknoten gleichzeitig eine Nachricht, überlagern sich die Funksignale und es ist für ein

empfangendes System, sofern beide Sender in Reichweite sind, nicht mehr möglich, die einzelnen Nachrichten auseinanderzuhalten.

Will man diese Kollisionen nun während der Entwicklung untersuchen, so muss eine Möglichkeit existieren, diese auch darstellen zu können. Des Weiteren wird für die Darstellung sicher auch eine gewisse Präzision erforderlich sein, da Kollisionen innerhalb eines Netzwerkes immer eine Gleichzeitigkeit bedeuten.

2.2.3. Weitere mögliche Szenarien

Mit steigender Komplexität des zu entwerfenden Funkprotokolls steigt die Komplexität solcher Abläufe. So ist zum Beispiel, wie im Funkprotokoll ZigBee vorgesehen[8], denkbar, dass die Funkknoten verschiedene Aufgaben im Routing und Aufbau des eigenen Funknetzwerkes übernehmen. Auch hierfür ist unter Umständen eine möglichst genaue Erfassung der Abläufe nötig.

2.3. Anforderungen an das Entwicklungssystem

Nachdem klar ist, aus welchen Bestandteilen ein Entwicklungssytem besteht, müssen nun die konkreten Anforderungen gestellt werden.

Diese Anforderungen sollen als Ziel dieser Arbeit erfüllt werden und die Entwicklung eines Zielsystems, bestehend aus mehreren unabhängigen Modulen, ermöglichen. Um dies zu demonstrieren, soll außerdem exemplarisch ein minimales Zielsystem entwickelt werden.

2.3.1. Debugging

Die Entwicklung einer Software und die Suche nach Fehlern in dieser wird durch den Einsatz eines Debuggers enorm erleichtert.

Da es sich um ein verteiltes Netzwerk handelt, wäre es wünschenswert, mehrere Systeme gleichzeitig debuggen zu können. Hierdurch könnte man zum Beispiel genauestens untersuchen, warum ein System zu einem Zeitpunkt eine bestimmte Nachricht sendet.

Da für Debugging häufig auch ein Reset des Zielsystems notwendig ist, sollte diese Funktion auch für mehrere Systeme simultan durchführbar sein. So kann man das gesamte Netzwerk aus Zielsystemen jederzeit in einen wohldefinierten Ausgangszustand bringen.

Gefordert wird:

- Die Möglichkeit, ein Zielsystem **schrittweise zu Debuggen**
- Eine **Integration** in eine IDE
- **Gleichzeitiger Start/Reset** mehrerer Zielsysteme

2.3.2. Deployment

Für das *Deployment* des Zielystems ist es erforderlich, die einzelnen Knoten des Zielsystems mit einer **neuen Firmware beschreiben** zu können. Dies geschieht sowohl testweise und zu Debuggingzwecken während der Entwicklung als auch für die endgültige Fertigstellung.

Da es sich um eine Vielzahl identischer Zielsysteme handelt, sollte sich dieser Prozess im Idealfall auf eine beliebige Anzahl an Zielsystemen ausweiten lassen. Es wäre wünschenswert, eine Anzahl von Komponenten **in einem einzigen Schritt** mit einer Firmware beschreiben zu können.

2.3.3. Datenanalyse

Die Möglichkeit ein Entwicklungssystem beliebige Daten als „Debuginformationen“ ausgeben zu lassen erlaubt es, eine Übersicht über die zeitlichen Abläufe zu erstellen. Gerade für die Analyse der Abläufe miteinander interagierender Zielsysteme ist es wichtig, diese Informationen zentral zu sammeln und zu sortieren, um sie später zur Fehlersuche heranziehen zu können.

Zur Umsetzung dieses Systems müssen also zwei Komponenten erstellt werden:

1. Eine Software zur Anzeige und Organisation der, von mehreren Zielsystemen gesammelten, Daten
2. Eine Möglichkeit die Daten mehrerer Module zu sammeln

3. Konzeption des verteilten Entwicklungssystems

Inhaltsangabe

In diesem Kapitel soll das Entwicklungssystem konzipiert werden.

Hierfür muss anfänglich ein grundsätzliches Entwicklungskonzept festgelegt werden. Aus diesem Konzept ergeben sich dann die Voraussetzungen, welche die zu wählende Hardware erfüllen muss.

Das Entwicklungssystem soll außerdem dazu verwendet werden, ein beispielhaftes Zielsystem entwickeln zu können. Auch über die Hardware dieses Zielsystems muss eine Entscheidung getroffen werden, die möglichst repräsentativ für eine Vielzahl möglicher Zielsysteme ist.

Letztendlich muss sichergestellt werden, dass alle einzelnen Komponenten in dem zu planenden System miteinander interagieren. Es müssen Schnittstellenspezifikationen festgelegt werden.

3.1. Grundlegende Konzeption

Der Umstand, dass die einzelnen Zielsysteme räumlich von einander und dem Entwicklersystem getrennt sind, erfordert gewisse Änderungen an den Entwicklungsabläufen.

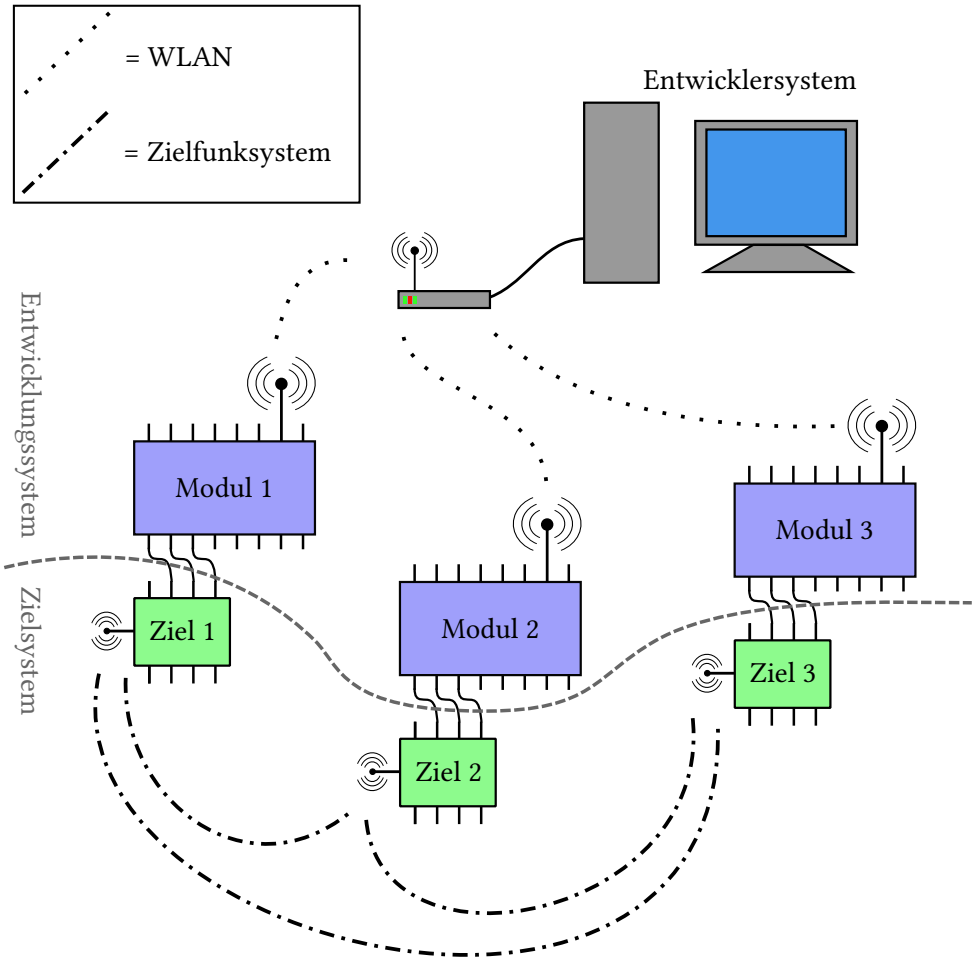
Ein direkter Zugriff auf die einzelnen Zielsysteme ist nicht ohne Weiteres möglich, beziehungsweise vergleichsweise umständlich. Würde man die in Kapitel 2 aufgezählten Abläufe für ein verteiltes System umsetzen wollen, müsste man die Zielsysteme zum Beispiel neben das Entwicklersystem legen, da für viele Vorgänge eine **physische Verbindung** bestehen muss.

So benötigt zum Beispiel das Debugging über JTAG oder SWD eine Verbindung zwischen einem Adapter und dem Zielsystem sowie zwischen Entwicklersystem und Adapter. Sollen beliebige Daten vom Zielsystem erfasst werden, so wird auch hierfür oft eine Verbindung in Form von direkter Verkabelung benötigt.

Die Auswahl existierender Netzwerk-Debugging-Lösungen ist geradezu minimal. Als eigenständige Lösung findet sich hier zum Beispiel *ZY1000 (Ultimate Solutions Inc.)*[9] oder *WiFi-Demon (Macraigor Systems)*[10]. Beide Systeme stellen eine Schnittstelle zwischen JTAG und Ethernet oder WLAN dar und sind somit bereits existierende Teilumsetzungen des geplanten *Entwicklungssystems*.

Beide Lösungen sind jedoch auch geschlossene Systeme. Wollte man also, wie vorgesehen, zusätzliche Daten sammeln, müsste man ein weiteres Gerät (mit eigenem Anschluss an das Netzwerk) verwenden.

Der Ansatz vom in Abschnitt 1.2 erwähnten *TinyOS*, das Funknetzwerk der Zielsysteme selbst für das Deployment von Firmware oder das Analysieren von internen Vorgängen zu nutzen, hat den Nachteil, dass dafür ein solches Netzwerk bereits bestehen muss. Dies kann jedoch nicht immer garantiert werden. Insbesondere nicht dann, wenn erwähntes Funknetzwerk erst noch entwickelt werden muss.



Die “Module“ werden über WLAN vom Entwicklersystem gesteuert und übermitteln an dieses auch alle Daten, die sie von ihrem jeweils angeschlossenen „Ziel“ erhalten.

Abbildung 3.1. Das geplante Gesamtsystem

Wünschenswert wäre es also, ein von dem Zielsystem und dessen Hardware völlig unabhängiges System zu entwickeln. Dies hat gleichzeitig den Vorteil, dass es bezüglich der Entscheidungen über ein Zielsystem **größtmögliche Flexibilität** bietet.

Der Grundgedanke hinter dem zu entwerfenden Entwicklungssystem ist deshalb der, dass bereits bestehende Infrastruktur wie Ethernet oder WLAN für die Verbindung von „Modulen“ zum Entwicklersystem genutzt wird. Die „Module“ übernehmen dann die Funktion einer Art „Daten- und Debuggingbrücke“. Sie ermöglichen die entfernte Steuerung der Zielsysteme und

die Erfassung von Daten, die anschließend auf dem System des Entwicklers verfügbar gemacht und zusammengefasst werden.

In Abbildung 3.1 ist dargestellt, wie die eingezeichneten „Module“ über WLAN in die bestehende Infrastruktur, symbolisiert durch einen Access-Point, eingegliedert werden. Auch eine Verbindung über Ethernet wäre hierbei denkbar.

Die einzelnen „Module“ sind in der Lage, die Ein- und Ausgänge der Zielsysteme („Ziele“) direkt zu erfassen und zu manipulieren. Dies ermöglicht es, Daten zu sammeln und weiterzuleiten, sowie auf die „Ziele“ zuzugreifen.

3.2. Vorentscheidungen

In diesem Abschnitt sollen die Vorgaben für das Entwicklungssystem und ein beispielhaftes Zielsystem getroffen und erläutert werden.

3.2.1. Entwicklungssystem

Für das *Entwicklungssystem* selbst muss eine Entscheidung über das verwendete Modul getroffen werden, das als „Brücke“ zwischen Entwicklersystem und Zielsystem funktioniert.

Unter den eingebetteten Betriebssystemen hat unter anderem **Linux** eine hohe Verbreitung. Es existieren verschiedene Distributionen, die speziell für diese Einsatzzwecke ausgelegt sind.

Zusätzlich bietet Linux eine hohe Konfigurierbarkeit und Abstraktion von der zugrunde liegenden Hardware. So kann eine für Linux entwickelte Software meist mit relativ geringem Aufwand auf unterschiedlichste Plattformen portiert werden. Im Idealfall leistet die verwendete Linux Distribution hierbei Hilfe.

Grundsätzliche Anforderung an die zu verwendende Plattform ist, dass sie mindestens eine Ethernet- und im Idealfall auch eine WLAN-Schnittstelle bietet. Außerdem sollten möglichst viele Möglichkeiten zur Kommunikation mit dem Zielsystem gegeben sein. Dazu zählen zum Beispiel Schnittstellen wie USB, Universal Asynchronous Receiver Transmitter (UART), I^2C oder Serial Peripheral Interface Bus (SPI).

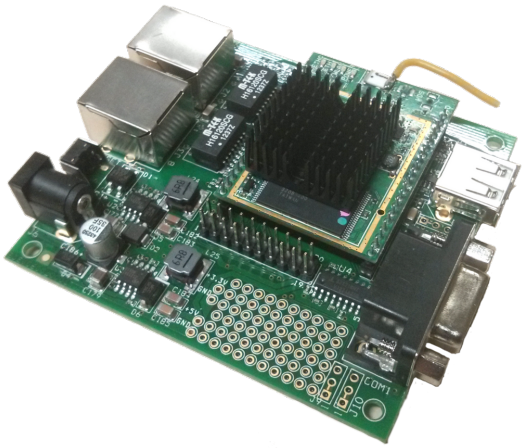


Abbildung 3.2. Fotografie des Entwicklungsboards mit aufgestecktem Carambola

Das Entwicklungsmodul **Carambola**[11] wird von der der Firma UAB „8devices“ hergestellt und besteht aus einer kleinen Platine mit einem Ralink RT3050[12] Mikrocontroller (MIPS-Architektur).

Der Mikrocontroller besitzt folgende Funktionen:

- Ralink RT3050 320 MHz MIPS SoC
- 5-Port Ethernetswitch
- 802.11n 2.4 GHz WLAN
- I^2C, SPI, UART

4 LEDs, eine WLAN-Antenne und ein 32 MB Flash-Baustein sind dabei bereits auf das Board integriert, die restlichen Anschlüsse sind über eine Steckerleiste verfügbar.

Zusätzlich bietet „8devices“ ein Entwicklungsboard an, auf das sich das Carambola direkt aufstecken lässt. So können viele der vom Mikrocontroler gebotenen Anschlüsse direkt verwendet werden. Das Entwicklungsboard besitzt unter anderem zwei Ethernet-Anschlüsse, einen RS232-Anschluss, einen USB-Anschluss und einen kleinen Bereich für eigene Elektronikaufbauten.

Das Carambola wird mit **OpenWRT**, einem linuxbasierten Betriebssystem für eingebettete Geräte, betrieben. Dieses wird in Unterabschnitt 3.3.3 näher beschrieben.

Da das Carambola eine MIPS-Architektur besitzt, muss ein Cross-Compiler eingesetzt werden, um Software für das Carambola erstellen zu können.

Definition 3.1 (Cross-Compiler) *Ein Cross-Compiler ist ein spezieller Compiler, der den Quellcode einer Software für eine andere als die ausführende Architektur kompilieren kann. Er kommt dann zum Einsatz, wenn es nicht praktikabel oder schlicht unmöglich ist, die Anwendung auf dem Zielsystem zu kompilieren. Dies kann zum Beispiel dann der Fall sein, wenn ein eingebettetes System über nicht ausreichend Ressourcen (Speicherplatz) verfügt, um den Kompilationsvorgang durchzuführen.*

3.2.2. Zielsystem

Das zu entwerfende Entwicklungssystem sollte ein möglichst großes Spektrum potentieller Architekturen und Funksysteme abdecken können, so dass es später einfach ist, das exemplarisch zu entwickelnde Zielsystem durch ein anderes zu ersetzen. Dies sollte sich in einer möglichst **generischen** Wahl der Komponenten eines beispielhaften Zielsystems widerspiegeln.

Da Mikroprozessoren mit **ARM-Architektur** einen Anteil von rund 71%[13] an allen verkauften CPUs haben, scheint es beinahe selbstverständlich, sich für eine solche CPU zu entscheiden.

Bedingt durch die hohe Verbreitung von ARM Rechenkernen existiert allerdings auch eine große Anzahl an Entwicklungsboards, die in Frage kommen.

Dabei ist es wichtig, eine Entscheidung eher nach Kriterien der Energieeffizienz als der Rechenleistung zu treffen, da das Zielsystem später stark integriert und unter Umständen auch batteriebetrieben arbeiten können soll.

Wählt man einen ARM-Kern nach der Energieeffizienz aus, fällt einem hierbei die Reihe der Cortex-M Prozessoren auf. Sie liegen mit ihrem geringen Energiebedarf in einem Bereich, der für Batteriebetrieb sehr gut geeignet ist. Aus diesem Grund optimieren viele Chiphersteller, die Cortex-M Kerne verbauen, diese genau dafür.

Das Entwicklungsboard „STM32-H152“[14] des Herstellers *Olimex Ltd.* besitzt mit dem STM32L152VBT6 der Firma *STMicroelectronics* genau einen solchen energieeffizienten ARM-Mikrocontroller.

Auch das Board ist speziell auf den Batteriebetrieb ausgelegt und besitzt neben einem USB-Anschluss für Stromversorgung und Datenübertragung eine 20-polige Buchse zum Anschluss

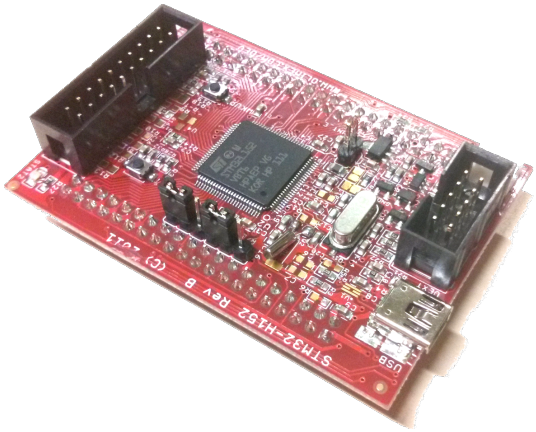


Abbildung 3.3. Fotografie des STM32-H152

eines JTAG-Debuggers und einen Schaltkreis zum Laden von Lithium-Ionen/Lithium-Polymer Batterien.

Außerdem befindet sich auf dem Board ein, bei *Olimex*-Produkten häufig eingesetzter, 10-poliger Anschluss. Für diesen „UEXT“ getauften Anschluss bietet *Olimex* eine Reihe von Erweiterungsboards an, die sich so einfach verbinden lassen. Dabei spezifiziert Olimex lediglich die Zuordnung verschiedener Pins zu Funktionen, die auf den meisten Mikrocontroller vorhanden sind (SPI, I^2C und UART).

3.2.3. Funkmodul

Die auf dem Markt verfügbaren Funkmodule lassen sich grundsätzlich in zwei Kategorien einteilen.

Jedes Funkmodul besteht mindestens aus einem **Transceiver**. Dieser Transceiver umfasst zumindest die analogen Komponenten der physikalischen Schicht wie Analog-Digital-Wandler, Digital-Analog-Wandler, Filter, Verstärker, Frequenzerzeuger. Je nach Ausführung werden, ahängig davon ob der Transceiver für ein spezifisches Protokoll eingesetzt werden soll, auch die Bauteile des Medium Access Control (MAC)-Layers (Checksummenprüfung, Buffer, Kollisionsvermeidung) in einen Transceiver integriert.

Für viele Funkprotokolle stehen jedoch auch **Single-Chip Solutions** zur Verfügung. Diese Module bestehen sowohl aus einem Transceiver als auch, neben anderen Bauteilen, aus einem

eigenen Mikrocontroller. Dieser übernimmt die Aufgaben höherer Netzwerkschichten und wird dafür oft über einfache Protokolle wie UART oder SPI angesprochen.

Dies bedeutet jedoch gleichzeitig mehr Hardware, da ein integrierter Mikroprozessor betrieben werden muss und somit Strom benötigt. Außerdem verhindert die hohe Abstraktion der Kommunikation genauere Einblicke in deren Abläufe.

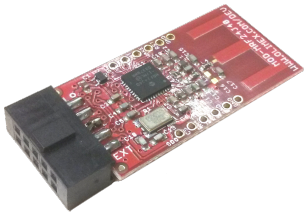


Abbildung 3.4. Fotografie des MRF24J40

Da es jedoch auch möglich sein soll, die Abläufe des Funkprotokolls erfassen und visualisieren zu können, sollte der Chip nicht zu abstrahiert funktionieren. Die Wahl muss also zwangsläufig auf einen **Transceiver** fallen.

Da, wie zuvor erwähnt, eine Vielzahl verschiedenster Module für den „UEXT“ Anschluss existieren, gibt es hierfür auch ein Modul mit einem Transceiver.

Das Modul „MOD-MRF24J40“ verwendet den Chip MRF24J40[15] der Firma Microchip Technology Inc. Der Chip ist eine IEEE 802.15.4[16] kompatible Realisierung eines Funktransceivers und implementiert die im Standard spezifizierten PHY- und MAC-Layer.

Das Modul enthält eine integrierte Antenne und stellt die vom Chip zur Datenkommunikation genutzte SPI-Schnittstelle über den „UEXT“-Stecker zur Verfügung.

3.3. Debugging

3.3.1. Schnittstellen

Das Debugging eines eingebetteten Systems erfordert meist eine kombinierte Hard- und Softwarelösung.

Der Softwareteil ist oft in die IDE des Entwicklers eingebunden oder lässt sich zumindest vom Arbeitsplatz des Entwicklers aus steuern. Die Befehle des Entwicklers (Setzen eines

Breakpoints, Abfrage von Registerwerten, etc.) werden vom Softwaretreiber anschließend in Steuerbefehle des Hardwarebauteils umgesetzt. Das Hardwaremodul übernimmt hierauf, unter Nutzung eine standardisierten Schnittstelle, die Kommunikation mit dem Zielsystem.

Als *Schnittstellen* sind für diese Zwecke vor allem JTAG und SWD verbreitet. Während JTAG ursprünglich primär zum Testen von Bauteilen konzipiert wurde, erweitert SWD dieses Konzept um speziell für das Debuggen nützliche Funktionen (Tracing, Paritätschecks) und reduziert die Anzahl zwingend notwendiger Pins von 4 auf 2. Da SWD aber von ARM speziell für Prozessoren mit eingebautem *CoreSight*-Modul entwickelt wurde, ist Unterstützung hierfür nicht auf allen ARM-Controllern anzutreffen. JTAG hat hier eine wesentlich größere Verbreitung und wird auch architekturübergreifend genutzt. Die **Wahl von JTAG** würde eine spätere Erweiterung auf andere Architekturen somit erleichtern.

3.3.2. OpenOCD - Open On-Chip Debugger

Betrachtet man die Möglichkeiten zum Debuggen von ARM-Mikrocontrollern via JTAG unter einem Linuxsystem, landet man beinahe zwangsläufig beim Open Source Projekt **OpenOCD**. Dieses Projekt ist im Rahmen einer Diplomarbeit[17] an der *FH Augsburg* entstanden und hatte die Entwicklung einer quelloffenen Software zum Debuggen von ARM-Mikrocontrollern zum Ziel. *OpenOCD* fungiert dabei als Schnittstelle zwischen einem JTAG-Debugger und einer Benutzeroberfläche.

OpenOCD ist hochgradig modular gehalten und unterstützt daher eine große Bandbreite von JTAG-Adaptern, verschiedenste Mikrocontroller und Architekturen (unter anderem *ARMv7*, *ARMv4*, *AVR32*, *MIPS*), einige Echtzeitbetriebssysteme (*FreeRTOS*, *ChibiOS*, *ThreadX*) und diverse Flash-Speicher.

Die Konfiguration von *OpenOCD* erfolgt dabei über simple Skripte, die vom Hauptprogramm beim Start eingelesen werden.

Auch der auf dem Olimex-Board verwendete Mikrocontroller „STM32L152VB“ wird von *OpenOCD* als „STM32L“ unterstützt.

OpenOCD lässt sich dabei entweder über eine eigene Benutzeroberfläche via Telnet-Protokoll oder mittels eines GNU Debugger (GDB)-Clients bedienen.

Definition 3.2 (GDB) *Der GDB ist der Standard-Debugger unter Unix-Betriebssystemen. Er bietet unter Anderem die Möglichkeit, ein Programm im laufenden Betrieb anzuhalten, Breakpoints zu setzen oder Single-Stepping durchzuführen. Der GDB besitzt keine grafische Oberfläche, sondern lässt sich über eine Kommandozeile bedienen.*

Der entscheidende Punkt hierbei ist, dass GDB (und auch *OpenOCD*) grundsätzlich einen „Remote"-Modus anbietet. Dies wird dadurch möglich, dass GDB in einen „Server" und einen „Client" unterteilt ist. Dabei übernimmt der „Server" die Kontrolle des Zielsystems oder der Zielsoftware, wird aber gleichzeitig selbst vom GDB-„Client" gesteuert. Der Entwickler bedient dann den Client und damit nur indirekt den Debugger selbst.

Verwendet man GDB im „Remote"-Modus, kann diese Steuerung über einen beliebigen TCP-Port und damit auch über das Netzwerk erfolgen.

Setzt man *OpenOCD* nun also auf dem Carambola ein, lässt sich diese Trennung ausnutzen, um ein Debugging über eine beliebige Netzwerkverbindung zu realisieren.

3.3.3. OpenWRT

OpenWRT ist ein ursprünglich aus dem von Linksys zur Verfügung gestellten Quellcode des Routers „WRT54G" [18] hervorgegangenes Opensource Projekt.

Ziel dieses Projektes ist die Entwicklung eines Betriebsystems, das sich über ein Paketverwaltungssystem beliebig anpassen und dessen Dateisystem sich beliebig beschreiben lässt. Dies soll es dem Nutzer ermöglichen, das System von nicht benötigten Funktionen zu befreien, beziehungsweise auf seine Bedürfnisse zuzuschneiden. Aus diesem Grund verfügt *OpenWRT* über eine große Anzahl für dieses System bereits portierter Bibliotheken und Anwendungen.

OpenWRT wird hauptsächlich als Firmware auf Routern diverser Hersteller angeboten. Es zeichnet sich durch eine hohe Konfigurierbarkeit bei gleichzeitiger Unterstützung verschiedenster Plattformen aus. So werden neben MIPS zum Beispiel noch ARM, PowerPC und x86 unterstützt.

Paketverwaltung und Kompiliervorgang

OpenWRT besitzt mit **opkg** ein Tool zur Paketverwaltung von Anwendungen. Hierbei kann die Software in Form von *.ipk-Files aus „Repositories“ installiert werden. Durch dieses Tool wird es möglich, das eingebettete System für die eigenen Bedürfnisse anzupassen.

Definition 3.3 (Repository) *Ein* Repository *enthält kompilierte Anwendungen für ein bestimmtes Betriebssystem und eine spezifische Architektur. Über eine Paketverwaltungssoftware kann auf das Repository zugegriffen, Software heruntergeladen und installiert werden. Abhängigkeiten gegenüber anderen Softwarepaketen werden meist selbstständig erkannt und zusätzlich installiert.*

Die Erstellung neuer Pakete für OpenWRT wird durch stark modifizierte Makefiles wesentlich vereinfacht.

Definition 3.4 (Makefile) *Ein* Makefile *ist eine Datei, die Steuerbefehle für das Tool* `make` *enthält.* `make` *wird überwiegend eingesetzt, um Quellcode in ausführbare Programme und Bibliotheken zu übersetzen. Durch seinen hohen Grad an Abstraktion können* Makefiles *jedoch auch für andere Aufgaben „zweckentfremdet“ werden.*

OpenWRT liefert alle nötigen Bestandteile mit, um ein vollständiges Image für das eingebettete System zu erstellen. Der Ablauf ist dabei wie folgt:

- Mittels des Befehls `make menuconfig` lässt sich der folgende Kompiliervorgang in einer grafischen Oberfläche anpassen. Dazu zählen die zu erstellenden Anwendungen, Konfigurationsparameter des Kernels und Voreinstellungen der Anwendungen.
- Durch das Eingeben von `make` wird der Kompiliervorgang gestartet.
- Der Quellcode der Toolchain wird heruntergeladen und kompiliert.
- Der Quellcode des Basissystems wird heruntergeladen und kompiliert.
- Die einzelnen Pakete werden erstellt:
  - Herunterladen des Quellcodes
  - Anwenden von nötigen OpenWRT-Patches
  - Kompilieren der Anwendung
  - Erstellen der *.ipk-Datei

Soll *OpenOCD* nun auf diese Plattform portiert werden, ist es wünschenswert, das Programm direkt in diesen Prozess des Paketverwaltungssystems einzubinden. Hierfür muss ein geeignetes *Makefile* erstellt werden.

Diese hohe Abstraktion hat gleichzeitig den Vorteil, dass statt des Carambolas jede beliebige Plattform verwendet werden kann, die von OpenWRT unterstützt wird.

3.3.4. Wahl eines JTAG-Adapters

OpenOCD unterstützt eine große Anzahl verschiedener *JTAG-Adapter*. Viele dieser Adapter haben dabei jedoch im Grunde ähnliche Komponenten verbaut.

Neben den industriellen Lösungen mit speziell entwickelten Bauteilen (Segger J-Link[19], ST Micro ST-Link[20]), existieren auch viele Alternativen, die bereits existierende Bauteile verwenden. So kommt zum Beispiel oft der *FT2232*-Chip zum Einsatz, der auf die Umsetzung von USB-Daten zu einem seriellen bzw. parallelen Ausgang spezialisiert ist. Neben kommerziellen Adaptern (Amontec JTAGkey2[21], Olimex ARM-USB-TINY-H[22]) existieren auch einige Open Source Lösungen, in denen der *FT2232* zum Einsatz kommt (Turtelizer 2[23], OOCDLink[24]). Eine Auflistung aller mit *OpenOCD* kompatiblen Adapter findet sich in dessen „User Guide“[25].

Entscheidend für die Wahl eines Adapters sollte zum einen die Anzahl der unterstützten Zielsysteme sein. Diese definiert sich hauptsächlich über die vom Adapter unterstützte **Spannung**. Da viele Mikrocontroller mit $3.3\,\mathrm{V}$ oder $5.5\,\mathrm{V}$ betrieben werden, sollte der Adapter zumindest diese beiden Spannungen unterstützen. Je größer diese Spanne jedoch ist, desto flexibler kann er später eingesetzt werden.

Auch unterscheiden die Adapter sich im Vorhandensein von **adaptivem Clocking**. Diese Funktion erlaubt es dem Adapter, sich an die aktuelle Taktfrequenz des Zielsystems anzupassen. Da viele Mikrocontroller direkt nach dem Start oder in einem Deep-Sleep-Zustand mit einer geringen Taktzahl arbeiten und die aktuelle Taktzahl des Zielsystems oftmals die maximal mögliche JTAG-Taktzahl vorgibt, müsste der Adapter anderenfalls unter Umständen langsamer laufen. *Adaptives Clocking* erlaubt es dem Zielsystem, eine Art „Rückmeldung“ über die, aktuell gültige, maximale JTAG-Taktzahl zu geben.

Im Rahmen dieser Arbeit soll der **ARM-USB-TINY-H** von Olimex zum Einsatz kommen. Dieser Adapter besitzt einen der verbreiteten *FT2232*-Chips, bietet Adaptives Clocking und unterstützt einen Spannungsbereich von $2.0\,\mathrm{V}$ bis $5.0\,\mathrm{V}$.

3.3.5. Zielsetzung

Folgende Punkte sollen im Rahmen dieser Arbeit für ein funktionierendes Debugsystem umgesetzt werden:

- OpenWRT muss **kompiliert** und konfiguriert werden werden.
- Für *OpenOCD* muss ein **Makefile** erstellt werden, mithilfe dessen über den (Cross-)Kompiliervorgang eine Paketdatei für OpenWRT erstellt wird.
- *OpenOCD* muss so **(vor-)konfiguriert** werden, dass sich das Zielsystem damit debuggen lässt.
- Eine **IDE** muss so eingerichtet werden, dass sie *OpenOCD* über GDB als entfernte Debugging-Schnittstelle verwendet.

3.4. Datenanalyse

Ein weiterer wichtiger Bestandteil dieses Entwicklungssystems ist die *Analyse* von Daten, die das Zielsystem erzeugt. Da das Zielsystem über eine Funkschnittstelle verfügt, muss die für das Zielsystem zu entwickelnde Software diese Schnittstelle sicher auch in Betrieb nehmen können.

3.4.1. Anforderungen

Aufgabe dieses Teils des Entwicklungssystems wird es sein, die Erfassung der in Abschnitt 2.2 erwähnten Szenarien zu erleichtern.

Bedingung für die Erfassung von Daten seitens des Entwicklungssystems ist, dass das Zielsystem diese Daten überhaupt in einer erfassbaren Form zur Verfügung stellt. Hierbei muss jedoch beachtet werden, dass nicht jedes Zielsystem über alle Anschlussmöglichkeiten verfügen kann.

Um die Daten vom Zielsystem zu erfassen, bietet sich zum Beispiel die Nutzung des **UART** an. Hierfür muss jedoch die Software des Zielsystems so angepasst werden, dass es die zu sammelnden Daten über UART ausgibt.

Definition 3.5 (UART) *Der Universal Asynchronous Receiver Transmitter (UART) ist eine, in Mikrocontrollern sehr weit verbreitete, elektronische Schaltung zur seriellen Übertragung von Daten. In seiner Minimalkonfiguration benötigt der UART nur zwei Pins (RX — Receive, TX — Transmit), da, durch Festlegung auf einen gemeinsamen Takt, auf ein Taktsignal verzichtet werden kann. Dies ermöglicht eine Nutzung von UART auch in einem stark integriertem Zielsystem.*

Um die Daten später sortieren und analysieren zu können, müssen vom Entwicklungssystem mindestens folgende Daten erfasst werden.

- Vom Zielsystem **ausgegebene Daten**
- **Zeitpunkt** der Erfassung der Daten

Besonderes Augenmerk muss hierbei auf die Synchronizität der einzelnen Module des Entwicklungssystems gelegt werden. Auf diesen Umstand wird in Unterabschnitt 3.4.4 weiter eingegangen.

3.4.2. Erforderliche Präzision

Für viele Vorgänge der Funkschnittstelle ist eine gewisse Genauigkeit wichtig. Diese drückt sich insbesondere in der Genauigkeit der vom Entwicklungssystem verwendeten Zeitstempel aus.

Eine Möglichkeit die erforderliche Präzision analysieren zu können ist die Untersuchung des Kollisionsvermeidungs-Algorithmus.

Hierfür setzt der, in unserem Funkmodul verwendete, IEEE 802.15.4 Standard in seinem MAC-Layer zum Beispiel CSMA/CA ein[16].

Definition 3.6 (CSMA/CA) CSMA/CA *ist ein Algorithmus zur Kollisionsvermeidung innerhalb eines Netzwerks. Hierbei prüft der Sender vor dem eigentlichen Sendevorgang über einen definierten Zeitraum, ob das Übertragungsmedium (zum Beispiel Funk oder Kabel) „frei" ist. Wird das Medium bereits verwendet, wartet der Sender eine zufällige Zeitperiode (Back-off Periode) und überprüft den Zustand erneut. Sobald der Sendekanal „frei" ist, beginnt der Sendevorgang. Der Empfang der Daten muss von der Gegenstation mittels* `ACK` *bestätigt werden.*

Da *CSMA/CA* jedoch im MAC-Layer stattfindet und dieser bereits in das gewählte Funkmodul integriert ist, wird es vermutlich schwierig, die Daten aus dem Modul selbst zu beziehen. Hier könnte man sich durch die Analyse der empfangenen Daten behelfen.

Durch das integrierte *CSMA/CA* lässt sich zwar der Versandzeitpunkt nicht exakt bestimmen, jedoch aber der Empfangszeitpunkt. Untersucht man nun die Differenz zwischen Sende- und Empfangsvorgang, lässt sich unter Umständen erkennen, ob ein Backoff und damit eine erfolgreiche Kollisionsvermeidung erfolgt ist.

Folgende Rechnung nach[26] erläutert die Dauer einer Backoff-Periode.

$$\begin{aligned} InitialbackoffPeriod &= (2^{BE} - 1) * aUnitBackoffPeriod \\ &= (2^3 - 1) * 320\,\mu s \\ &= 2240\,\mu s = 2.24\,ms \end{aligned} \tag{3.1}$$

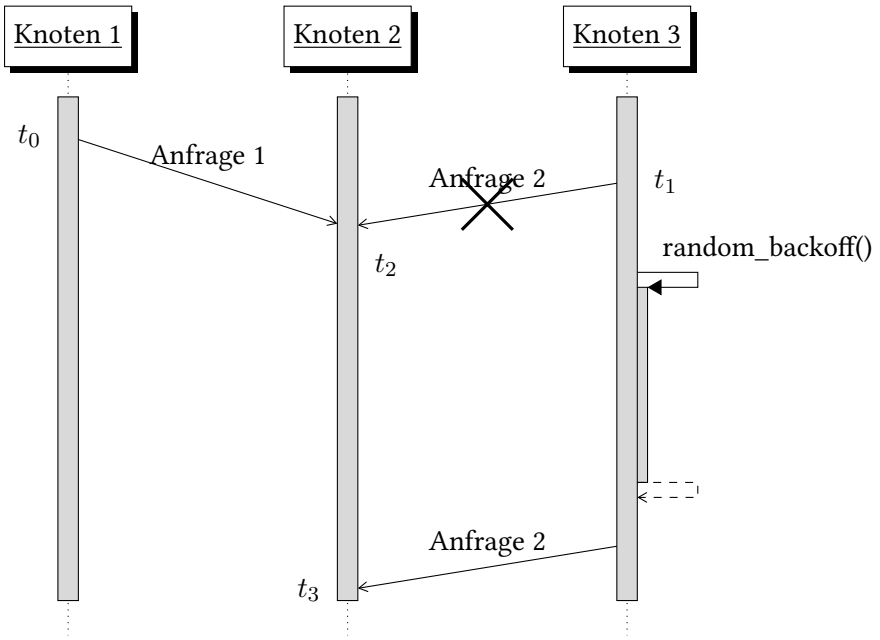
Vor jedem Sendevorgang wird nach Spezifikation des *CSMA/CA* ein zufälliger Zeitraum *InitialbackoffPeriod* gewartet. Erst anschließend wird der Kanal auf Belegung überprüft. Diese *InitialbackoffPeriod* berechnet sich nach Gleichung 3.1.

Zu dieser Backoff-Periode kommt die anschließende Prüfung auf Belegung (Clear Channel Assessment (CCA)) mit einer Dauer von 8 Symbol-Perioden (128 µs).

Im 2.4 GHz-Band besteht ein Symbol aus 4 Bit. Eine Symbol-Periode benötigt also, bei einer Übertragungsgeschwindigkeit von 250 kbit s^{-1}, eine Dauer von 16 µs. ($\frac{4\,bit}{250\,kbit\,s^{-1}} = 16\,\mu s$)

aUnitBackoffPeriod ist in der Spezifikation mit 20 Symbol-Perioden also 320 µs definiert. Zu Beginn ist *BE*, der Backoff-Exponent, standardmäßig auf 3 eingestellt. Dieser Zähler erhöht sich jedoch mit jedem Mal, den ein CCA fehlschlägt.

Selbst für einen erfolgreichen Sendevorgang vergehen also bereits bis zu 2.24 ms. Schlägt das CCA fehl, wird die Gleichung 3.1 mit einem um 1 erhöhten *BE* wiederholt und erneut gewartet. Hierdurch kann ein Datenpaket in der von der IEEE vorgegebenen Standardkonfiguration um bis zu 9.92 ms pro CCA-Versuch verspätet werden ($(2^5 - 1) * 320\,\mu s = 9920\,\mu s$ mit einem maximalen *BE* von 5).



Da Knoten 3 vor seinem ersten Sendevorgang erkennt, dass bereits Daten übertragen werden, muss er einen zufälligen Zeitraum warten, bevor ein Übertragungsversuch erfolgen kann.

Abbildung 3.5. Gleichzeitiger Sendevorgang mit Kollisionvermeidung

Abbildung 3.5 visualisiert einen Sendevorgang mit einspringender Kollisionsvermeidung. Zu Zeitpunkt t_0 beginnt *Knoten 1* mit dem Senden von Daten. *Knoten 3* führt bei t_1 ein CCA durch und stellt fest, das bereits Daten gesendet werden. Seine Übertragung wird also um eine Backoff-Periode zurückgestellt. *Knoten 2* empfängt bei t_2 die Daten von *Knoten 1* und eine Weile später bei t_3 auch die Übertragung von *Knoten 3*.

Da, wie zuvor erwähnt, das CCA auf Hardware-Ebene erfolgt, lässt sich der tatsächliche Zeitpunkt des Absendens nicht herausfinden. Jedoch kann man anhand der Differenz zwischen t_0 und t_2 erkennen, ob der ursprüngliche Absendezeitpunkt t_1 von der *Anfrage 2* dazwischen liegt. Auch könnte man über die (vergleichbar große) Differenz zwischen t_1 und t_3 darauf schließen, dass die Übertragung aufgrund eines fehlgeschlagenen CCA verzögert wurde.

Will man diesen Ablauf zentralisiert erfassen, ergäbe sich im Optimalfall ein Ablauf wie in Abbildung 3.6. Zu Zeitpunkt t_1 würde *Knoten 3* anfänglich die Daten akzeptieren, jedoch hardwareseitig und durch ein fehlgeschlagenes CCA noch nicht direkt absenden.

Zielsystem	Zeitstempel	Nachricht
<Knoten 1>	t_0	Anfrage 1 gesendet
<Knoten 3>	t_1	Anfrage 2 gesendet
<Knoten 2>	t_2	Anfrage 1 erhalten
<Knoten 2>	t_3	Anfrage 2 erhalten

Ein Ablauf wie in Abbildung 3.5 dargestellt, könnte diese Reihenfolge von Nachrichten erzeugen.

Abbildung 3.6. Erfassung der Daten des Ablaufs aus Abbildung 3.5

Bei einer minimal spezifizierten Größe des Payloads eines IEEE 802.15.4-Paketes von 2 B[16] werden 672 µs zur Übertragung benötigt. Dies zeigt Gleichung 3.2. Hierbei sind der *overhead*-Anteil 19 B bestehend unter anderem aus Sequenznummer, Adressfeldern und Längenangabe aus PHY- und MAC-Layer.

$$\frac{(2\,\mathrm{B} + overhead) * 8}{250\,\mathrm{kbit\,s^{-1}}} = 672\,\mu\mathrm{s} \tag{3.2}$$

Will man nun die erwähnten Abläufe sinnvoll erfassen, muss die Präzision der Timer also bei **unter 1 ms** liegen. Erst diese Genauigkeit würde ein sinnvolles Sortieren und Zuordnen der Abläufe ermöglichen.

3.4.3. Bestandteile

Der Teil des Entwicklungssystems, der für die Analyse der gesammelten Daten zuständig ist, muss im Kern aus drei Teilen bestehen.

- **Server** — Läuft als Anwendung auf dem Carambola. Empfängt Daten über UART und leitet diese an den Client weiter.
- **Client** — Läuft auf dem System des Entwicklers. Verbindet sich zu der Serveranwendung und empfängt von ihr die gesammelten Daten.
- **Protokoll** — Definiert die Kommunikation zwischen Server und Client.

Für die *Clientanwendung* ist es von großer Bedeutung, dass sie sich zu mehreren Serverinstanzen verbinden kann. Nur hierdurch wird es möglich, die Daten von mehreren Zielsystemen erfassen und auswerten zu können.

Für die *Serveranwendung* ist es nicht unbedingt notwendig, aber wünschenswert, mehrere verbundene *Clients* verwalten zu können.

3.4.4. Zeitsynchronisation

Ein wichtiger Punkt im Entwurf dieses Systems ist die **Synchronisation der Uhren** der jeweiligen Module. Nur so kann eine Vergleichbarkeit der verarbeiteten Daten gewährleistet werden.

Aus Unterabschnitt 3.4.2 ergibt sich, dass eine Präzision von mindestens 1 ms gewährleistet werden sollte. Zu den nicht beeinflussbaren Größen zählen hierbei zum Beispiel die für einen Kontextwechsel durch den Scheduler des Betriebssystems nötige Zeit oder die Genauigkeit des Systemquarzes.

Da der zu übertragene Zeitstempel die, seit eines durch den Entwickler-Rechner festgelegten Zeitpunktes, vergangenen µs darstellt, muss diese Zahl ausreichend groß dimensioniert sein. Eine 32-Bit Zahl reicht unter Umständen für die Betrachtung längerer Zeitabstände nicht mehr aus ($2^{32}\mu s \approx 71\,\mathrm{min}$).

Eine **64-Bit** Zahl bleibt somit als einzige Möglichkeit. Sie bietet ausreichend viel Platz ($2^{64}\mu s \approx 584942\ years$), um auch über sehr große Zeiträume Daten aufzuzeichnen.

Ein erster Ansatz für den Synchronisationsvorgang wäre, ein bereits bestehendes Synchronisationsprotokoll wie Network Time Protocol (NTP) zu verwenden. Diese haben jedoch den Nachteil, dass sie für die Synchronisation immer eine aktive Verbindung zum Internet oder einen lokalen NTP-Server benötigen. Außerdem sind, da die Carambola-Systeme keine Echtzeituhren besitzen, Datum und Uhrzeit nach einem Start des Systems auf die Unix Epoche gesetzt. Da NTP direkt die Systemzeit beeinflusst, würde dies unter Umständen Auswirkungen auf andere Prozesse bedeuten.

Für die Zeitsynchronisation soll eine Abwandlung des „Algorithmus von Christian“[27] zum Einsatz kommen. Entscheidend ist hierbei, dass der Synchronisationsvorgang vom Zeitgeber angestoßen wird. Dies erfordert eine zusätzliche abschließende Datenübertragung.

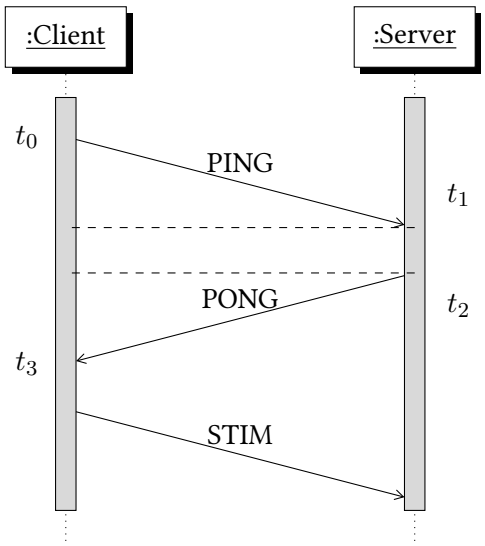


Abbildung 3.7. Ablauf der Zeitsynchronisation eines Servers

Da TCP durch seine Flusskontrolle die Messung der Netzwerklatenz durch eventuell Neuübertragungsversuche negativ beeinflussen kann, ist für die Zeitsynchronisation die Verwendung von UDP nötig. Dies bedingt allerdings die Verwendung einer Sequenznummer.

- Der Client stößt den Synchronisationsvorgang an, indem er dem Server seinen aktuellen Zeitstempel t_0 mittels einer `PING`-Nachricht übermittelt.
- Bei Erhalt der Nachricht erfasst der Server seinen eigenen Zeitstempel als t_1.
- Anschließend bereitet er das Senden einer `PONG`-Nachricht als Antwort vor, erfasst einen weiteren Zeitstempel (t_2) und sendet die Nachricht.
- Bei Erhalt der Nachricht durch den Client, erfasst dieser seine eigene Zeit t_3 und übermittelt sie dem Server in einer, diesen Synchronisierungsprozess abschließenden, `STIM`-Nachricht.

Dieser Vorgang ist in Abbildung 3.7 visualisiert.

Nach Abschluss dieser Kommunikation lässt sich die Zeitdifferenz vom Server durch folgende Formel errechnen.

$$\frac{t_1 - t_0 - t_3 + t_2}{2} = \Delta t \tag{3.3}$$

Um Abweichungen, zum Beispiel durch unterschiedliche Latenzen auf dem Hin- und Rückweg im Netzwerk, auszugleichen, sollten mehrere dieser Vorgänge durchgeführt und ein arithme-

tisches Mittel über die gesammelten Differenzen gebildet werden. Dieser Mittelwert ist eine etwas bessere Annäherung an die tatsächliche Differenz.

3.4.5. Protokoll

Insgesamt müssen für das Entwicklungssystem zwei *Protokolle* spezifiziert werden, da drei verschiedene Systeme eingesetzt werden.

Zielsystem — Serveranwendung

Für die Kommunikation zwischen dem Zielsystem und dem Carambola und damit der Serveranwendung wird UART als Schnittstelle eingesetzt.

UART ist relativ flexibel in seiner Umsetzung. So lassen sich diverse Paramter verändern (Baudrate, Stopbits, Paritätsbits). Dies bedeutet jedoch auch gleichzeitig, dass es keine einheitliche Lösung geben kann. Aus diesem Grund muss es möglich sein, diese Parameter unter Umständen an die Bedürfnisse des Zielsystems anpassen zu können.

Eine hohe Flexibilität soll auch in der Art der übertragenen Daten gewährleisten werden.

Es wird nur spezifiziert, dass über den UART-Anschluss gesendete Daten mit einem Zeilenumbruch enden müssen. Anschließend wird von der Serveranwendung ein Zeitstempel generiert und die Daten im System weitergereicht.

Dies erlaubt es dem Zielsystem, neben Debugausgaben des Funkmoduls, auch beliebige andere Nachrichten an das Entwicklersystem abzusetzen.

Serveranwendung — Entwicklersystem

Damit der Client und die Serveranwendungen über das Netzwerk kommunizieren können, wird ein Protokoll zum Datenaustausch spezifiziert. Dieses Protokoll legt sowohl den Aufbau der Datenpakete als auch die Abläufe der Kommunikation fest.

Während die Übertragung der Nutzdaten (vom Zielsystem ausgegebene Daten) über TCP erfolgt, läuft die Zeitsynchronisation über UDP ab. Die Verwendung von UDP garantiert, dass keine Mehrfachübertragung eines fehlerhaften Datenpaketes erfolgt, welche eine korrekte Zeitmessung verhindern würde.

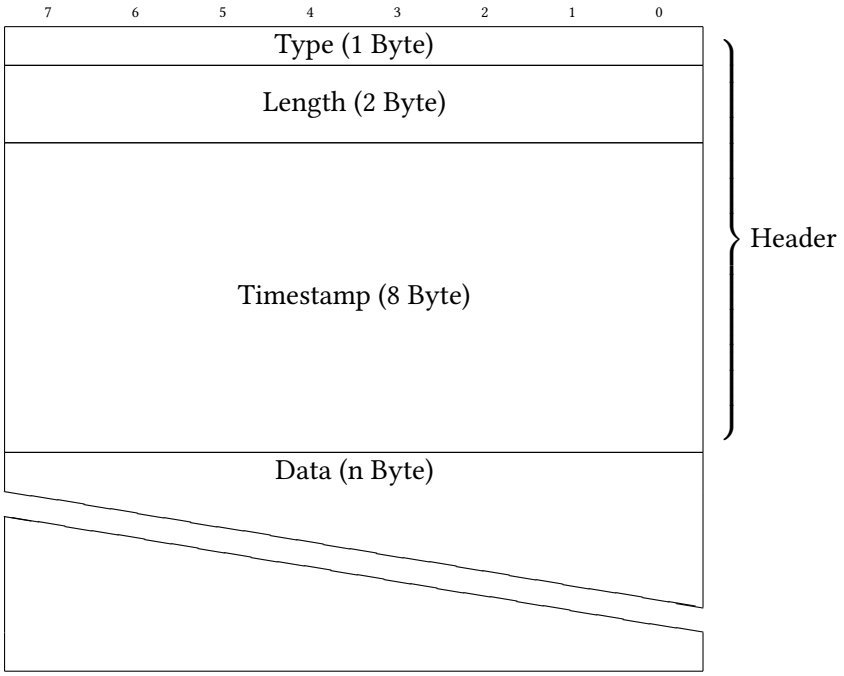
Für die Übertragung der Nutzdaten ist aber zu entscheiden ob der Nagle Algorithmus[28] deaktiviert werden muss, da die Daten ansonsten unter Umständen zu stark verzögert werden.

Um das Protokoll erweiterbar zu halten, sollte es eine Möglichkeit geben, das System später um andere Nutzdatentypen wie zum Beispiel Ausgaben einer SPI oder I^2C-Schnittstelle erweitern zu können.

Das Protokoll soll folgende Funktionen erlauben:

- **Zeitsynchronisation** wie in Unterabschnitt 3.4.4 aufgeführt
- Übertragung von **UART-Daten**
- **Erweiterungsmöglichkeit** für andere Datentypen

Jedes Datenpaket besteht dafür aus einem **Pakettyp**, der **Länge** (Anzahl Byte) der Nutzdaten, einem **Zeitstempel** und optional den **Nutzdaten** selbst. Die Übertragung erfolgt dabei, wie für die meisten Netzwerkprotokolle üblich, in Big-Endian. Der Aufbau eines Datenpaketes ist in Abbildung 3.8 visualisiert.



Der Aufbau eines TCP-Paketes ist mit dem eines UDP-Paketes identisch. Die Menge *n* der Daten ist auf 512 Byte beschränkt.

Abbildung 3.8. Systematischer Aufbau eines Datenpaketes

Die Größe des Headers beträgt 11 B und setzt sich wie folgt zusammen:
Für die Bestimmung des **Typs** eines Datenpaketes reichen 8 bit. Dies ermöglicht 256 verschiedene Pakettypen und somit ausreichend Spielraum für potentielle Erweiterungen. Die Größe des **Zeitstempels** von 64 bit ist in Unterabschnitt 3.4.4 erläutert worden. Die maximale Größe der an ein Datenpaket anhängbaren Nutzdaten ist auf 512 B beschränkt. Dies erscheint ausreichend groß für „kurze“ Statusnachrichten. Hieraus ergibt sich, dass für die Angabe der **Länge** 16 bit benötigt werden.

Aus dem Ablauf in Unterabschnitt 3.4.4 ist ersichtlich, dass bei den UDP-Nachrichten einzig für `PING` und für `STIM` eine Übermittlung des Zeitstempels wichtig ist, gleichzeitig allerdings auch eine Sequenznummer benötigt wird. Verlagert man den zu übertragenden Zeitstempel der `STIM`-Nachricht in den Bereich der *Nutzdaten*, lässt sich das *Zeitstempel*-Feld im UDP-Ablauf durchgehend als Sequenznummer verwenden. Die erste `PING`-Nachricht übermittelt somit indirekt auch die für den weiteren Synchronisierungsablauf notwendige Sequenznummer in Form ihres eigenen Zeitstempels.

In Abbildung 3.9 sind die verschiedenen TCP- und UDP-Pakettypen aufgeführt.

3.5. Deployment

Die zum *Debuggen* bereits bestehende JTAG-Verbindung lässt sich ebenso verwenden, um Software auf das Zielsystem zu bringen.

Grundvoraussetzung ist hierfür jedoch, dass sich das ROM des Zielsystems über den TAP erreichen lässt. Dies ist bei dem verwendeten Cortex M3 (und vielen anderen Mikrocontrollern) der Fall.

3.5.1. Zielsetzung

Es muss eine Möglichkeit gefunden werden, die Verwendung eines GDB-Clients zu automatisieren. Dadurch könnte man ein und dieselbe Firmware auf eine beliebige Anzahl Zielsysteme verteilen.

Existiert diese Möglichkeit, kann der gleiche Weg genommen werden, um diese Zielsysteme gesammelt resetten und somit in einen definierten Augangszustand bringen zu können.

Server zu Client

Protokoll	Abkürzung	Wert	Beschreibung
TCP	MESS	`0x01`	Allgemeine Nachricht des Servers. Eventuell als Broadcast verursacht durch MESS von einem Client.
	UART	`0x02`	Der Server hat eine UART-Zeile empfangen. Die empfangenen Daten sind als Payload angehängt.
	SPID	`0x03`	Der Server hat eine SPI-Zeile empfangen. Die empfangenen Daten sind als Payload angehängt.
	PONG	`0x06`	Antwort des Servers auf TCP-PING vom Client.
UDP	PONG	`0x06`	Antwort des Servers auf PING vom Client. Zeitstempel identisch mit PING.

Client zu Server

Protokoll	Abkürzung	Wert	Beschreibung
TCP	MESS	`0x01`	Payload an alle verbundenen Clients senden. (Broadcastfunktion)
	PING	`0x05`	Eine TCP-PONG-Antwort vom Server anfordern. Unabhängig von der Zeitsynchronisation!
	EXIT	`0x10`	Den Server zum Schließen aller Verbindungen und Beenden der Serveranwendung veranlassen.
UDP	PING	`0x05`	Einen PING mit aktuellem Zeitstempel des Clients senden.
	STIM	`0x04`	Letzter Schritt der Zeitsynchronisation. Den Zeitpunkt des zuletzt erhaltenen *PONG* als Payload anhängen. Zeitstempel identisch mit PONG.

In dieser Tabelle ist mit „Client“ das Softwaremodul auf dem Entwickler-PC und als „Server“ das Gegenstück auf dem eingebetteten System gemeint. Die Spalte „Wert“ enthält die im Header als „Typ“ bezeichnete Kodierung des Pakettyps.

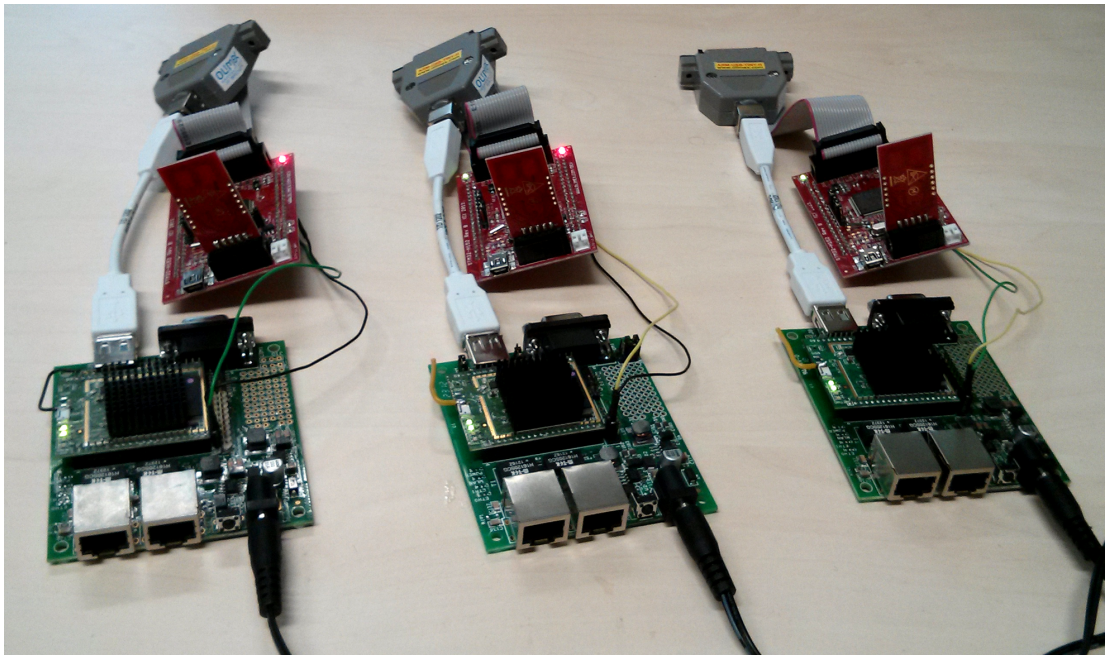
Abbildung 3.9. Nachrichtentypen des Protokolls

4. Realisierung und Analyse

Inhaltsangabe

In diesem Kapitel wird die Realisierung des Entwicklungssystems beschrieben. Es werden die Funktionen der einzelnen Komponenten und deren Umsetzung erläutert.

Abschließend soll die Genauigkeit des Systems untersucht werden.



Diese Fotografie zeigt einen Teil des aufgebauten Entwicklungssystems. Erkennen lassen sich im Vordergrund die „Carambolas“, die über zwei Leitungen mit dem UART des roten „Zielsystems“ verbunden sind. Über USB sind an die „Carambolas“ die grauen JTAG-Adapter angeschlossen, die ihrerseits ebenso mit den „Zielsystemen“ verbunden sind.

Abbildung 4.1. Aufbau des Entwicklungssystems

4.1. Konfiguration des Carambola

OpenWRT bietet mit dem Programm Unified Configuration Interface (UCI) eine Anwendung, Einstellungen zentralisiert zu verwalten. Über das UCI-System lassen sich unter anderem Ethernet, WLAN, DHCP und SSH-Server konfigurieren. Die einzelnen Einstellungsdateien liegen alle im Verzeichnis `/etc/config/` und können mittels des Tools modifiziert werden.

Nach Systemstart werden außerdem alle Dateien im Ordner `/etc/uci-defaults/` eingelesen, ausgeführt und anschließend gelöscht. Dies erlaubt es Softwarepaketen, Einstellungen am System vorzunehmen. Werden diese Softwarepakete bereits in die Firmware integriert, entspricht dies einer Art **Vorkonfiguration** des gesamten Systems.

Darum wurde die Datei `freejtag.uci-defaults` erstellt, um eine entsprechende Einrichtung vornehmen zu können. Da alle diese Einstellungen für das zu entwerfende Entwicklungssystem

spezifisch sind, wird diese Datei zusammen mit der kompilierten Serveranwendung installiert und nach einem Neustart beziehungsweise einem ersten Systemstart ausgeführt.

4.1.1. WLAN und Netzwerkkonfiguration

Da die Netzwerkkonfiguration von OpenWRT vollständig durch die UCI-Konfigurationsdateien abgewickelt wird, muss auch UCI genutzt werden, wenn das Netzwerk vorkonfiguriert werden soll.

UCI bietet mit `uci batch` explizit einen Befehl an, um umfangreiche Einstellungsänderungen vorzunehmen. Hierfür werden die einzelnen Befehle mittels *Here document* übergeben.

Definition 4.1 (Here document) *Ein* Here document *ermöglicht es, einem einzelnen Unix-Befehl mehrere, durch Zeilenumbrüche getrennte, Befehle zu übergeben.*

Die Verwendung des *Here documents* ist dabei wie folgt. Die zu tätigenden Einstellungen befinden sich dabei an der Stelle des mit `<Einstellungen>` bezeichneten Bereichs. Ein abschließendes `commit network` weist das UCI an, die Einstellungen zu speichern.

```
1 uci batch <<-EOF_network
2         <Einstellungen>
3         commit network
4         EOF_network
```

Da diese Einstellungen nur einmalig getätigt werden müssen, wird dafür die in Abschnitt 4.1 erwähnte Datei in `/etc/uci-defaults/` verwendet.

4.1.2. Aktivierung des UART

Der Mikrochip[12] des Carambolas besitzt zwei UARTs. Einer dieser UARTs wird für die Serielle Konsole über den RS232-Anschluss des Carambolas verwendet. Somit kann der zweite Anschluss für die Datenschnittstelle des Entwicklungssystems verwendet werden. Da die Pins des Carambolas für den zweiten UART standardmäßig auf General purpose input/output (GPIO)-Betrieb eingestellt sind, müssen sie vor der Verwendung entsprechend konfiguriert werden.

Hierzu muss, laut Wiki des Herstellers „8devices“[11], das Tool `io` installiert und mittels `io 0x10000060 0x01` ausgeführt werden. Dies weist den Mikrocontroller an, den UART zu

aktivieren. Dieser Vorgang muss nach jedem Systemstart erfolgen und kann durch die, bei jedem Systemstart aufgerufene, Datei `/etc/rc.local` erfolgen.

Zusätzlich muss verhindert werden, dass auf diesem UART eine Linuxterminal betrieben wird. Diese Änderung wird in der Datei `/etc/inittab` durchgeführt.

Die Änderungen werden wie folgt ebenso mittels des Skripts in `/etc/uci-defaults/` nach Abschnitt 4.1 durchgeführt.

```
1 sed -i '/exit 0/ i\io 0x10000060 0x01' /etc/rc.local
2 sed -i '/ttyS0/ s/^/# /' /etc/inittab
```

4.2. OpenOCD

Für *OpenOCD* existiert keine Portierung in die Paketverwaltung von OpenWRT. Es muss also ein Makefile erstellt werden, das die Einbindung ermöglicht. Der grundlegende Aufbau solcher Makefiles ist im Wiki von OpenWRT festgelegt[18].

Hierbei folgt der Ablauf, der durch ein solches Makefile ausgeführt wird, grob den Abläufen die nötig sind, um ein mit den GNU Build System entworfenes Programm zu kompilieren und installieren.

Es werden von dem Makefile nacheinander die Befehle `./configure` und `make` ausgeführt. Über das Makefile kann dabei angegeben werden, welche Parameter an diese Befehle übergeben werden.

4.2.1. Portierung auf OpenWRT

Um *OpenOCD* kompilieren zu können, müssen zuerst die Voraussetzungen festgestellt werden. Da ein auf dem FT2232-Chip basierender JTAG-Adapter mit USB-Anschluss eingesetzt werden soll, müssen laut `README` von *OpenOCD* sowohl libftdi als auch libusb installiert sein. Diese Pakete stehen für OpenWRT bereits zur Verfügung und müssen im Makefile als Abhängigkeiten definiert werden. Dies geschieht mit dem Befehl `DEPENDS:=+libftdi +libusb`.

Der Quellcode von *OpenOCD* in der Version 0.6.1 wird durch das Makefile von Sourceforge selbstständig heruntergeladen, über eine MD5 Hashfunktion auf Konsistenz überprüft und entpackt.

Um eine hohe Kompatibilität gegenüber allen möglichen Zielsystemen zu gewährleisten, musste das `Makefile` der `libftdi` angepasst werden, da die in den Repositories vorhandene Version 0.19 keine Kommunikation mit Adaptern gewährleistet, die einen FT232H-Chip besitzen. Diese Funktion gewährleistet erst Version 0.20.

Vor dem Kompilierungsvorgang müssen noch die an `./configure` zu übergebenden Argumente festgelegt werden. Der verwendete Adapter verwendet einen FT2232H-Chip und erfordert daher die Option `--enable-ft2232_libftdi` . Die Option `--enable-parport` wird in der `README` von *OpenOCD* als empfohlen angegeben und aktiviert die Unterstützung für JTAG-Adapter, die den Parallelport verwenden.

```
1 CONFIGURE_ARGS+= \
2         --prefix=$(STAGING_DIR)/usr \
3         --enable-dummy \
4         --enable-parport \
5         --enable-ft2232_libftdi
```

Zu den Kompilerargumenten muss die Option `-Wno-error=maybe-uninitialized` hinzugefügt werden, da anderenfalls eine den Kompiliervorgang abbrechende Warnung beim Kompilieren der Datei `src/target/dsp5680xx.c` auftritt.

4.2.2. Konfiguration

Die Konfiguration von *OpenOCD* erfolgt durch die Datei `openocd.cfg` . Diese Datei wird während der Installation in den Ordner `/etc/` kopiert. Sie beinhaltet alle Einstellungen, um *OpenOCD* mit dem *ARM-USB-TINY-H* und dem *STM32L* zu verwenden. Außerdem legt die Konfigurationsdatei den GDB-Port auf 3333 und den Telnet-Port von *OpenOCD* auf 4444 fest.

Zusätzlich muss sichergestellt werden, dass *OpenOCD* mit jedem Systemstart ausgeführt wird. Dazu wurde die Datei `openocd.init` erstellt, die bei der Installation in den Ordner `/etc/init.d/` kopiert wird. In diesem Ordner befinden sich unter Linux traditionell alle Skripte, die bei Systemstart ausgeführt werden sollen. Dies ermöglicht es, Daemons[3] zu starten.

[3] Anwendungen, die als Hintergrundprozesse laufen und meist auf spezielle Ereignisse wie Netzwerkverkehr reagieren.

4.2.3. Integration in Eclipse

Da *OpenOCD* neben einer Telnet-Konsole auch die Möglichkeit bietet, sich über einen GDB-Client steuern zu lassen, ist die Einbindung in verschiedenste Entwicklungsumgebungen oft denkbar einfach.

So bieten zum Beispiel „Qt Creator“, „Netbeans“ oder auch „Eclipse“ eine Unterstützung für GDB an. Für diese Arbeit soll exemplarisch eine Verbindung zum Entwicklungssystem mittels Eclipse hergestellt werden.

Um das System Flashen beziehungsweise Debuggen zu können, muss eine neue „Debug Konfiguration“ erstellt werden.

Als GDB-Debugger muss für ARM-Mikrocontroller dabei `arm-none-eabi-gdb` verwendet werden. Dieser Client ist in den meisten ARM-Toolchains bereits vorhanden. Außerdem muss als Zielsystem noch die IP des Carambolas und der konfigurierte Port angegeben werden.

Zusätzlich lässt sich eine Initialisierungssequenz festlegen, die vor dem Flashen ausgeführt wird. Basierend auf den Angaben des „OpenOCD User’s Guide“[25] und der Webseite des Real-time operating system (RTOS) „ChibiOS/RT“[29] erwies sich dabei folgende Initialisierungssequenz als vernünftig.

```
1 monitor soft_reset_halt
2 monitor soft_reset_halt
3 monitor wait_halt
4 monitor poll
5 monitor flash probe 0
6 monitor soft_reset_halt
```

Für das eigentliche Flashen des Bausteins muss lediglich die Projektdatei, üblicherweise eine `*.elf` -Datei, ausgewählt werden. Soll ein Debugging erfolgen, muss zusätzlich noch ein Laden der Symbolinformationen aus der selben Datei erfolgen.

4.3. Server - FreeJTAG

Der im Rahmen dieser Arbeit entwickelte Server für das Entwicklungssytem wurde in der Programmiersprache C++ geschrieben und wird im Weiteren *FreeJTAG* genannt. Er wird auf dem Carambola installiert, als Serveranwendung bei jedem Systemstart gestartet und leitet die gesammelten Daten an jeden verbundenen Client weiter.

4.3.1. Bibliotheken, Abhängigkeiten und Installation

Als Bibliotheken kommen bei *FreeJTAG* vor allem Teile der *Boost Bibliothek* zum Einsatz. Diese dienen der asynchronen Netzwerkkommunikation (*Boost ASIO*), dem hierfür nötigen Einsatz von Threads (*Boost Thread*), der Verwaltung der Zeitstempel (*Boost Chrono*) und dem Speichern und Einlesen von Programmeinstellungen (*Boost Program_Options*).

Um eine leichte Einbindung in das System der Makefiles von OpenWRT zu erreichen, wird für *FreeJTAG* das GNU Build System (Autotools) eingesetzt. Hierbei kommt zusätzlich die Datei `boost.m4` von Benoit Sigoure[30] zum Einsatz, um das System auf Existenz der benötigten Bibliotheken überprüfen zu können.

Außerdem hängt *FreeJTAG* von dem Paket „io“ ab, da dieses für die Aktivierung des UART, wie in Unterabschnitt 4.1.2 beschrieben, zuständig ist und installiert werden muss. Die in Abschnitt 4.1 beschriebene Skriptdatei, die mittels `/etc/uci-defaults/` eine Vorkonfiguration durchführt, ist ebenso in diesem Paket enthalten.

Die Konfiguration der Serveranwendung erfolgt über die Datei `/etc/freejtag.cfg`. Diese Datei enthält standardmäßig folgende Einstellungen und wird zusammen mit *FreeJTAG* installiert.

```
# Common
ping=10000000

# UART
device=/dev/ttyS0
baud=115200
data=8
parity=none
stop_bits=one
flow_control=none

# Network
port=12323
```

Die Konfiguration des UART ist umfangreich möglich. Dies erleichtert die Kommunikation mit verschiedensten Zielsystemen.

„port“ bezeichnet den TCP-Port, über den die Clientsoftware eine Verbindung zum Server herstellen kann.

Außerdem besitzt das System mit der Option „ping“ eine Möglichkeit, zu Testzwecken alle paar µs eine `MESS`-Nachricht mit dem Payload „PING!“ an alle verbundenen Clients zu senden. Diese Funktion ist für den regulären Betrieb nicht nötig und kann wahlweise abgeschaltet werden.

Alle diese Einstellungen lassen sich bei einem Start des Programmes `freejtag` von einer Linux-Konsole aus auch als Parameter übergeben. Alle verfügbaren Parameter lassen sich durch das Ausführen von `freejtag --help` anzeigen.

Da die Konfiguration von *OpenOCD* und dessen Betrieb direkt nach Systemstart spezifisch für dieses Projekt ist, werden bei der Installation von *FreeJTAG* auch die beiden Dateien `/etc/openocd.cfg` und `/etc/init.d/openocd` installiert. Dadurch besteht das Paket *OpenOCD* nur aus einem Makefile und kann so auch unabhängig von diesem Entwicklungssystem eingesetzt werden.

Durch die Installation dieser Anwendung wird das Carambola also komplett für den Einsatz als Entwicklungssystem konfiguriert.

4.3.2. Strukturierung der Serversoftware

Abbildung 4.2 stellt eine grobe Übersicht über den Aufbau der Serveranwendung dar. Hierbei wurden Details wie weitere Klassen und zusätzliche Aggregationen zwischen den einzelnen Elementen zugunsten einer besseren Übersicht ausgelassen.

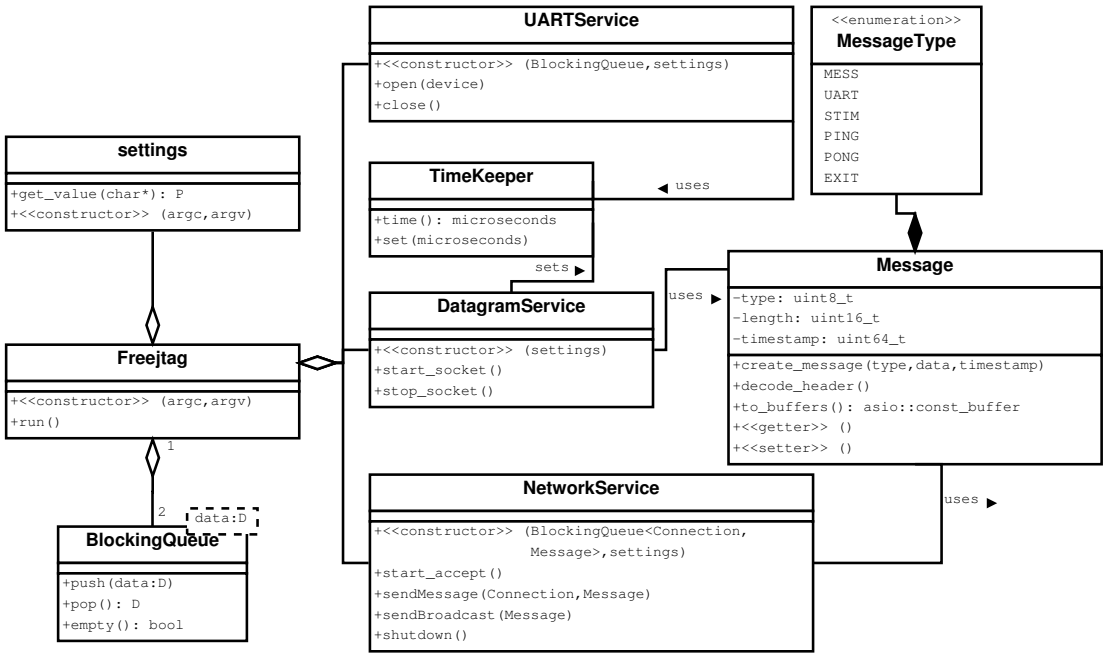
Die Hauptanwendung `Freejtag` dient als „Bindeglied“ zwischen den einzelnen Bestandteilen der Software. Die genaueren Abläufe hierzu sollen in Unterabschnitt 4.3.3 erläutert werden.

Weiterhin besteht die Serversoftware aus einer Klasse `NetworkService` zur Übertragung der Nutzdaten, `DatagramService` um die Zeitsynchronisation über UDP zu managen und `UARTService` für den Empfang der Daten vom Zielsystem.

Die Klasse `Message` dient zum Parsen der Netzwerkdaten und wird somit sowohl vom TCP- als auch vom UDP-Dienst verwendet.

Da die Zeitstempel synchronisiert werden sollen, wurde die Klasse `TimeKeeper` erstellt. Sie dient dem Setzen der Zeitdifferenz und der Abfrage des aktuellen Zeitstempels unter Berücksichtigung der gespeicherten Differenz.

Die Hilfsklasse `BlockingQueue` wird als Schnittstelle zwischen dem Hauptprogramm und dem `UARTService` sowie zwischen dem Hauptprogramm und dem `NetworkService` eingesetzt.



Diese Abbildung verdeutlicht den Aufbau der Serveranwendung (*FreeJTAG*). Hierbei wurden Details zu Gunsten der Übersichtlichkeit ausgelassen.

Abbildung 4.2. Grober Aufbau der Serveranwendung

Sie ist nötig, da jeder dieser Dienste in einem anderen Thread läuft und der Datenaustausch aus diesem Grund synchronisiert werden muss.

4.3.3. Funktionsweise der Serversoftware

In diesem Abschnitt sollen die wichtigsten Abläufe in der Serveranwendung erläutert werden. Der Schwerpunkt liegt hierbei hauptsächlich auf der Synchronisierung der Zeit und dem Weiterleiten der gesammelten Daten.

Programmstart

Die Anwendung wird zuerst initialisiert und anschließend ausgeführt. Dies erfolgt durch folgenden Ablauf.

```
1 int main(int argc, char* argv[]) {
2     freejtag::Freejtag *prog;
3     prog = new freejtag::Freejtag(argc, argv);
4     int res = prog->run();
5     delete prog;
6     return res;
7 }
```

Für die Funktion dieser Software müssen zu Beginn einige Einstellungen festgelegt werden. Hierzu zählen die Parameter der UART-Verbindung (Baudrate, Parität, Stopp-Bit) und der Netzwerkport.

Bei Erstellung des `Freejtag`-Objektes werden zuerst die Werte aus `/etc/freejtag.cfg` und die per Parameter (und in diesem Fall Kommandozeile) übergebenen Argumente als Einstellungen eingelesen. Das Einlesen dieser Werte erfolgt unter Nutzung der *Boost.Program_options*-Bibliothek. Tritt hierbei ein Fehler auf, wird das Programm mit einer Fehlermeldung beendet.

Intern werden die über das Netzwerk zu versendenden und empfangenen Nachrichten mittels `boost::shared_ptr` (Smart Pointer) von `Message`-Objekten verwendet. Es muss nun eine `BlockingQueue` initialisiert werden, die die über den TCP-Port empfangenen Nachrichten verwaltet. Da asynchrone TCP-Sendevorgängen mittels der `boost::asio` -Bibliothek erfolgen, ist eine weitere `BlockingQueue` zur Synchronisierung hierfür nicht notwendig.

Da es möglich sein soll mehrere Verbindungen (siehe auch Unterabschnitt 3.4.3) zu verwalten, muss in der `BlockingQueue` zusammen mit der `Message` auch immer die Verbindung von der sie erhalten wurde gespeichert werden.

Eine zweite `BlockingQueue` dient zur Erfassung der über UART gesammelten Daten mit zugehörigen Zeitstempeln.

Nun werden sowohl `NetworkService` als auch `DatagramService` initialisiert.

Der `NetworkService` erhält hierfür die vorher erstellte `BlockingQueue` (den Buffer), legt das Protokoll auf TCP fest und öffnet den in den Einstellungen spezifizierten Port. Der Ablauf des `DatagramService` verläuft analog mit dem Unterschied, dass für die über diesen Dienst erfolgende Zeitsynchronisation kein Zugriff auf den Nachrichten-Buffer nötig ist.

Anschließend werden zwei Threads gestartet, die für die Abwicklung der, durch den Empfang von UART- oder Netzwerkdaten angestoßenen, Vorgänge zuständig ist. Hierfür warten die Threads mittels `UARTMessage msg = input_uart_.pop();` beziehungsweise `MessageDatagram msgd = input_network_.pop();` auf die Ankunft neuer Daten in ihrere zugehörigen `BlockingQueue`. Dies schließt die Initialisierung ab.

Nun muss `Freejtag::run();` aufgerufen werden, um den Start der Anwendung zu vollziehen. Dies startet ein asynchrones Accept auf dem Socket des Netzwerkmoduls, so dass es nun möglich ist, eine Verbindung zum Server herzustellen. Der UART-Anschluss wird auf dem angegebenen Gerät geöffnet und mit den eingestellten Parametern konfiguriert.

Schließlich ruft der startende Thread die blockierende Methode `io_service_.run();` auf und dient nun zur Abarbeitung der asynchronen Netzwerkvorgänge.

Damit ist der Programmstart abgeschlossen und die Serveranwendung kann verwendet werden.

Zeitsynchronisation

Der aktuell gültige Zeitstempel wird bei Aufruf von `Timekeeper::time()` durch eine simple Subtraktion errechnet.

```
1 static high_resolution_clock::time_point epoch;
2 microseconds TimeKeeper::time(){
3     high_resolution_clock::duration now =
4                         high_resolution_clock::now() - epoch;
5     return duration_cast<microseconds>(now);
6 }
```

Um diese Berechnung durchführen zu können, muss jedoch zuerst eine Synchronisation des Systems zum Client erfolgen. Der Ablauf dieser Synchronistion erfolgt wie in Unterabschnitt 3.4.4 beschrieben.

Der Einfachheit halber ist der Server nicht in der Lage, mehrere Synchronisierungsvorgänge gleichzeitig durchzuführen. Diese müssen also nacheinander ausgeführt werden.

Um die Abweichung zuverlässig ermitteln zu können, erfolgt nach 20 erfolgreichen Synchronisierungsvorgängen eine Bildung des arithmetischen Mittels aller bis zu diesem Zeitpunkt gewonnenen Daten.

Eine Ausnahme bilden hierbei Ergebnisse, die einen Differenz zum Client von mehr als 90 ms aufzeigen. Tritt eine solch „hohe“ Differenz auf, wird die Abweichung unverzüglich und auf Basis dieses einen Ergebnis justiert. Hierdurch soll einer Verfälschung der Mittelwertbildung durch anfänglich stark abweichende Uhren vorgebeugt werden.

Da nun die gemittelte Abweichung zur Clientsoftware berechnet wurde, kann die Funktion `TimeKeeper::set(microseconds difference)` verwendet werden, um die aktuell gültige `epoch` anzupassen. Durch diese Justierung wird der `TimeKeeper` effektiv auf 0 µs gesetzt.

Empfang und Weiterleitung von UART-Daten

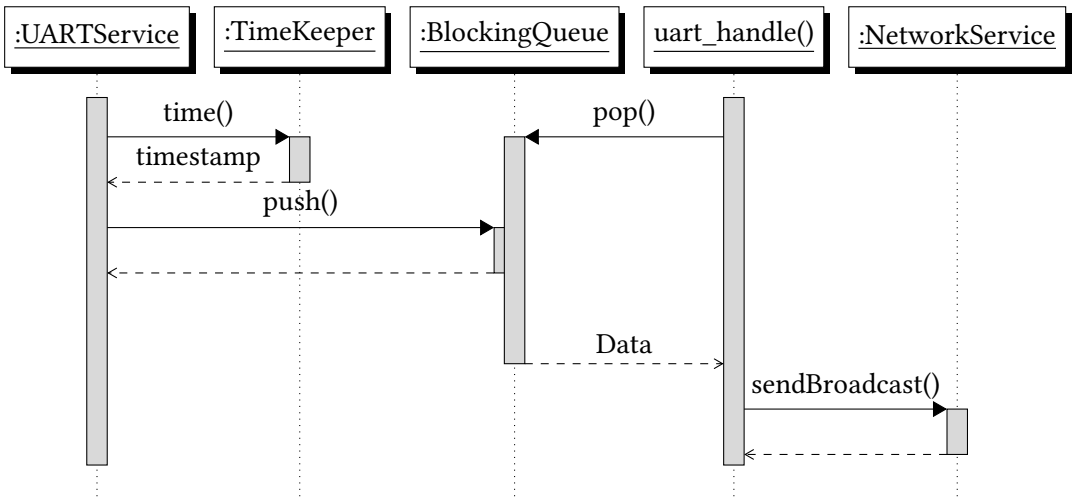
Die Hauptfunktion des Servers liegt im Weiterleiten von über den UART-Anschluss erhaltenen Daten.

Zu diesem Zweck wird bei Systemstart durch den `UARTService` die asynchrone Funktion `boost::asio::async_read_until()` ausgeführt. Dieses speichert alle über UART eintreffenden Zeichen in einen internen Buffer, wertet sie aus und ruft bei Eintreffen des Newline-Zeichens ’\n’ die Funktion `Freejtag::handle_read` auf.

Sobald diese „Handler-Routine“ aufgerufen wird, fragt sie den aktuellen Zeitstempel über den `TimeKeeper` ab. Zu diesem Zweck besitzt dieser die statische Funktion `time()`, die den aktuellen Zeitpunkt in Mikrosekunden wie in Abschnitt 4.3.3 erläutert zurückgibt.

Anschließend wird der Zeitstempel zusammen mit der erhaltenen Textzeile in die Buffer-Queue gegeben. Aus dieser Queue wird das Zeitstempel-Nachrichten-Paar abgeholt und mittels `NetworkService::sendBroadcast(Message::pointer msg)` asynchron an alle verbundenen Clients gesendet.

Ein beispielhafter Ablauf ist in Abbildung 4.3 abgebildet.



In diesem Ablauf werden die Vorgänge nach Empfang einer Zeile des UART dargstellt. Hierbei ist zu beachten, dass der `UARTService`, anders als hier eingezeichnet, keinen eigenen Thread besitzt. Stattdessen wird ein asynchrones Ereignis ausgelöst.

Abbildung 4.3. Ablauf bei Empfang einer Zeile von UART

Beenden des Servers

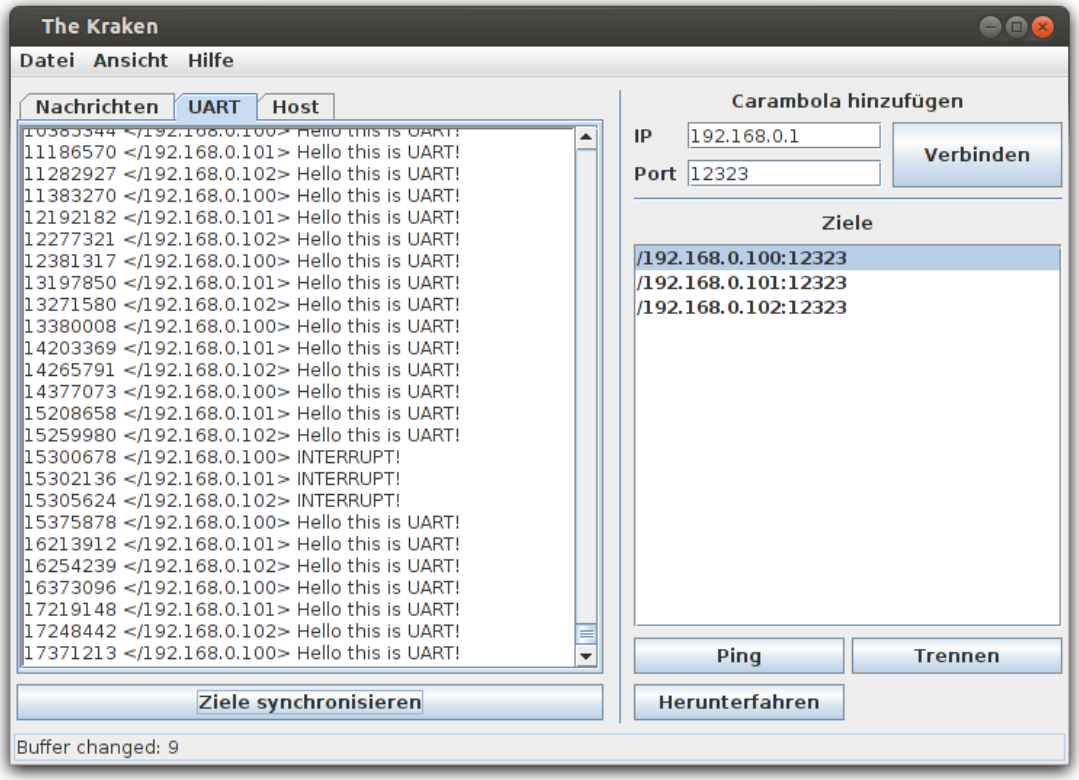
Zum Beenden der Serveranwendung wird durch einen der verbundenen Clients eine `EXIT`-Nachricht an den Server geschickt. Dieser führt anschließend folgende Schritte aus:

- Ausgeführt duch Thread `network_dispatcher_`
  - Variable `_running` auf `false` setzen.
  - Acceptor für neu ankommende Verbindungen schließen.
  - Auf dem Socket der TCP-Verbindung ein `socket_.shutdown()` und `socket_.close()` ausführen, um die Verbindungen zu schließen.
  - Den Seriellen Port des `UARTService` mittels `port_.close()` schließen.
  - Den Socket des UDP-Servers mittels `socket_.close()` schließen.
  - Da keine Netzwerkoperationen anstehen, ist der Hauptthread nun nicht mehr auf `io_service_.run()` blockiert.

- Ausgeführt durch Hauptthread
  - Den auf `input_network_.pop();` blockierenden `network_dispatcher_` unterbrechen.
  - Den auf `input_uart_.pop();` blockierenden `uart_dispatcher_` unterbrechen.

Abschließend laufen alle Threads wieder zusammen und die Serveranwendung ist beendet.

4.4. Client - The Kraken



Diese Abbildung zeigt die Clientanwendung „The Kraken“ beim Erfassen der gesammelten Daten. Ein bei den Zielsystemen gleichzeitig ausgelöster Interrupt ist erkennbar.

Abbildung 4.4. Die Clientanwendung

Die im Rahmen dieser Arbeit entwickelte Clientsoftware *The Kraken* dient zur Anzeige der über *FreeJTAG* gesammelten Daten. Sie wurde in der Programmiersprache Java geschrieben

und kann eine beliebige Anzahl Verbindungen zu *FreeJTAG*-Servern hergestellen. Da während der Übertragung der Nachrichten Verzögerungen auftreten können, muss eine Sortierung der Nachrichten nach ihren Zeitstempeln gewährleistet werden.

4.4.1. Funktionalität

Die Clientsoftware dient hauptsächlich dem Erfassen und Anzeigen der von der Serversoftware gesammelten Daten. Abbildung 4.4 stellt die Oberfläche der Software dar.

Die Software erfüllt folgende Funktionen:

- Verbindungsaufbau zu einer beliebigen Anzahl Server
- Sortierung der Daten (siehe Unterabschnitt 4.4.2)
- Beenden einer Serveranwendung
- Senden von `MESS`-Nachrichten an einen oder alle verbundenen Server (Broadcast)
- Senden von TCP-`PING`-Nachrichten um vom Server ein TCP-`PONG` zu erhalten.
- Unterbrechen der Datenaufnahme
- Synchronisieren der Zeiten aller Serveranwendungen mit dem Client
- Anzeige der realen Differenz zwischen einzelnen Empfangsvorgängen

4.4.2. Sortierung der Daten

Verzögerungen von Nachrichten sind in einem IP-Netzwerk nichts ungewöhnliches. Da für dieses Entwicklungssystem die Genauigkeit jedoch höchste Priorität hat, ist es wichtig, dass die Nachrichten ihren Zeitstempeln entsprechend geordnet werden.

Da die Daten im Betrieb fortlaufend eintreffen, wird eine entsprechende Methode benötigt, die Daten zu sortieren.

Ein Sortieralgorithmus, wie zum Beispiel „Quicksort“, funktioniert nur so lange, wie ein gleichbleibender Datensatz existiert. Da das in diesem Fall jedoch nicht so ist, ist nach einer alternativen Lösung zu suchen.

Ein **TreeSet** sorgt dafür, dass die Daten direkt beim Eintreffen nach ihren Zeitstempeln in den Baum einsortiert werden. Dies erlaubt es relativ einfach, das jeweils kleinste Element (das Element mit dem niedrigsten Zeitstempel) auszugeben.

Hierbei muss jedoch sichergestellt werden, dass immer eine Anzahl an Nachrichten in diesem Buffer verbleibt, da ansonsten keine Sortierung erfolgen kann. Die Größe des Buffers beträgt aus diesem Grund $AnzahlVerbindungen * 3$.

4.4.3. Aufbau der Anwendung

Die Strukturierung der Clientanwendung folgt lose dem Model View Presenter (MVP)-Modell.

Der EventBus der *Google Guava*-Bibliothek[31] wird verwendet, um die Ereignisse der Softwareelemente miteinander zu verknüpfen.

Um die Funktionsweise der Clientsoftware zu erläutern, soll hier exemplarisch der Empfangsvorgang einer Nachricht dargestellt werden.

Für jeden verbundenen Client wird ein `TCPConnection`-Objekt angelegt. Jedes dieser Objekte wartet über einen `DataInputStream` auf ankommende Daten. Werden nun Daten empfangen, dekodiert die Funktion `KrakenMessage.read()` die Bestandteile des Datenpaketes. Anschließend fügt die `TCPConnection` die empfangene `KrakenMessage` dem in Unterabschnitt 4.4.2 erwähnten `TreeSet` hinzu.

Dieses `TreeSet` wird von dem `MessageDispatcher` verwaltet. Er sorgt dafür, dass immer eine gewisse Mindestanzahl an `KrakenMessage`-Objekten im `TreeSet` verbleibt und leitet die Nachricht mit dem derzeit kleinsten Zeitstempel an den `EventBus` über die Funktion `EventBus.post()` weiter.

Das Event wird von der Funktion `Presenter.handleNewMessage()` aufgefangen. Dieser untersucht die Nachricht auf ihren Typ (`UART`, `MESS` oder `PONG`) und führt die, dieser Nachricht entsprechenden, Funktionen der GUI aus.

Wird durch Benutzerinteraktion ein Ereignis in der GUI ausgelöst, wird der `EventBus` in „umgekehrter Reihenfolge“ verwendet.

Die Anwendung wurde mithilfe des Build-Management-Tools „Maven“[32] entwickelt und verwendet zusätzlich Proguard[33] zum Minimieren der ausführbaren Datei und launch4j[34] zur Erstellung der *.exe-Dateien.

4.5. Deployment

Mittels einer Linux-Scriptdatei lässt sich der GDB im Batch-Modus steuern. Zum Einsatz kommt hier ein Here-document, dass die in Unterabschnitt 4.2.3 erwähnten Initialisierungsbefehle ausführt, um anschließend die Firmware durch Ausführen von `load` zu flashen.

```
#!/bin/bash
echo "Flashing␣$1"
for target in "${@:2}"
do
 echo "Flash␣$target"
 arm-none-eabi-gdb $1 <<EOF
 target remote $target:3333
 monitor reset halt
 monitor soft_reset_halt
 monitor wait_halt
 monitor poll
 monitor flash probe 0
 monitor soft_reset_halt
 tbreak main
 load
EOF
done
```

Dieses Script erfordert als erstes Argument die zu flashende Datei und anschließend die IPs einer beliebigen Anzahl von Zielsystemen.

Das Zurücksetzen der Zielsysteme erfolgt analog hierzu. Dabei entfallen jedoch die Initialisierungsbefehle und das Flashen. Lediglich der Befehl `monitor reset run` muss von gdb ausgeführt werden.

Die Einbindung in Eclipse ist durch die Verwendung der Scripte als „Externe Tools“ einfach möglich.

4.6. Aufgetretene Probleme

Während der Inbetriebnahme von *OpenOCD* traten häufig Probleme mit der Verbindung zum JTAG-Adapter auf. Dies äußerte sich darin, dass das Linux-System regelmäßig die Verbindung zu dem Adapter verlor. Der Betrieb von *OpenOCD* wurde dadurch erheblich erschwert.

Jedoch ist dies, laut dem Benutzerforum des Herstellers, ein bekanntes Konstruktionsproblem der „Carambolas“ hervorgerufen durch eine unzulängliche Schirmung des USB-Anschlusses gegenüber elektromagnetischen Wellen. Die integrierte WLAN-Antenne beeinflusst deshalb den Betrieb von USB-Geräten unter Umständen stark.

Um dieses Problem zu beheben wurde ein 32 mm langer Draht an die Masse der WLAN-Antenne gelötet. Anschließend lies sich der JTAG-Adapter problemlos betreiben. Alternativ hätte auch die Möglichkeit bestanden, die Sendeleistung des WLAN-Moduls über die Einstellung `option txpower 10` in der Datei `/etc/config/wireless` zu reduzieren.

Bei der Übertragung von Testdaten über das System zeigte sich, dass der Nagle-Algorithmus[28] auch den TCP-Datenverkehr stark beeinflusst. So musste für eine funktionierende Sortierung der Daten ein sehr hoher Buffer-Wert eingestellt werden, wodurch die Anzeige stark verzögert wurde. Um dies zu verhindern, wurde auch für TCP der Nagle-Algorithmus deaktiviert.

4.7. Analyse des Systems

Da die Präzision der Zeitstempel von zentraler Bedeutung für die Aussagekraft der gesammelten Daten ist, muss festgestellt werden, wie exakt zum Beispiel die Zeitsynchronisation funktioniert.

Um die Genauigkeit messen zu können, müssen alle Zielsysteme möglichst zeitgleich ein Signal erhalten, um dann unverzüglich über UART eine Nachricht an die angeschlossenen „Carambolas“ zu übermitteln.

Hierfür wurde exemplarisch das RTOS ChibiOS/RT[29] auf den Zielsystemen installiert. Es wurde so konfiguriert, dass ein Interrupt ausgelöst wird, sobald ein bestimmter Pin als „Auslöser“ auf Masse gezogen wird.

Da die „Carambolas“ durch separate Netzteile alle verschiedene Massepotentiale haben, müssen diese zuerst miteinander verbunden werden. Anschließend kann man alle „Auslöser“-Pins miteinander verbinden. Dies erlaubt es einem, mittels eines Knopfes oder eines Überbrückungskabels, die so verbundenen Pins zeitgleich auszulösen.

Hat man vorher einen Synchronisierungsvorgang durch den Client durchgeführt, lässt sich durch die Differenz der Zeitstempel nun die tatsächliche Abweichung voneinander messen. Im Idealfall ist die Differenz hierbei möglichst gering (nach Unterabschnitt 3.4.4 weniger als 1 ms). Das Eintreffen dieser erwähnten Interrupts ist auch auf Abbildung 4.4 zu erkennen.

	Knoten 0		Knoten 1		Knoten 2
Minute	Zeitstempel	Diff zu Knoten 1	Zeitstempel	Diff zu Knoten 2	Zeitstempel
0	10613705	-0.02ms	10613684	0.2ms	10613859
15	950992475	6.4ms	950998898	12.5ms	951011363
30	1854818581	12.6ms	1854831162	24.3ms	1854855460
45	2754582459	18.7ms	2754601202	36.2ms	2754637360
60	3655782708	24.9ms	3655807605	47.9ms	3655855473
75	4554104959	31.0ms	4554135948	59.6ms	4554195595
90	5453785960	37.0ms	5453823040	71.5ms	5453894583
	Drift pro 15 min gegenüber Knoten 1		**Drift pro 15 min gegenüber Knoten 2**		
	6.2 ms		**11.9 ms**		

Die Zeitstempel sind die von der Clientsoftware ausgegebenen Werte. Die Differenzen zwischen den Knoten sind gerundete Angaben, da eine Genauigkeit der Zeitstempel von 1 µs nicht zu erwarten ist. Der durchschnittliche Drift über 15 min errechnet sich über die Differenz zwischen den einzelnen Knoten nach einer Testzeit von 90 min.

Abbildung 4.5. Testdaten einer Testreihe zur Prüfung der Genauigkeit der „Carambolas“

Allerdings kann dies natürlich nur die Differenzen der Zielysteme untereinander aufzeigen. Die Abweichung gegenüber dem Entwicklerrechner ist für eine Analyse der Netzwerkabläufe jedoch auch von untergeordneter Relevanz, da dieser in den Abläufen keine Rolle spielt.

Es wurde ein Testtlauf über 90 min durchgeführt. Direkt vor dessen Durchführung wurde eine Zeitsynchronisation des gesamten Entwicklungssystems und damit aller „Carambolas“ durchgeführt.

Im Versuchsablauf wurde, wie weiter oben erläutert, ungefähr alle 15 min ein Interrupt bei allen Zielsystemen gleichzeitig ausgelöst. Bei diese Testreihe war nicht entscheidend, den Interrupt exakt alle 15 min auszulösen, da hauptsächlich die sich verändernden Differenzen gezeigt werden sollten.

Wie aus Abbildung 4.5 ersichtlich, ist direkt nach der Synchronisation die absolute Differenz zwischen „Knoten 0“ und „Knoten 1“ bei nur 0.02 ms und zwischen „Knoten 1“ und „Knoten 2“ bei 0.2 ms. Dies zeigt, dass die Zeitsynchronisation eine ziemlich hohe Genauigkeit erreicht.

Aus dem Verlauf der Testreihe ist jedoch auch ersichtlich, dass die Differenz zwischen den Systemen nicht gleichbleibend ist. So steigt jeweils über einen Zeitraum von 15 min die Differenz zwischen „Knoten 0“ und „Knoten 1“ um ungefähr 6.2 ms und zwischen „Knoten 1“ und „Knoten 2“ um 11.9 ms.

Dies erklärt sich aus den Fertigungstoleranzen der Schwingquarze. Die Abweichung zwischen „Knoten 0“ und „Knoten 2“ vergrößert sich alle 15 min um ungefähr 18 ms was einer Abweichung von 20 ppm entspricht ($\frac{18\,\text{ms}}{15\,\text{min}} * 1000000 = 20\text{ppm}$). Diese Abweichung ist für einen Schwingquarz noch innerhalb der üblichen Frequenztoleranz.

Die Genauigkeit der Zeitsynchronisation selbst jedoch liegt in der in Unterabschnitt 3.4.4 geforderten Genauigkeit von **unter 1 ms**. So wäre eine exakte Erfassung der Testdaten zumindest anfänglich noch möglich.

Wollte man diesen Drift nun ausgleichen, könnte man dafür die in Abbildung 4.5 aufgeführten Drifts der Clocks als Grundlage nehmen.

Der Linux-Kernel verwendet den Algorithmus nach NTP[35] zur Justierung der internen Clock. Hierfür lassen sich verschiedene Kernelparameter konfigurieren, die diesen Algorithmus beeinflussen. Einer dieser Parameter spezifiziert auch die tatsächliche Abweichung von der erwarteten Taktrate. Zum Setzen dieser Parameter wird das Tool `adjtimex` verwendet.

Würde man nun diese Funktionalität des Kernels verwenden, ließe sich der aufgetretene Drift der Uhr ausgleichen.

5. Ausblick und Fazit

Inhaltsangabe

5.1. Verbesserungen des entwickelten Systems

Einige Komponenten des entwickelten Systems erfüllen zwar die gestellten Anforderungen, könnten in ihrer Funktionsweise jedoch noch verbessert werden. Diese Verbesserungen betreffen zum Beispiel die Genauigkeit der Zeitstempel oder den Ressourcenverbrauch der Software.

Einige denkbare Verbesserungen sollen hier angeführt werden.

5.1.1. Kompensation des Clock-Drifts

Die Analyse aus Abschnitt 4.7 zeigt, dass die Herstellungstoleranzen der Schwingquarze einen starken Drift verursachen und so die Zeitmessung massiv beeinflussen.

Die erwähnten Funktionen von `adjtimex` könnte man in der Serversoftware in Zusammenspiel mit dem Client verwenden, um eine „gemeinsame Taktrate“ zu errechnen.

Dadurch wäre es unter Umständen möglich, Betrachtungen von Zeitabläufen auch über einen längeren Zeitraum durchführen zu können, ohne regelmäßig eine Zeitsynchronistion durchführen zu müssen.

5.1.2. Parallelisierung

Da die in Abschnitt 4.5 vorgestellten Skripte die Programmierung der einzelnen Zielsysteme seriell ausführen, wird für eine zunehmende Anzahl an Zielsystemen auch zunehmend mehr Zeit zum Flashen nötig.

Außerdem ist ein durch solch ein Skript verursachter „gemeinsamer Neustart“ immer signifikant zeitversetzt.

Eine denkbare Lösung hierfür wäre es, das Flashen und Neustarten unterschiedlicher Zielsysteme von mehreren Threads gleichzeitig ausführen zu lassen.

5.1.3. Speicherverbrauch des Servers

Im Betrieb von *FreeJTAG* stellte sich heraus, dass das Programm einen relativ hohen Speicherverbrauch hat ($\approx 9400\,\mathrm{kB}$).

Dies ist für eine derartige Anwendung auf einem eingebetteten System ein ziemlich hoher Wert. So benötigt zum Beispiel das relativ komplexe *OpenOCD* ungefähr 2600 kB.

Dieser Speicherverbrauch ist für das Carambola mit seinen 32 MB Arbeitsspeicher zwar noch keine Schwierigkeit, schränkt jedoch die Verwendung weiterer Software ein und verhindert unter Umständen die Portierung auf leistungsschwächere Hardware.

Eine Optimierung der Serversoftware ist also unter Umständen möglich und eröffnet weitere Optionen, mit dem Carambola zu arbeiten.

5.2. Mögliche Erweiterungen

Da das gesamte Entwicklungssystem in Hinblick auf größtmögliche Flexibilität und Modularisierung entwickelt wurde, sind eine Vielzahl möglicher Erweiterungen denkbar. Einige dieser Möglichkeiten sollen hier aufgeführt werden.

5.2.1. Zusätzliche Anschlussmöglichkeiten

Während für die Kommunikation zwischen Carambola und Zielsystem zur Zeit UART eingesetzt wird, ist auch eine Erweiterung auf die SPI-Schnittstelle denkbar.

Hierfür müsste in die Serversoftware ein `SPIService` integriert werden. Durch die Entkoppelung zwischen `NetworkService` und `UARTService` wäre es zum Beispiel möglich, den bereits vom `UARTService` benutzten Buffer zu verwenden, um die erhaltenen SPI-Nachrichten an den Rest des Systems weiterzuleiten.

Diese Vorgehensweise hätte den Vorteil, dass dadurch keine weiteren Änderungen am Rest des Systems nötig wären.

Ein anderer Ansatz wäre, eine der ungenutzten IDs des Netzwerkprotokolls zu verwenden, um SPI-Daten als separate Werte im Client erfassen zu können. Dies erfordert jedoch etwas umfangreichere Änderungen am Server, da ein weiterer Thread und ein weiterer Buffer angelegt werden müssten. Außerdem müsste die Clientsoftware um das Empfangen von SPI-Datenpaketen erweitert werden.

Eine ähnliche Erweiterung wäre auch für die Weiterleitung von I^2C-Daten denkbar.

5.2.2. Analyse via Wireshark

Das das Entwicklungssystem hauptsächlich auf die Analyse von Funknetzwerken zugeschnitten ist, wäre auch diesbezüglich eine Erweiterung denkbar.

So könnte man den Client zum Beispiel die gesammelten Netzwerkdaten in eine Datei schreiben lassen.

Da der Netzwerkanalyzer „Wireshark“[36] umfangreiche Möglichkeiten bietet, ihm neue Protokolle hinzuzufügen, könnte man ihn unter Umständen auch dazu verwenden, diese Datei einzulesen.

Würde man diesen Gedanken noch weiter tragen, wäre es wünschenswert, mit Wireshark eine direkte Verbindung zu den einzelnen Carambolas aufzubauen. So ließe sich hierüber ein verteilter Netzwerksniffer realisieren.

5.3. Fazit

Es konnte erfolgreich ein System entwickelt werden, mit dessen Hilfe die Entwicklung eines verteilten Funknetzwerks eingebetteter Systems wesentlich erleicht wird.

Wird eine Software für ein solches eingebettetes System entwickelt, ist es nun möglich, fast alle Entwicklungsphasen netzwerkgestützt durchzuführen.

Dazu besteht das gesamte Entwicklungssystem aus einer Anzahl von OpenWRT-Plattformen, von denen jeweils eine einem Zielsystem zugeordnet ist. Diese Plattform ist durch eine direkte Verbindung in der Lage, physischen Zugriff auf die JTAG-Schnittstelle des Zielsystems zu nehmen. Durch den Einsatz von OpenOCD kann ein GDB-Client verwendet werden, um das Zielsystem flashen und debuggen zu können.

Zusätzlich ist es dem OpenWRT-Modul mit Hilfe der entworfenen Software „FreeJTAG“ möglich, die von seinem Netzwerkknoten ausgegebenen UART-Daten zu erfassen.

Über ein WLAN oder Ethernet sind alle einzelnen OpenWRT-Module mit der Clientsoftware „The Kraken“ verbunden, die die über UART erfassten Daten sammelt und nach ihren, von „FreeJTAG“ gesetzten, Zeitstempeln sortiert. Die Kommunikation zwischen „FreeJTAG“ und „The Kraken“ erfolgt dabei über ein selbstentwickeltes Protokoll.

Da für die Erfassung von Funkdaten üblicherweise eine hohe zeitliche Genauigkeit benötigt wird, wurde besonderes Augenmerk auf die Zeitsynchronisation zwischen den beiden Softwarebestandteilen gelegt. So wurde für das entworfene Protkoll ein Synchronisationsablauf entwickelt, der in Tests eine hohe Präzision zeigte.

Allerdings war es im Zeitraum dieser Arbeit nicht möglich, ein beispielhaftes Funksystem zu implementieren, sondern nur eine Software zur Prüfung der Präzision des Entwicklungssystems. Dies war jedoch auch keine für die Arbeit essentielle Aufgabe, da nur die Voraussetzungen für die Entwicklung eines solchen Systems geschaffen werden sollten.

Die Testergebnisse zeigen außerdem, dass sowohl Debugging als auch Deployment und Analyse funktionieren und sich somit ein solches Funksystem unter Nutzung dieses Entwicklungssystems entwickeln ließe.

Ergänzt man die vorgeschlagenen Änderungen, könnte sich das Entwicklungssystem als flexible, robuste und günstige Lösung zur Vereinfachung der Entwicklung und Analyse eingebetteter Funknetzwerke erweisen.

A. Installationshinweise

Die beigelegte CD enthält den Quellcode der im Rahmen dieser Bachelorarbeit entwickelten Software.

Da die Serversoftware stark in das Build-System von OpenWRT integriert ist, folgt eine Auflistung der selbstentwickelten oder modifizierten Dateien.

```
Hauptordner
├── Thesis.pdf Installation
├── scripts
│   └── flash-all run-all
├── binary
│   ├── TheKraken.exe
│   ├── chibios.elf
│   └── openwrt-ramips-rt305x-carambola-squashfs-sysupgrade.bin
├── thekraken
│   ├── src
│   │   └── ...
│   └── pom.xml
├── chibios
│   └── ...
└── carambola
    ├── package
    │   ├── freejtag
    │   │   ├── files
    │   │   │   └── freejtag.uci-defaults openocd.cfg openocd.init
    │   │   ├── src
    │   │   │   └── ...
    │   │   └── Makefile
    │   └── openocd
    │       └── Makefile
    └── feeds/packages/libs/libftdi
        └── Makefile
```

Alle weiteren Dateien sind ausdrücklich nicht Bestandteil dieser Arbeit, allerdings für die Verwendung der Software notwendig. In der Datei `Installation` befinden sich Hinweise zur Konfiguration des Entwicklersystems und zum Kompilieren eines OpenWRT Images.

Abbildungsverzeichnis

Literatur

[1] *TinyOS*. TinyOS Alliance. URL: http://www.tinyos.net/ (besucht am 13. 04. 2013).

[2] *Contiki*. URL: http://www.contiki-os.org/ (besucht am 13. 04. 2013).

[3] *FreeRTOS™*. URL: http://www.freertos.org/ (besucht am 13. 04. 2013).

[4] Jonathan Hui. *Deluge 2.0 - TinyOS Network Programming*. URL: http://www.cs.berkeley.edu/~jwhui/deluge/deluge-manual.pdf (besucht am 13. 04. 2013).

[5] Maciej Jaros. 4. Sep. 2006. URL: http://commons.wikimedia.org/wiki/File:Wasserfallmodell.svg (besucht am 17. 03. 2013).

[6] Altera Corporation. *Boundary-Scan Tool*. URL: http://www.altera.com/support/devices/tools/boundary-scan/tls-boundary-scan.html (besucht am 17. 03. 2012).

[7] Michael Williams. *Low Pin-count Debug Interfaces for Multi-device Systems*. ARM Limited. 2009. URL: http://www.arm.com/files/pdf/Low_Pin-Count_Debug_Interfaces_for_Multi-device_Systems.pdf (besucht am 13. 04. 2013).

[8] *ZigBee Specification*. ZigBee Document. 053474r17. ZigBee Standards Organization, 17. Jan. 2008.

[9] *ZY1000 JTAG Debugger*. Ultimate Solutions, Inc. URL: http://www.ultsol.com/index.php/products/bdmjtag-emulators/210-zylin-zy1000-main (besucht am 05. 04. 2013).

[10] *WiFiDemon*. Macraigor Systems LLC. URL: http://www.macraigor.com/mpDemon.htm.

[11] *Carambola - Easily embeddable wireless module*. UAB "8 devices". URL: http://8devices.com/carambola (besucht am 11. 04. 2013).

[12] *RT3050/52 Datasheet*. Ralink Technology, Corp. 14. Aug. 2008.

[13] Mario Morales. "Intelligent Systems - Connectivity, Cloud, and Data". In: *Konferenzband des 2012 Intel Embedded Research and Education Summit*. 2012. URL: http://embedded.communities.intel.com/servlet/JiveServlet/previewBody/7160-102-1-2406/Intelligent-Systems-Transforming-the-Embedded-Industry.pdf (besucht am 18. 03. 2013).

[14] *STM32-H152 Entwicklungsboard*. Olimex Ltd. URL: https://www.olimex.com/Products/ARM/ST/STM32-H152/ (besucht am 13. 04. 2013).

[15] *MRF24J40 - IEEE 802.15.4™2.4 GHz RF Transceiver*. Microchip Technology Inc. URL: http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en027752 (besucht am 14. 04. 2013).

[16] IEEE. "IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)". In: *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)* (Mai 2011), S. 1 –314. DOI: 10.1109/IEEESTD.2011.6012487.

[17] Dominic Rath. "Design and Implementation of an On-Chip Debug Solution for Embedded Target Systems based on the ARM7 and ARM9 Family". 18. Juli 2005. URL: http://openocd.sourceforge.net/files/thesis.pdf (besucht am 01. 04. 2013).

[18] *OpenWrt Wiki*. URL: http://wiki.openwrt.org (besucht am 06. 04. 2013).

[19] *J-Link*. SEGGER Microcontroller GmbH & Co. KG. URL: http://www.segger.com/jlink.html (besucht am 05. 04. 2013).

[20] *ST-LINK/V2*. STMicroelectronics N.V. URL: http://www.st.com/internet/evalboard/product/251168.jsp (besucht am 05. 04. 2013).

[21] *JTAGkey2*. Amontec, Gauch. URL: http://www.amontec.com/jtagkey2.shtml (besucht am 05. 04. 2013).

[22] *ARM-USB-TINY-H - JTAG-Debugger*. Olimex Ltd. URL: https://www.olimex.com/Products/ARM/JTAG/ARM-USB-TINY-H/ (besucht am 05. 04. 2013).

[23] *Turtelizer 2*. egnite GmbH. URL: http://www.ethernut.de/en/hardware/turtelizer/index.html (besucht am 05. 04. 2013).

[24] Joern Kaipf. *OOCDLink*. URL: http://www.joernonline.de/contrexx2/cms/index.php?page=126 (besucht am 05. 04. 2013).

[25] *OpenOCD User's Guide*. The OpenOCD Project. URL: http://openocd.sourceforge.net/doc/html/index.html (besucht am 25. 03. 2013).

[26] *Application Note: JN-AN-1035*. Jennic Limited. 15. Aug. 2006. URL: http://www.jennic.com/files/support_files/JN-AN-1035%20Calculating%20802-15-4%20Data%20Rates-1v0.pdf (besucht am 08. 04. 2013).

[27] Flaviu Christian. "Probabilistic clock synchronization". In: *Distributed Computing* 3 (3 1. Sep. 1989), S. 146–158.

[28] John Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. IETF, 6. Jan. 1984.

[29] *ChibiOS/RT Homepage*. URL: http://www.chibios.org (besucht am 11. 04. 2013).

[30] Benoit Sigoure. *boost.m4 - M4 macros to use Boost with the autotools*. Version 0.4. URL: https://github.com/tsuna/boost.m4 (besucht am 15. 04. 2013).

[31] *Guava: Google Core Libraries for Java 1.6+*. Version 14.0-rc1. Google Inc. URL: https://code.google.com/p/guava-libraries/ (besucht am 15. 04. 2013).

[32] *Apache Maven - Build-Management-Tool*. Apache Software Foundation. URL: http://maven.apache.org/ (besucht am 15. 04. 2013).

[33] *ProGuard - Java class file shrinker, optimizer, obfuscator and preverifier*. Version 4.8. URL: http://proguard.sourceforge.net/ (besucht am 15. 04. 2013).

[34] Grzegorz Kowal. *Launch4j - Cross-platform Java executable wrapper*. URL: http://launch4j.sourceforge.net/ (besucht am 15. 04. 2013).

[35] David L. Mills. *Network Time Protocol (Version 3) Specification, Implementation and Analysis*. RFC 1305. IETF, März 1992.

[36] *Wireshark - Network Protocol Analyzer*. Wireshark Foundation. URL: http://www.wireshark.org/ (besucht am 15. 04. 2013).

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

Hamburg, 17. April 2013 Arne Wischer