Zeitsynchronisation mit Hilfe von Audiosignalen

Bachelorarbeit

von

Felix Hein

aus

Erlenbach

vorgelegt am

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

November 2014

Betreuer:
Norbert Goebel, M. Sc.

Zusammenfassung

Der Lehrstuhl für Rechnernetze hat ein Messframework zum Ausmessen von Mobilfunknetzen entwickelt und auf Android übertragen. Der Androidversion steht allerdings kein hochgenaues Pulse-Per-Second-Signal zur Verfügung, weil die in den verwendeten Geräten verbauten GPS-Empfänger dieses nicht durchreichen. Da das PPS-Signal aber für die Zeitsynchronisation benötigt wird, um sinnvolle Latenzmessungen durchführen zu können, wurde im Rahmen dieser Arbeit überprüft, ob die Übertragung des PPS eines externen GPS-Empfängers per Audio dafür nutzbar ist. Dazu wurde ein Proof-of-Concept auf Basis eines Raspberry Pi entwickelt, der neben dem PPS auch noch jede Sekunde einen GPS-Datensatz an einen PC überträgt. Mit den dabei entstandenen Programmen toa_send und toa_rcv (kurz für time over audio) wurden anschließend Testmessungen durchgeführt, die ergaben, dass die Erkennung weitgehend fehlerfrei machbar ist. Die Latenzen bei der Erkennung variierten jedoch stark, sodass die Programme in dieser Version nicht für eine Zeitsynchronisation geeignet sind. Ideen, die möglicherweise zu besseren Ergebnissen führen, konnten aus Zeitgründen nicht mehr umgesetzt werden.

Danksagung

Während dieser Arbeit standen mir einige Personen zur Seite, denen ich an dieser Stelle danken möchte:

- meinen Eltern, die immer ein offenes Ohr für mich hatten.
- Janine Haas und Michael Schlapa für die Anregungen und Korrekturen, die sie mir durch ihr Lektorat haben zukommen lassen.
- meiner Freundin Anne Schröder, die mir den Rücken frei gehalten hat und mich motiviert hat, wann immer es nötig war.
- meinem Betreuer Norbert Goebel natürlich, dessen Tür immer offenstand und mit dem ich viele ausführliche, interessante und hilfreiche Gespräche führen konnte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Motivation

Die Automobilindustrie arbeitet mit Hochdruck daran, autonomes Fahren zu ermöglichen. So hat aktuell Audi einen fahrerlosen Rennwagen vorgestellt [aud]. Im Zuge dieser Automatisierung werden auch neue Technologien für konventionelle Fahrzeuge entwickelt. Eine solche Entwicklung ist die Nutzung von Fahrzeug-zu-Fahrzeug-Kommunikation (C2C Communication) vor allem für Sicherheitsanwendungen. So soll zum Beispiel ermöglicht werden, dass Autos automatisch nachfolgende Wagen über Staus oder Gefahrenstellen informieren und damit als Frühwarnsystem fungieren. Der Lehrstuhl Rechnernetze entwickelt im Rahmen der Forschung zur C2C Communication einen Simulator für mobilfunkgestützte Kommunikation, der auf Messungen realer Mobilfunknetze basiert. Damit soll es Fahrzeugherstellern ermöglicht werden, auf einfache und kostengünstige Weise zu überprüfen, ob neu entwickelte Anwendungen in Mobilfunknetzen umsetzbar sind.
In vorangegangenen Arbeiten wurde ein Messframework zur Latenz- und Datenratenmessung in Mobilfunknetzwerken für x86 Systeme in C/C++ entwickelt und dieses auf Android portiert. Der Androidversion steht allerdings kein hochgenaues Pulse-Per-Second-Signal zur Verfügung, weil die in den verwendeten Geräten verbauten GPS-Empfänger dieses nicht durchreichen. Da das PPS-Signal für die Zeitsynchronisation benötigt wird, um sinnvolle Latenzmessungen zu ermöglichen, wird nun nach einer Möglichkeit gesucht, dem Androidgerät ein externes PPS-Signal zur Verfügung zu stellen. Aufgrund der begrenzten Anschlussmöglichkeiten eines Smartphones fiel die Wahl auf den Audioanschluss als Verbindungsmöglichkeit, da dieser relativ geringe Eigenlatenzen

verspricht.
Im Rahmen der vorliegenden Arbeit wurde ein Poof of Concept entwickelt, das aufzeigt, dass die Übertragung eines PPS-Signals mitsamt den zugehörigen GPS-Informationen über Audio realisierbar ist. Daran anknüpfend wurde die Qualität der implementierten Übertragungslösung evaluiert.

Kapitel 2

Die schnelle Fouriertransformation

2.1 Grundlagen der Audiosignalverarbeitung

Obwohl Audiosignale in der Realität kontinuierlich sind, werden sie im Computer als eine diskrete Folge von Werten betrachtet. Für die Diskretisierung des Signals werden Analog-Digital-Wandler (ADCs) verwendet. Diese tasten das analoge Signal in gleichmäßigen Abständen ab und liefern eine digitale Wertefolge zurück. Die Häufigkeit, mit der das Signal abgetastet wird, bezeichnet man als Abtastrate bzw. Samplingrate. Sie beschränkt nach dem Nyquist-Abtasttheorem (vgl. [OS92, S.96]) die Frequenzen, die erkannt werden können. Es gilt:

$$f_{tast} > 2 \cdot f_{max}$$

Das heißt, man muss ein Signal, das eine maximale Frequenz f_{max} enthält, mit einer Rate von mindestens dem Doppelten dieser höchsten Frequenz abtasten, um keine Informationen zu verlieren. Verlorene Informationen würden dazu führen, dass das Ursprungssignal nicht mehr fehlerfrei aus der durch die Abtastung entstandenen Wertereihe rekonstruiert werden kann (vgl. [Hof11, S.39]). In der Praxis wählt man meist einen höheren Wert. So werden CDs, auf denen Signale mit einer Maximalfrequenz von 20kHz abgespeichert werden, mit 44,1kHz abgetastet. Hier entspricht die Abtastrate also ungefähr dem 2,2-fachen der Maximalfrequenz.

Für die Umwandlung von diskreten Wertefolgen in analoge Signale steht in einem PC

typischerweise ein Digital-Analog-Wandler (DAC) zur Verfügung. Auch hier gilt, dass die maximale im erzeugten Signal enthaltene Frequenz f_{max} immer kleiner als die Hälfte der Samplingrate ist.

Um Daten per Audio übertragen zu können, bietet es sich an, verschiedene Werte als unterschiedliche Tonhöhen bzw. Frequenzen zu kodieren. Ein Algorithmus zur Bestimmung der in einem Audiosignal enthaltenen Frequenzen ist die *Fouriertransformation*.

2.2 Die Fouriertransformation

Die Fouriertransformation transformiert ein Signal vom Zeitbereich in den Frequenzbereich. Dabei wandelt die diskrete Fouriertransformation (kurz DFT) n Abtastwerte in n Spektrallinien um. „Sie ist damit die einzige Transformation aus der Familie der Frequenzanalyseverfahren, die sowohl im Zeitbereich als auch im Frequenzbereich diskrete Werte verarbeitet bzw. erzeugt. Deshalb ist sie das bevorzugte Verfahren für die computergestützte Frequenzanalyse“ [Hof11, S.43] und kommt auch in dieser Arbeit zur Anwendung.

Zu einem Vektor $a = (a_0, a_1, ..., a_{n-1})$ von n Abtastwerten ist die Fouriertransformierte gegeben als der Vektor $y = (y_0, y_1, ..., y_{n-1})$ für den gilt:

$$y_k = \sum_{j=0}^{n-1} a_j e^{ijk\frac{2\pi}{n}} \qquad \text{([Gur14, S.55])}$$

Jede dieser Frequenzlinien y_k gibt den Pegel eines Frequenzbereichs an, mit dem Frequenzen aus diesem Bereich im abgetasteten Ausgangssignal enthalten sind. Da sie nicht für diskrete Frequenzen stehen, werden die Frequenzlinien auch als Frequenzkörbe oder (Frequenz-)Bins bezeichnet. Wie groß der Frequenzbereich eines solchen Frequenzkorbs ist, hängt von der Anzahl der Abtastwerte n und der Abtastrate r ab. Es gilt:

$$Breite\ der\ Frequenzlinie = \frac{r}{n}$$

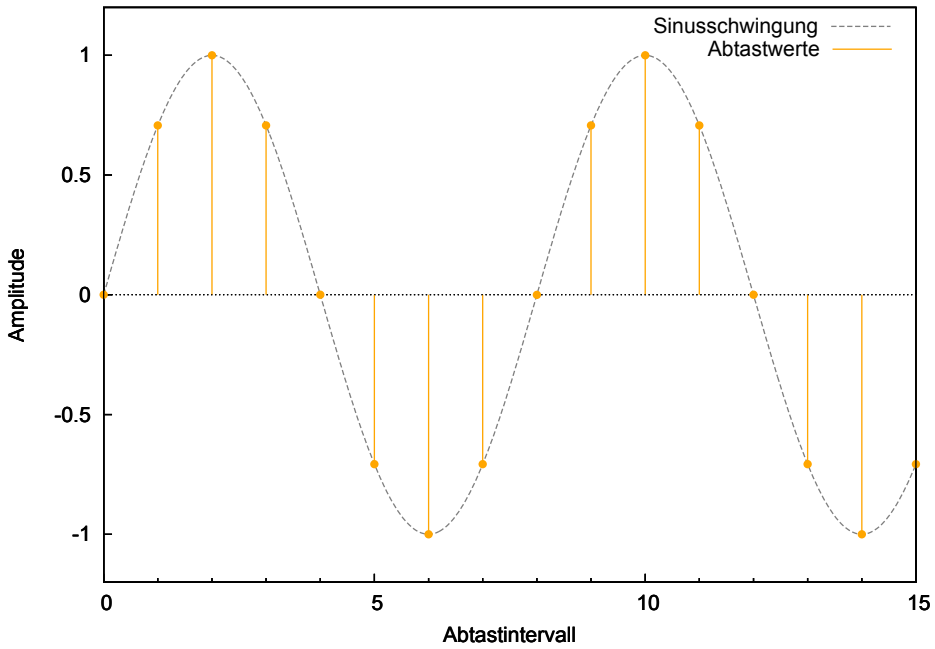

Abbildung 2.1: Abtastung einer Sinusschwingung

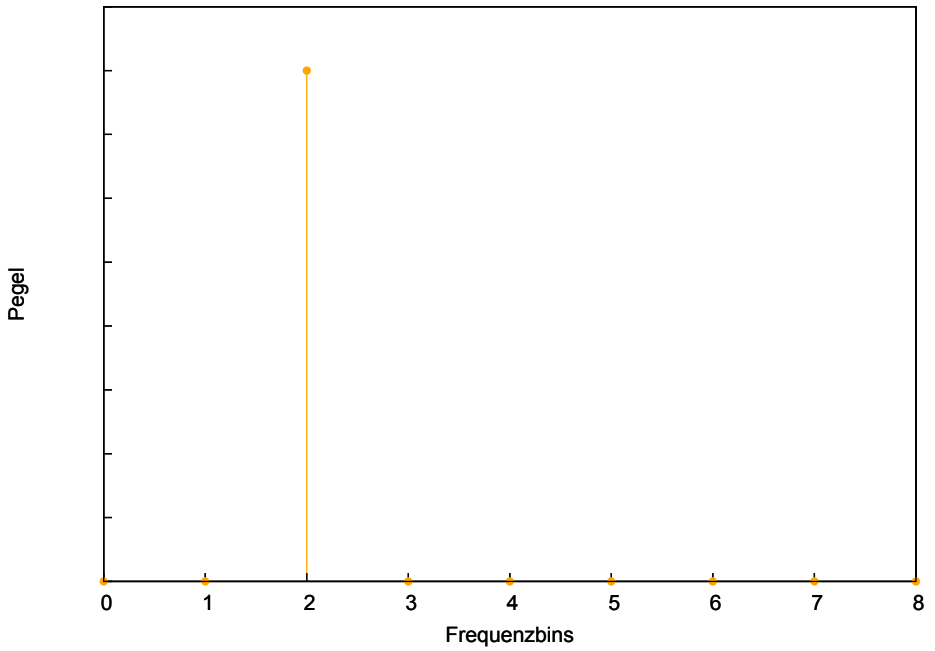

Abbildung 2.2: Werte aus Abbildung 2.1 fouriertransformiert

Generell arbeitet die Fouriertransformation im komplexen Zahlenraum. Dadurch, dass die Abtastwerte eines Audiosignals alle im Reellen liegen, ergibt sich jedoch folgende Vereinfachung:

$$y_{n-k} = y_{-k} = y_k^* \qquad \text{(vgl. [Hof11, S.44])}$$

Dabei ist a^* die komplex Konjugierte von a. Somit reicht es, die Hälfte der Werte per Fouriertransformation zu errechnen. Die übrigen Wert lassen sich dann durch Konjugation trivial berechnen. Diese sind allerdings für diese Arbeit nicht weiter von Interesse, da hier jeweils der Betrag der Frequenzlinien betrachtet wird und dieser mit dem Betrag der komplex Konjugierten übereinstimmt. Es ergeben sich also aus n Abtastwerten $\lfloor \frac{n}{2} \rfloor + 1$ unterschiedliche Beträge im Frequenzspektrum. Dies deckt sich mit dem Abtasttheorem2.1. Bei einer Abtastrate von $r = 100Hz$ und genau so vielen Abtastwerten erhält man 51 Frequenzlinien mit unterschiedlichen Beträgen, die jeweils eine Breite von $1Hz$ haben und den Bereich von 0Hz bis 50Hz abdecken. Alle weiteren Frequenzlinien haben für die Frequenzanalyse keine Bedeutung, da sich alle durch die Transformation gewonnenen Informationen aus diesen 51 Linien ableiten lassen.

In den Abbildungen 2.1 und 2.2 sind diese Zusammenhänge noch einmal verdeutlicht. In Abbildung 2.1 wird eine Schwingung mit genau zwei Perioden über das Abtastintervall an 16 Stellen abgetastet. Deshalb zeigt sich in Abbildung 2.2 bei den 9 transformierten Werten, von denen jeweils der Betrag errechnet wurde, der Ausschlag in Frequenzbin 2.

Die diskrete Fouriertransformation funktioniert allerdings nur dann perfekt, wenn das Abtastintervall einem ganzzahligen Vielfachen der Periode des Signals entspricht. Ist dies nicht der Fall, kommt es zu Leckeffekten. Das heißt, um die richtige Frequenzlinie verteilen sich weitere, niedrigere Pegel, wie in Abbildung 2.3 dargestellt.

Der Grund dafür ist, dass die Fouriertransformation damit rechnet, dass sich das abgetastete Intervall periodisch fortsetzt. Dies kann leicht dazu führen, dass in dem virtuellen Signal, mit dem die Transformation rechnet, Unstetigkeitsstellen auftreten, was dann zu besagten Leckeffekten führt. Abbildung 2.4 zeigt ein Beispiel für diese unstetige Fortsetzung eines Abtastintervalls.

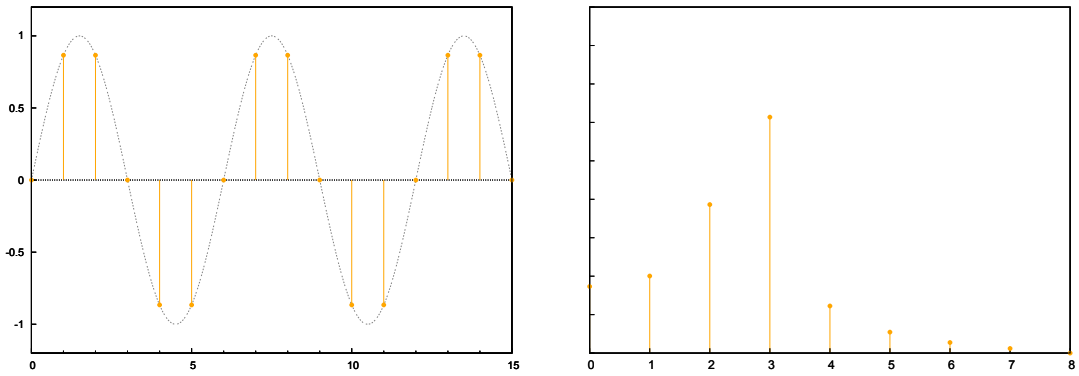

Abbildung 2.3: Fouriertransformation von 2,66 Perioden mit Leckeffekt

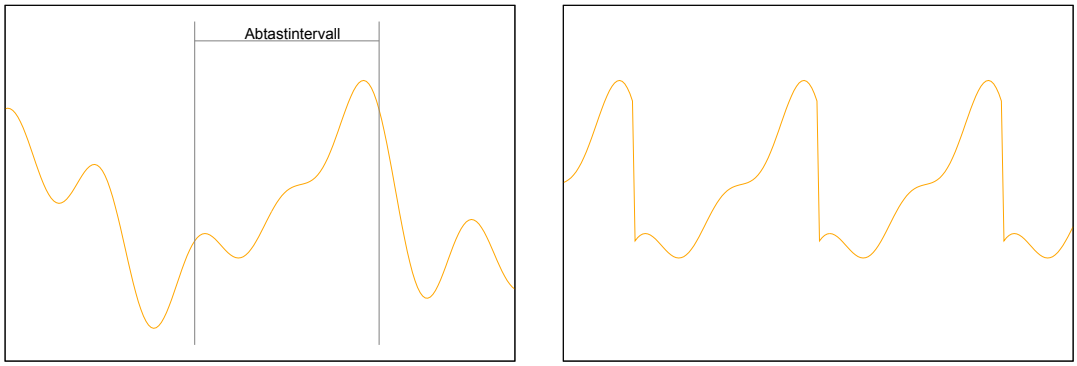

Abbildung 2.4: Periodische Fortsetzung des Abtastintervalls
(Abtastwerte sind aus Gründen der Anschaulichkeit kontinuierlich dargestellt.)

Um die Leckeffekte zu minimieren, können die Abtastwerte, die transformiert werden sollen, mit einer Fensterfunktion multipliziert werden. Fensterfunktionen versuchen künstlich Periodizität herzustellen, indem sie das Signal und seine Ableitungen an den Rändern des Abtastintervalls gegen Null laufen lassen. In Abbildung 2.5 ist links eine solche Fensterfunktion dargestellt. Auf der rechten Seite wurde diese Fensterfunktion mit den Abtastwerten aus Abbildung 2.3 multipliziert, sodass das Signal an den Rändern sichtbar abgeflacht ist.

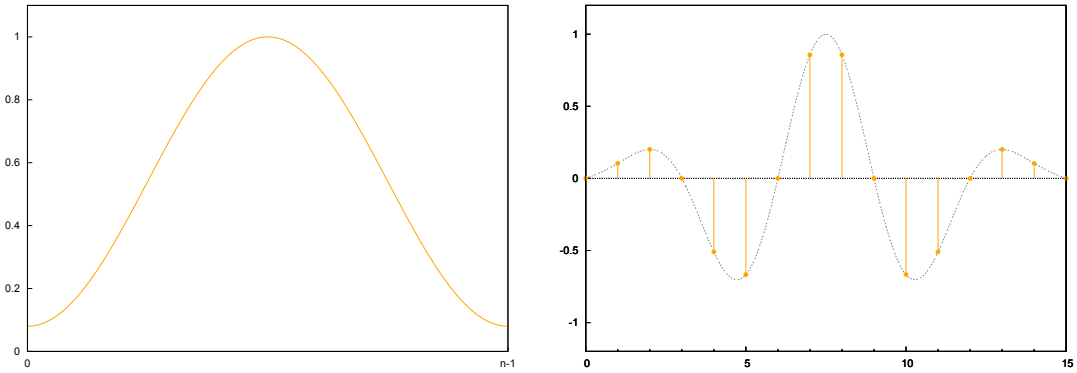

Abbildung 2.5: Eine Fensterfunktion und ihre Anwendung auf das Abtastintervall aus Abbildung 2.3

Ein wichtiger Punkt, der zur weiten Verbreitung der Fouriertransformation geführt hat, ist, dass „effiziente Algorithmen zur expliziten Berechnung der DFT existieren.“ [OS92, S.661] Diese Algorithmen werden unter dem Begriff schnelle Fouriertransformation (fast Fourier transform FFT) zusammengefasst. Wie diese im Einzelnen arbeiten, ist für die vorliegende Arbeit nicht von Bedeutung, da die für die Transformationen verwendete Bibliothek automatisch einen möglichst effizienten Algorithmus auswählt. Mehr dazu in Kapitel 3.2.1.

Die Fouriertransformation lässt sich nicht nur dazu verwenden, Daten vom Zeit- ins Frequenzspektrum zu transformieren, auch die Gegenrichtung ist möglich. Dafür kommt die *inverse Fouriertransformation* zum Einsatz. Ihre Formel lautet:

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j e^{-ijk\frac{2\pi}{n}} \qquad \text{(vgl. [Hof11, S.42f])}$$

Offensichtlich unterscheiden sich Hin- und Rücktransformation nur marginal, was dazu führt, dass FFT-Algorithmen auch für die inverse Fouriertransformation genutzt werden können. Es ist allerdings zu beachten, dass der Faktor $\frac{1}{n}$ in der Literatur teilweise der Hin- statt der Rücktransfromation zugeordnet wird.

Kapitel 3

Umsetzung

Das Ziel dieser Arbeit war es, einen Puls pro Sekunde *PPS*, der von einem externen GPS-Empfänger bereitgestellt wird, einem PC über Audio zur Verfügung zu stellen. Ein PPS ist ein Signal, das eine scharfe (entweder steigende oder fallende) Flanke hat, die mit einem Abstand von exakt einer Sekunde auftritt. Im Rahmen dieser Arbeit sollte damit die Möglichkeit geschaffen werden, zu evaluieren, ob die Zeitsynchronisation eines Geräte über einen per Audiokabel angeschlossenen GPS-Empfänger realisierbar ist. In Abbildung 3.1 ist der grundsätzliche Aufbau dafür veranschaulicht. Als Empfänger der Audiodaten wurde für diesen Prototypen ein normaler PC verwendet.

3.1 Auswahl der Hardware

Für die Codierung der Daten in Audio standen verschiedene mögliche Plattformen zur Auswahl.
Eine Option war die Nutzung der Arduino-Plattform [ard]. Arduino ist eine Open Source Elektronik-Plattform bestehend aus leicht nutzbarer Hard- und Software. Sie kann mit Hilfe der Programmiersprache Arduino gesteuert werden, die extra für diese Plattform entwickelt wurde.
Genutzt wurde letztendlich aber ein Raspberry Pi in Verbindung mit einem GPS-Shield.

[1] Lizenzhinweis: Die in diesem Bild verwendten Icons stammen von der Website www.freepik.com und wurden unter Creative Commons BY 3.0 zur Verfügung gestellt.

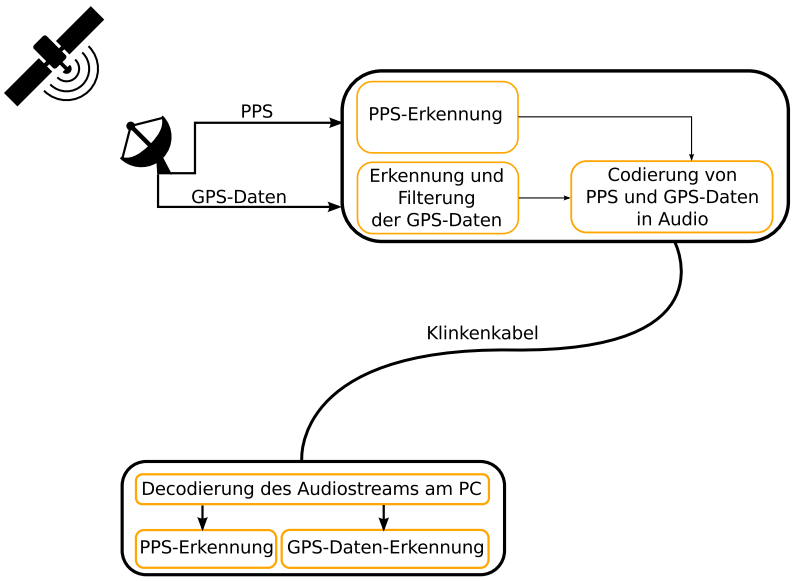

Abbildung 3.1: Der grundsätzliche Aufbau[1]

Der Raspberry Pi [ras] ist ein vollwertiger Kleincomputer auf ARM-Basis, der deutlich mehr Rechenleistung als ein Arduinoboard bietet. Er unterstützt alle gängigen Hochsprachen, was die Implementierung der Software erleichterte, da Bibliotheken wie FFTW 3.2.1 und PortAudio 3.2.2 damit sowohl auf Sender-, als auch auf Empfängerseite genutzt werden konnten. Wichtiger war allerdings die höhere Rechenleistung des Raspberry Pi, der somit eine kleinere Rolle in der Verzögerung der Übertragung spielt, als dies bei einem Arduino der Fall gewesen wäre.
Außerdem ist für den Raspberry Pi ein vorgefertigtes GPS-Shield verfügbar, für dessen Nutzung ausführliche Erläuterungen vorhanden sind [sat]. Dieses Shield wird über die GPIO-Pins des Raspberry Pi mit diesem verbunden. Auf ihm ist ein Ublox MAX-7Q GPS-Modul verbaut und es bietet einen Anschluss für eine aktive Antenne und eine Kontroll-LED für das PPS-Signal.
Damit sind alle Komponenten für den Hardwareaufbau gefunden. Die GPS-Daten und der PPS werden vom Raspberry Pi in Audiodaten umgerechnet und dann über ein Klinkenkabel zu einem anderen PC übertragen, wo die Audiodaten aufgezeichnet und decodiert werden können.

3.2 Verwendete Bibliotheken

In der entwickelten Software werden verschiedene Bibliotheken verwendet, die an dieser Stelle vorgestellt werden sollen.

3.2.1 FFT-Bibliotheken

Für die Frequenzanalyse der auf dem PC eingehenden Daten wurde auf die schnelle Fouriertransformation zurückgegriffen. Es existieren verschiedene Bibliotheken, die einen oder mehrere FFT-Algorithmen umsetzen. In dieser Arbeit kam die FFTW-Bibliothek zum Einsatz, die der De-Facto-Standard unter den FFT-Bibliotheken ist (vgl. [fft]). Sie bietet sehr schnelle Berechnungen und eine hervorragende Dokumentation in Form eines ausführlichen Handbuchs. Als Alternativen boten sich Kiss FFT [kis] oder FFTSS [fft] an.

Kiss FFT ist eine Bibliothek, die unter dem Grundsatz der Einfachheit entwickelt wurde. Der Vorteil dieser Bibliothek liegt darin, dass sie sehr schlank gehalten ist und deshalb nicht viel Speicherplatz verbraucht. Gegen sie spricht, dass sie in den meisten Anwendungsfällen deutlich langsamer als FFTW ist [spe].

FFTSS ist eine FFT-Bibliothek, die für großangelegte wissenschaftliche Anwendungen gedacht ist. Sie nutzt Fähigkeiten moderner Prozessoren wie SSE3 und ist so bis zu 18% schneller als FFTW [Nuk06]. Allerdings bietet sie nur Algorithmen für komplexwertige double-Arrays mit 2^n Werten. Da in dieser Arbeit aber reellwertige float-Arrays verwendet werden und sich damit ungefähr die Hälfte der Berechnungen einsparen lässt (siehe Kapitel 2.2), ist dieser Vorteil nicht nutzbar.

Die FFTW-Bibliothek

Die FFTW-Bibliothek erreicht ihre hohe Berechnungsgeschwindigkeit, indem sie vor der eigentlichen Berechnung der Transformation das System analysiert, auf dem sie aus-

geführt werden soll. Dabei bestimmt sie unter anderem, welchen Befehlssatz der Prozessor unterstützt. Anschließend wird der Algorithmus dem System und der Größe von Eingabe- beziehungsweise Ausgabearray so anpasst, dass er die Fähigkeiten des Systems möglichst gut ausnutzt. Diese Analyse des Systems und Anpassung des Algorithmus wird von FFTW als *planning* bezeichnet und erzeugt eine Datenstruktur, den *plan*. Dieser plan kann dann beliebig oft ausgeführt werden, allerdings werden Eingabe- und Ausgabearray bei der Erstellung des plans festgelegt und können im Nachhinein nicht mehr geändert werden. Der Inhalt der Arrays lässt sich aber noch nachträglich ändern. Diese Einschränkung stellt für die vorliegende Arbeit aber kein Problem dar, da immer nur die aktuellsten Daten von Interesse sind. Einmal transformierte und analysierte Daten können also problemlos überschrieben werden.
Insgesamt garantiert FFTW eine Berechnungsdauer in $\mathcal{O}(n \log n)$ für beliebige Arraygrößen, ist aber am schnellsten, wenn sich die Größe in kleine Primfaktoren zerlegen lässt (vgl. [FGJ, S.4]). Die planning-Phase kann dagegen einige Zeit dauern. Im Laufe dieser Arbeit blieb diese Zeit immer im Bereich einiger Sekunden.
FFTW bietet zwar Flags an, um diese Zeit einzuschränken. Dabei muss aber immer ein Kompromiss zwischen möglichst optimalem Algorithmus und möglichst kurzer planning-Phase geschlossen werden. Dieses Problem lässt sich mit der *wisdom*-Funktion von FFTW umgehen. Sie bietet die Möglichkeit, Informationen, die beim planning ermittelt wurden, in einer Datei abzuspeichern, sodass sie bei wiederholtem Programmaufruf nicht neu berechnet werden müssen. Dadurch wird erreicht, dass das Programm nur beim ersten Aufruf eine längere Zeit für den Start benötigt. Bei jedem weiteren Aufruf muss in der planning-Phase nichts mehr berechnet werden. Diese Funktion kommt auch in dem im Laufe dieser Arbeit entstandenen Programm zum Einsatz.

3.2.2 Die PortAudio-Bibliothek

Für die eigentliche Übertragung der Daten per Audio kommt in dieser Arbeit die PortAudio-Bibliothek zum Einsatz, die sich um Audioausgabe und -aufnahme kümmert. Sie stellt eine einfache Schnittstelle bereit und ist plattformunabhängig. Außerdem bietet sie eine ausführliche Dokumentation.

Als Alternative bot sich zum Beispiel die FMOD-Bibliothek [fmo] an, die unter anderem auch Android unterstützt. Sie bietet viele Funktionen, die für Spieleentwickler interessant sind, was dazu führt, dass die Schnittstelle deutlich komplexer ist und die Bibliothek insgesamt recht groß ausfällt - schon in gepacktem Zustand ist sie fast 35MB groß. Diese Komplexität ist in der vorliegenden Arbeit weder nötig, noch gewünscht. Die Entscheidung fiel deshalb gegen die FMOD-Bibliothek.

Die PortAudio-Bibliothek bietet zwei Möglichkeiten um Ton aufzunehmen oder auszugeben.
Die eine Möglichkeit sind blockierende I/O-Funktionen, die sehr leicht zu verstehen sind. Nachdem ein Audiostream geöffnet wurde, können mit *Pa_WriteStream* beziehungsweise *Pa_ReadStream* Daten in diesen geschrieben oder von diesem gelesen werden.
Die zweite Möglichkeit ist der Zugriff auf Audiostreams über Callback-Funktionen. Dabei wird PortAudio beim Öffnen des Audiostreams eine Funktion übergeben. Diese wird dann in einem separaten Thread jedes Mal aufgerufen, sobald neue Audiodaten zur Verfügung stehen bzw. benötigt werden. Der Thread wird nach Möglichkeit mit sehr hoher oder Echtzeit-Priorität ausgeführt. Für eine störungsfreie Tonausgabe bzw. -aufnahme muss deshalb bei der Implementierung der Callback-Funktion darauf geachtet werden, keine möglicherweise blockierenden Funktionen zu verwenden.
Für maximale Geschwindigkeit und Portabilität sollten nach Entwicklerangaben [blo] Callback-Funktionen verwendet werden, weshalb dieser Weg auch für die im Zuge dieser Arbeit entstandenen Programme gewählt wurde. Des weiteren wird bei der PortAudio-Bibliothek ein Ringbuffer mitgeliefert, der für den Empfangsteil verwendet wurde.

3.2.3 Die WiringPi-Bibliothek

Für die Kommunikation zwischen dem GPS-Empfänger und dem Raspberry Pi wurde die WiringPi-Bibliothek [wPi] verwendet. Sie bietet eine einfach nutzbare Schnittstelle zur Interaktion mit den GPIO-Pins des Raspberry Pi. Außerdem bietet sie die Funktion *wiringPiISR*, die ähnlich zum Prinzip der Callbackfunktionen von PortAudio eine selbst geschriebene Funktion übergeben bekommt. Diese wird automatisch immer dann ausgeführt, wenn ein Interrupt an einem festgelegten Pin auftritt. Dabei kann mit dem Parame-

ter *edgeType* festgelegt werden, ob auf steigende oder fallende Flanken reagiert werden soll, oder auch auf beide. Außerdem ist mit Hilfe dieser Bibliothek die Nutzung der seriellen Schnittstelle sehr leicht umsetzbar. Die Bibliothek bietet darüber hinaus noch viele weitere Funktionen, die an das Wiring-Paket für Arduino angelehnt sind. Diese wurden für die vorliegende Arbeit aber nicht gebraucht.

3.3 Der Sender

Aufgabe des Senders ist es, die GPS-Daten und den PPS vom GPS-Empfänger entgegenzunehmen und diese in Audio codiert auszugeben. Das Programm *toa_send* (kurz für time over audio sender), das diese Funktionen übernimmt, ist während des Implementierungsprozesses aufbauend auf dem Quellcode der Datei sine.cxx entstanden, die als Beispiel mit der PortAudio-Bibliothek mitgeliefert wird. Grundsätzlich besteht toa_send aus drei parallel laufenden Threads. Diese haben die folgenden Aufgaben:

Der *pps*-Thread wird mit der *wiringPiISR*-Funktion gestartet. Er wartet auf steigende Flanken am GPIO-Pin 18, über den der PPS übertragen wird, und setzt ein Flag für den PortAudio-Thread, sobald eine solche registriert wird.
Der *PortAudio*-Thread ist für die Audioausgabe zuständig. Er mischt, abhängig von den entsprechenden Flags, den dem PPS zugeordneten Audiostrom und denjenigen, in dem die GPS-Daten codiert sind.
Der *main*-Thread schließlich ist für den Start der anderen Threads zuständig und übernimmt dann das Auslesen der seriellen Schnittstelle, über die die GPS-Daten übertragen werden. Aus diesen Daten werden die für dieses Programm interessanten heraus gefiltert und in Audio codiert. Wenn codierte Daten für die Übertragung bereitstehen, wird ein Flag für den PortAudio-Thread gesetzt, sodass dieser sie in den Audiostrom mischt.

Der Ublox MAX-7Q überträgt standardmäßig 7 unterschiedliche Typen von NMEA Datensätzen über die serielle Schnittstelle (vgl. [ublb, S.21]). NMEA Datensätze sind ein von der National Marine Electronics Association entwickelter und weit verbreiteter Standard zur Kommunikation mit GPS-Empfängern. Der Empfänger verwendet ohne weitere Konfiguration das NMEA Protokoll in der Version 2.3. Für diese Arbeit ist der RMC

(Recommended Minimum data C) der einzige relevante Datensatz. Er enthält Daten über die Zeit, die Position und die Geschwindigkeit zum Zeitpunkt des letzten Zeitimpulses. Der Aufbau und Inhalt eines solchen Datensatzes wird in Tabelle 3.1 dargestellt. Dabei ist oberhalb der Tabelle ein abstraktes Beispiel angegeben. Die durch Kommata und Stern getrennten Felder sind dann in der Tabelle aufgeschlüsselt.

$GPRMC,time,status,lat,NS,long,EW,spd,cog,date,mv,mvEW,posMode*cs<CR><LF>

Feldnr.	*Bedeutung*	*Format/Datentyp*	*mögliche verwendete Zeichen*
0	Identifikation	$GPRMC	$ G P R M C
1	Uhrzeit	hhmmss.ss	0 1 2 3 4 5 6 7 8 9 .
2	Status	char	V A
3	geogr. Breite	ddmm.mmmmm	0 1 2 3 4 5 6 7 8 9 .
4	Nord/Süd	char	N S
5	geogr. Länge	dddmm.mmmmm	0 1 2 3 4 5 6 7 8 9 .
6	Ost/West	char	E W
7	Geschwindigkeit	numerisch	0 1 2 3 4 5 6 7 8 9 .
8	Kurs	numerisch	0 1 2 3 4 5 6 7 8 9 .
9	Datum	ddmmyy	0 1 2 3 4 5 6 7 8 9
10	magn. Abweichung	*von Ublox MAX-7Q nicht unterstützt*	
11	Vorzeichen der Abweichung	*von Ublox MAX-7Q nicht unterstützt*	
12	Positionsmodus	char	N E A D
13	Prüfsumme	hexadezimal	0 1 2 3 4 5 6 7 8 9 A B C D E F

Tabelle 3.1: Der RMC Datensatz [ubla, S.62]

Der main-Thread filtert den RMC aus den über die serielle Schnittstelle kommenden Daten und codiert ihn in Audio. Dazu wurden im Laufe dieser Arbeit zwei Möglichkeiten erdacht.
Die eine Möglichkeit ist, alle Zeichen eines Datensatzes gleichzeitig in den Audiostrom zu codieren, indem jeder Position im Datensatz ein Frequenzband zugeordnet wird. Innerhalb dieser Frequenzbänder wird dann jedem möglichen Zeichen eine bestimmte Frequenz zugeordnet.
Die andere Möglichkeit ist es, jedem möglichen Zeichen eine Frequenz zuzuordnen.

Dann können die den einzelnen Zeichen eines Datensatzes zugeordneten Frequenzen nacheinander ausgegeben werden. Da im RMC nur wenige unterschiedliche Zeichen verwendet werden, reichen entsprechend wenige Frequenzbins und damit auch Samples für die Unterscheidung aus. Es genügt also, die einzelnen Frequenzen nur für eine kurze Zeit zu übertragen.
Für die vorliegende Arbeit wurde die zweite Möglichkeit umgesetzt. Für diese Möglichkeit spricht, dass die Samplereihen für die unterschiedlichen Frequenzen nur einmal berechnet und abzuspeichert werden müssen. Um einen Datensatz in Audio zu codieren, müssen dann nur die gespeicherten Samplereihen an die passende Stelle des auszugebenden Audiosignals kopiert werden. Für die erste Möglichkeit wäre die Berechnung des Audiosignals mittels inverser Fouriertransformation praktikabler, da mit ihrer Hilfe in einem Durchgang alle benötigten Frequenzen fertig gemischt berechnet werden können. Dies hätte aber eine größere Verzögerung der Übertragung zur Folge als das reine Kopieren, wie es umgesetzt wurde.
Da nur RMC Datensätze übertragen werden und keine weiteren, ist es nicht notwendig das Identifikationsfeld mitzusenden. Somit bleiben 23 unterschiedliche Zeichen, für die Frequenzen gewählt werden müssen (vgl. Tabelle 3.1). Das Programm toa_send berechnet beim Start die Samplereihen mit Hilfe der inversen Fouriertransformation, da so ohne weitere Überlegung klar ist, in welchen Frequenzbins auf der Empfangsseite mit einem Ausschlag zu rechnen ist, und somit die Wartung des Codes vereinfacht wird. Konkret wurde für jede Samplereihe eine 256 Werte erzeugende Fouriertransformation verwendet. Für jedes Zeichen wurde in genau einem Bin der reelle Teil auf einen Wert größer Null gesetzt. Es wurde ein Abstand von 3 Bins gewählt, um einen Spielraum für mögliche Fehler bei der Messung zu schaffen. Da ein Ausschlag in Bin x bei der inversen Fouriertransformierten eine Schwingung mit x Perioden auf dem erzeugten Intervall bewirkt, ergibt sich Tabelle 3.2. Als Abtastfrequenz wurde 44,1kHz gewählt, da dies die kleinste Frequenz war, die von allen Geräten, die im Laufe dieser Arbeit verwendet wurden, unterstützt wird.

Ein vollständiger RMC Datensatz wird dann in Audio codiert, indem für jedes Zeichen 256 Abtastwerte der zugeordneten Frequenz gesendet werden und anschließend 256 Samples Nullen. Diese angehängten Nullen erleichtern die Decodierung erheblich. Genaueres dazu findet sich in Kapitel 3.4. In Abbildung 3.2 ist beispielhaft die Zeichenfolge „2014 “auf diese Weise codiert.

Zeichen	*Bin*	*Frequenz*
0	3	516,8 Hz
1	6	1033,6 Hz
2	9	1550,4 Hz
3	12	2067,2 Hz
4	15	2584,0 Hz
5	18	3100,8 Hz
6	21	3617,6 Hz
7	24	4134,4 Hz
8	27	4651,2 Hz
9	30	5168,0 Hz
A	33	5684,8 Hz
B	36	6201,6 Hz
C	39	6718,4 Hz
D	42	7235,2 Hz
E	45	7752,0 Hz
F	48	8268,8 Hz
.	51	8785,5 Hz
,	54	9302,3 Hz
*	57	9819,1 Hz
S	60	10335,9 Hz
W	63	10852,7 Hz
N	66	11369,5 Hz
V	69	11886,3 Hz

Tabelle 3.2: Zeichen und zugeordnete Frequenzen bei 44,1kHz Abtastrate

Laut [gar] ist die maximale Länge eines NMEA Datensatzes inklusive $-Zeichen und Zeilenumbruch 82 Zeichen. Da die Anfangszeichenfolge „$GPRMC,“ und der Zeilenumbruch nicht versandt werden, bleiben noch maximal 74 Zeichen, die übertragen werden müssen. Daraus ergibt sich eine maximale Länge von $74 \cdot 256 \cdot 2 = 37888$ Samples für einen codierten RMC Datensatz, was etwa 859ms bei einer Abtastrate von 44,1kHz entspricht. Es ist also problemlos möglich, auf diese Weise einen Datensatz pro Sekunde zu übertragen.

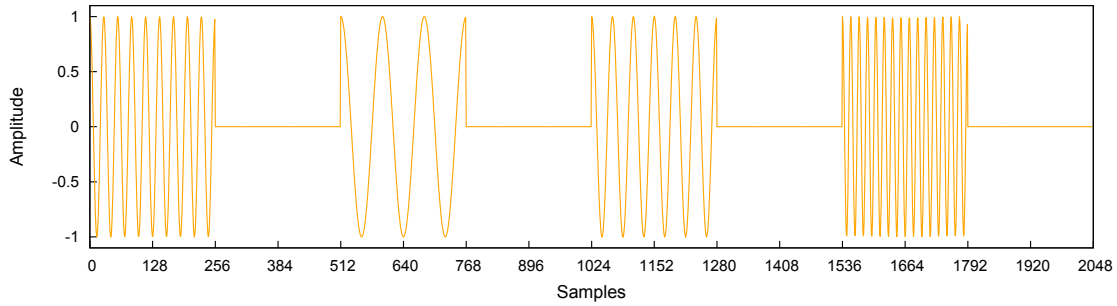

Abbildung 3.2: „2014“ als Audiostrom

Der PPS wird auf ähnliche Weise übertragen. Sobald ein PPS erkannt und somit das Flag gesetzt wurde, werden vorberechnete Werte aus einem Array in den Datenstrom kopiert. Um eine schnelle Erkennung auf Empfängerseite zu ermöglichen, wurde eine hohe Frequenz gewählt. Konkret wurde eine Schwingung mit 100 Perioden über 256 Abtastwerte gewählt, was einer Frequenz von ca. 17227Hz entspricht. Damit ist der Abstand zur maximalen theoretisch erkennbaren Frequenz (vgl. Kapitel 2.1) ausreichend groß, sodass auch in der Praxis eine problemlose Erkennung möglich ist. Die PPS-Frequenz wird immer 512 Samples lang übertragen.

3.4 Der Empfänger

Als Empfänger der Audiodaten dient für diesen Proof of Concept ein normaler PC. Auf ihm läuft das Programm *toa_rcv* (kurz für time over audio reciever), dessen Aufgabe es ist, den ankommenden Audiostrom aufzuzeichnen und die in ihm enthaltenen Informationen zu decodieren. Dieses Programm wurde aufbauend auf dem Beispiel paex_record.c

entwickelt, das bei der PortAudio Bibliothek mitgeliefert wird.
Es besteht aus zwei Threads. Der eine ist der *PortAudio*-Thread, dessen Aufgabe es ist, die aufgezeichneten Abtastwerte in einen Ringbuffer zu schreiben. Ein Ringbuffer ist eine Datenstruktur mit zwei Zeigern, die nach dem FIFO-Prinzip arbeitet und ringförmig aufgebaut ist. Dies bietet den Vorteil, dass nur einmal Speicher alloziert werden muss. Bei einem Ringbuffer kommt es nicht zu einem Überlauf, solange er mit einer höheren Geschwindigkeit ausgelesen wird als er beschrieben wird. Dies ist für unsere Anwendung der Fall, da die Daten nur mit einer Rate von 44,1kHz geschrieben werden. Die Ausleserate dagegen ist abhängig von der Leistung des PCs, ist aber immer deutlich höher als 44,1kHz.
Der andere Thread ist der *main*-Thread, der nach dem Start für die Decodierung der Audiodaten zuständig ist. Er liest 256 Werte vom Ringbuffer, multipliziert sie mit der Hamming-Fensterfunktion und transformiert sie dann mit Hilfe der FFT. Das Hammingfenster ist eine weit verbreitete Fensterfunktion, die die Nebenmaxima im Frequenzspektrum besonders gut unterdrückt. Ihre Formel lautet:

$$h(n) = 0,54 + 0,46 \cdot cos(\frac{2\pi n}{M}) \text{ für } 0 \leq n \leq M \qquad \text{([OS92, S.508])}$$

Dabei ist M die Anzahl der Abtastwerte. Ihr Graph findet sich in Abbildung 2.5 links.
Nach der Fouriertransfomation wird überprüft, ob ein Ausschlag in Frequenzbin 100 vorhanden ist. Ist dies der Fall, bedeutet es, dass die PPS-Frequenz im Audiostrom enthalten ist. Wenn diese Frequenz im vorangegangenen Abtastintervall noch nicht enthalten war, wurde ein neuer PPS erkannt. In diesem Fall wird eine entsprechende Meldung zusammen mit der aktuellen Systemzeit ausgegeben. War die Frequenz dagegen schon im vorherigen Intervall enthalten, handelt es sich nicht um einen neuen PPS und nichts geschieht.
Nach der PPS-Erkennung wird die Ausgabe der FFT noch daraufhin überprüft, ob in dem Intervall ein Zeichen des RMC codiert ist. Ist dies der Fall, wird das entsprechende Zeichen in einem String zwischengespeichert, bis der vollständige Datensatz decodiert ist. Ein Datensatz gilt als vollständig erkannt, wenn nach einem * zwei weitere Zeichen (die Prüfsumme, vgl. Tabelle 3.1) decodiert wurden.

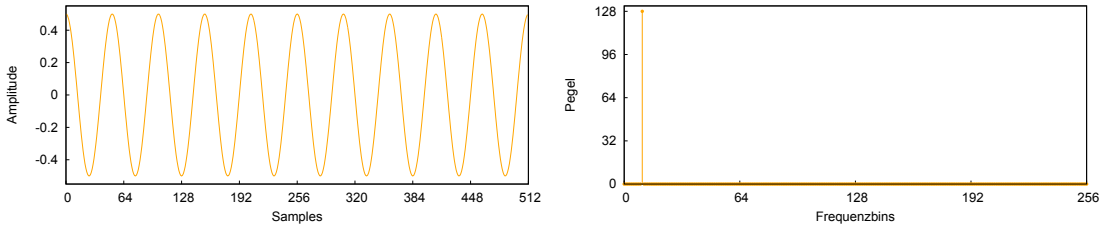

Abbildung 3.3: Eine Schwingung mit 10 Perioden und ihre Fouriertransformierte
(Abtastwerte sind aus Gründen der Anschaulichkeit kontinuierlich dargestellt.)

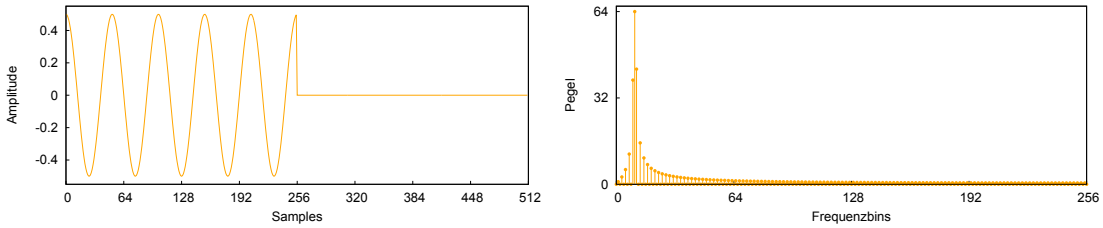

Abbildung 3.4: Die Schwingung aus Abbildung 3.3 mit einer Hälfte Nullen und die Fouriertransformierte dieses Intervalls
(Abtastwerte sind aus Gründen der Anschaulichkeit kontinuierlich dargestellt.)

Bei der Erkennung des PPS und der übertragenen Zeichen ist es von größter Bedeutung, dass die Schwelle gut gewählt wurde, ab der der Ausschlag in einem Frequenzbin auch als solcher gewertet wird. Um dieses Problem vollständig zu verstehen, ist es notwendig zu wissen, wie sich die Fouriertransformation verhält, wenn eine Schwingung nur in einem Teil des Abtastintervalls auftritt und der Rest des Intervalls mit Nullen gefüllt ist. Dieser Fall ist in den Abbildungen 3.3 bis 3.5 dargestellt. Die Fouriertransformierten in diesen Abbildungen wurden mit Hilfe der FFTW-Bibliothek erstellt.

Wie man deutlich sieht, ist der Ausschlag im der Frequenz entsprechenden Bin umso kleiner, je kleiner der Anteil mit Schwingungen im Abtastintervall ist. Außerdem sind um diesen Frequenzbin weitere Ausschläge zu sehen, sobald ein Teil des Abtastintervalls Null ist. Das liegt daran, dass ein Graph, wie er in Abbildung 3.4 links zu sehen ist, mathematisch entsteht, indem man die Schwingung mit der Rechtecksfunktion multipliziert. Nach [Hof11, S.31] gilt, dass die Multiplikation zweier Signale im Zeitspektrum eine Faltung der Signale im Frequenzbereich nach sich zieht.

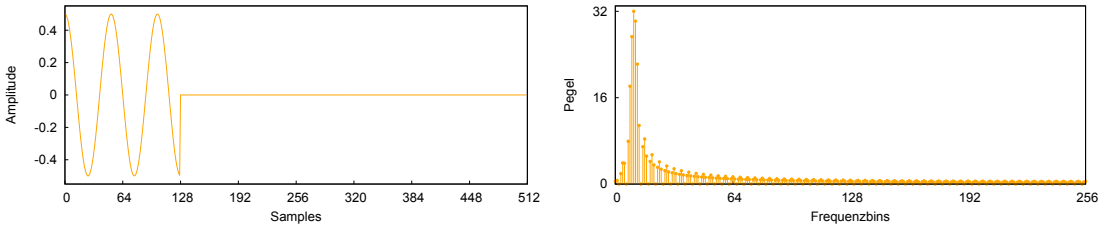

Abbildung 3.5: Die gleiche Schwingung mit $\frac{2}{3}$ Nullen
und die Fouriertransformierte dieses Intervalls
(Abtastwerte sind aus Gründen der Anschaulichkeit kontinuierlich dargestellt.)

Die diskrete Faltung zweier Signale x und y ist wie folgt definiert:

$$z(k) = \sum_{n=-\infty}^{\infty} x(n) \cdot y(k-n)$$

wobei $x(i)$ bzw. $y(i)$ für alle i, für die $x(i)$ bzw. $y(i)$ nicht definiert ist, auf 0 gesetzt wird. In unserem Fall bedeutet das, dass sich um jeden Ausschlag im Frequenzspektrum des abgetasteten Signals das Spektrum der Rechtecksfunktion verteilt. Genau diesen Effekt kann man z.B. in Abbildung 3.4 rechts beobachten.

Da die Abtastintervalle nicht mit dem PPS synchron sind, sollte für eine möglichst frühzeitige PPS-Erkennung die Erkennungsschwelle für die zugeordnete Frequenz niedrig gehalten werden. Dadurch kann sichergestellt werden, dass das Signal auch dann erkannt wird, wenn es erst gegen Ende eines Abtastintervalls beginnt. Andererseits muss die Schwelle hoch genug gewählt werden, dass etwaiges Rauschen, das bei der Übertragung auftritt, nicht als PPS gewertet wird.

Auch die RMC-Übertragung ist nicht mit den Abtastintervallen synchron. Dadurch kann die Frequenz eines übertragenen Zeichens leicht in zwei aufeinander folgenden Intervallen auftreten. Die Schwelle für die Zeichenerkennung muss also hoch genug gewählt werden, dass ein übertragenes Zeichen nicht doppelt erkannt wird. Andererseits muss die Schwelle so niedrig sein, dass ein Zeichen auch im schlechtesten Fall, bei dem sich die Schwingung eines Zeichens jeweils genau zur Hälfte auf zwei Abtastintervalle verteilt, erkannt wird. Diese Schwelle kann also nicht ideal gewählt werden, weil das Zeichen im Worst Case immer entweder doppelt oder gar nicht erkannt wird. Durch Rauschen kann

dieses Problem noch verschärft werden, da es den Frequenzausschlag vermindern oder erhöhen kann. Die Pausen zwischen den Schwingungen der einzelnen Zeichen stellen dabei sicher, dass immer nur die Frequenz höchstens eines Zeichens auf einmal decodiert wird.

Wie hoch die Schwellwerte gewählt werden sollten, hängt maßgeblich von den Lautstärkeeinstellungen im Raspberry Pi und im empfangenden PC ab. Je höher die Lautstärke (und damit auch die Amplituden), umso höher ist auch der Ausschlag in den Frequenzlinien. Um dieses Problem zu beseitigen, wurde im Laufe dieser Arbeit ein Kalibrierungsprogramm entwickelt. Dieses besteht aus zwei Teilen.
Der eine Teil (*calibrate_send*) wird auf dem Raspberry Pi ausgeführt und sendet 10 Sekunden lang die Frequenz, die dem PPS zugeordnet ist. Der andere Teil (*calibrate_rcv*) wird auf dem Empfänger-PC ausgeführt. Er berechnet über 500 Runden die Fouriertransformierte von 256 Abtastwerten und speichert jeweils den Wert von Frequenzbin 100 ab. Abschließend berechnet er den Mittelwert dieser 500 Werte und gibt diesen aus. Dieser Teil zeichnet also $\frac{256*500}{44100} \approx 2,9$ Sekunden lang auf. Der Empfangsteil liefert damit einen Richtwert, wie hoch die Ausschläge zu erwarten sind. Dieser Richtwert muss dann toa_rcv beim Start als Parameter übergeben werden.

Dieser Parameter p wird dann mit 0,7 multipliziert und dient dann als Schwellwert für die RMC-Erkennung. Tests haben gezeigt, dass bei der RMC-Erkennung der Schwellwert für die niedrigen Frequenzen höher gewählt werden muss, um möglichst viele Dopplungen zu vermeiden. Für hohe Frequenzen muss er dagegen niedriger gewählt werden, da sonst teilweise Zeichen verloren gehen. Der Schwellwert wird deshalb je nach Frequenz mit einem Wert zwischen 1,05 und 0,8 multipliziert. Dadurch konnte eine Fehlerrate von 5.9375% bei der RMC-Erkennung erreicht werden. Diese Fehlerrate wurde bei einer Messung über 1280 Sekunden und entsprechend viele RMC-Datensätze bestimmt[2].
Für die PPS-Erkennung wird ein Schwellwert von $0,35 \cdot p$ verwendet, da hier Dopplungen nicht weiter von Bedeutung sind. Etwaiges Rauschen übertraf diesen Schwellenwert während der Testmessungen nicht, sodass eine hundertprozentige Erkennungsrate für den PPS erreicht wurde.

[2]Das entsprechende Log messung1.txt findet sich auf der DVD im Order Messungen

Kapitel 4

Evaluation

Die Fehler, die in der am Ende von Kapitel 3.4 angesprochenen Messung auftraten, spiegeln größtenteils den Fall wider, dass sich die Schwingung eines Zeichens ungefähr hälftig auf zwei Abtastintervalle verteilt. Die Folge ist, dass fast alle Zeichen eines Datensatzes doppelt erkannt werden. Die Ausgabe sieht dann zum Beispiel wie in Listing 4.1 aus[1].

Listing 4.1: Fehler in Messung 1

```
1 $GPRMC,000000665522.0000,,AA,,55111111.667785500,,NN
     ,,00006649.8668779,,EE,,0.011#
2 9,,,,0111111114,,,,,,AA**7766
```

Dabei zeigt das #-Zeichen an, dass hier die Zeile umgebrochen wurde, obwohl der Datensatz noch nicht vollständig decodiert wurde. Es ist also kein decodiertes Zeichen. Dieser Fehler tritt allerdings nicht dauerhaft auf. Der Grund dafür ist wahrscheinlich, dass sich die Abtastraten von Sender und Empfänger leicht unterscheiden. Außerdem läuft die Codierung der Datensätze möglicherweise nicht absolut gleichmäßig ab, da die Datensätze nicht exakt im Abstand von einer Sekunde über die serielle Schnittstelle ankommen.
Solche Dopplungen lassen sich vermeiden, indem das auf ein decodiertes Zeichen folgende Abtastintervall ignoriert wird. Dabei wird ausgenutzt, dass auf jedes Zeichen 256 Samples Stille folgen. Diese Verbesserung wurde in *toa_rcv* implementiert. Dadurch

[1] Beispiel entnommen aus messung1.txt, zu finden auf der DVD im Ordner Messungen

konnte die Fehlerrate in einer neuen Messung auf ungefähr 0,06% gesenkt werden[2]. Für diese Messung wurden 1665 Datensätze decodiert, was einer Laufzeit von knapp 28 Minuten entspricht. Dabei trat nur ein einziger Fehler auf. Der Logausschnitt in Listing 4.2 zeigt diesen Fehler.

Listing 4.2: Fehler in Messung 2

```
1 pps interrupt occured. time: 2014-11-01 18:49:28.037467
2 $GPRMC,174928.00,A,5111.67566,N,00649.87379,E,0.083,,011114,,,A*77
3 pps interrupt occured. time: 2014-11-01 18:49:29.047557
4 pps interrupt occured. time: 2014-11-01 18:49:30.046347
5 $GPRMC,174929.00,A,5111.67557,N,00649.87387,E,0.066,,011114,,,
     AE174930.00,A,5111.#
6 67556,N,00649.87397,E,0.081,,011114,,,A*7F
7 pps interrupt occured. time: 2014-11-01 18:49:31.038986
8 $GPRMC,174931.00,A,5111.67552,N,00649.87404,E,0.018,,011114,,,A*77
```

Es ist erkennbar, dass das Zeichen * und eine Ziffer der Prüfsumme verloren ging. Dadurch wurde der Datensatz vom Programm nicht als vollständig erkannt und deshalb erst ausgegeben, als der darauf folgende Datensatz entschlüsselt war. Fehler dieser Art lassen sich vermeiden, indem die Schwelle niedriger gesetzt wird, ab der eine Frequenz als erkannt gilt. Da Dopplungen keine Rolle mehr spielen, muss die Schwelle nur noch verhindern, dass etwaiges Rauschen als Signal gewertet wird. Sie wurde deshalb auf die Hälfte des Übergabeparameters gesetzt. In zwei anschließenden Messungen wurden Fehlerraten von circa 0,19% über 1563 Datensätze[3] und 0,00% über 1031 Datensätze[4] erreicht. Die Fehler in Messung 3 folgten alle dem gleichen Schema. Ein Beispiel hierzu findet sich in Listing 4.3.

[2]Das zugehörige Log findet sich auf der DVD unter Messungen/messung2.txt
[3]Das zugehörige Log findet sich auf der DVD unter Messungen/messung3.txt
[4]Das zugehörige Log findet sich auf der DVD unter Messungen/messung4.txt

Listing 4.3: Fehler in Messung 3

```
1 pps interrupt occured. time: 2014-11-02 00:33:17.054008
2 $GPRMC,233317.00,A,5111.67839,N,00649.87460,E,0.070,,011114,,,A*75
3 pps interrupt occured. time: 2014-11-02 00:33:18.053504
4 pps interrupt occured. time: 2014-11-02 00:33:19.044235
5 $GPRMC,233318.00,A,5111.67866,N,00649.87491,E,0.856,42.75,011114,,,
     A*52
6 $GPRMC,33319.00,A,5111.67883,N,00649.87515,E,0.759,,011114,,,A*75
7 pps interrupt occured. time: 2014-11-02 00:33:20.038500
8 $GPRMC,233320.00,A,5111.67874,N,00649.87518,E,0.110,,011114,,,A*71
9 pps interrupt occured. time: 2014-11-02 00:33:21.053305
```

Es ist zu erkennen, dass der Datensatz in Zeile 5 so spät decodiert wurde, dass zwischenzeitlich ein neuer PPS registriert wurde. Der Fehler entsteht, weil die 2, die zum Anfang des folgenden Datensatzes gehört, als Teil der Prüfsumme interpretiert wurde. Dieser Fehler ließe sich wahrscheinlich dadurch beheben, dass ein Datensatz als vollständig interpretiert wird, sobald aus mehr als 2 Abtastintervallen hintereinander kein Zeichen decodiert werden konnte. Um dies zu implementieren und zu testen, fehlte leider die Zeit.
Die drei Fälle, in denen ein Datensatz falsch decodiert wurde, weil die führende Ziffer 2 zum vorhergehenden Datensatz gezählt wurde, waren in dieser Messung nicht die einzigen Fälle von verzögerten Datensätzen. Insgesamt wurden 11 Datensätze (ca. 0.7%) erst nach dem darauf folgenden PPS vollständig decodiert. Grund dafür ist vermutlich, dass der RMC an der seriellen Schnittstelle erst einige Zeit nach dem PPS zur Verfügung steht. Wenn dieser zeitliche Abstand zu groß ist, bleibt nicht mehr genug Zeit, um den Datensatz, der codiert eine maximale Länge von etwa 859ms hat, rechtzeitig zu übertragen. Gegen diese Überlegung spricht allerdings, das die Verzögerung nicht hauptsächlich bei längeren Datensätzen auftritt.
Auch in den anderen Messungen traten solche Verzögerungen auf. In Messung 1 waren es 7, was ungefähr 0.55% entspricht, in Messung 2 gab es 3 (ca. 0.18%) Verzögerungen und in Messung 4 wurden 7 Datensätze verspätet decodiert, was einem Anteil von etwa 0.67% entspricht. Da bekannt ist, dass exakt zu Beginn jeder Sekunde ein Puls übertragen wird, ist es nicht erforderlich, den zugehörigen Datensatz zu jedem Puls zu kennen, um diesen dem richtigen Zeitpunkt zuzuordnen zu können. Somit verhindern diese Verzögerungen eine stabile Zeitsynchronisation nicht.

Für eine solche Synchronisation ist es allerdings wichtig, dass die Zeit zwischen dem tatsächlichen Puls des GPS-Empfängers und dem erfolgreichen Decodieren durch toa_rcv möglichst kurz und vor allem möglichst konstant ist. Diese Verzögerung wurde in den oben genannten Messungen 3 und 4 evaluiert. Dazu wurde auf dem Raspberry Pi ein NTP-Server eingerichtet. *NTP* ist ein Protokoll zur netzwerkgestützten Zeitsynchronisation [ntp]. Der Server wurde mit deutschen NTP-Poolservern und, mit Hilfe des Programms *rpi_gpio_ntp* [rpi], mit dem PPS des GPS-Empfängers synchronisiert. Auf dem Empfänger-Rechner, der per Ethernet an den Raspberry Pi angeschlossen wurde, wurde ein NTP-Client eingerichtet, der sich mit dem NTP-Server auf dem Raspberry Pi verband. Auf diese Weise wurde die Systemzeit des Rechners, auf dem toa_rcv lief, sehr genau mit dem PPS synchronisiert.
Dies war so schon für Messung 2 eingerichtet, allerdings existierte bis dahin noch ein Fehler in toa_rcv, der dazu führte, dass die CPU zu 100% ausgelastet wurde. Das wiederum behinderte den NTP-Client, der sich nicht mehr richtig synchronisieren konnte. Deshalb ist Messung 2 für die Evaluation der Übertragungslatenzen unbrauchbar. Für Messung 3 wurde dieser Fehler behoben. In Abbildung 4.1 sind die Verzögerungen, bis der PPS von toa_rcv erkannt wurde, gegen die Zeit aufgetragen und in Abbildung 4.2 ist die Verteilung dieser Verzögerungen veranschaulicht. Die Verzögerungen entsprechen dabei dem Nachkommateil in der Zeitangabe, wie sie zum Beispiel in der ersten Zeile von Listing 4.3 zusehen ist. Diese wurden mit Hilfe der Funktion *gettimeofday* aus der Standardbibliothek berechnet, die die Systemzeit abruft.

In Abbildung 4.2 sieht man, dass ca. 85% der Verzögerungen in einem 15ms breiten Intervall von 39ms bis 54ms liegen. Abbildung 4.1 zeigt, dass die Verzögerungen stark schwanken. Unterschiede von über 10ms zwischen zwei aufeinander folgenden Pulsen treten häufig auf. Über einen längeren Zeitraum betrachtet kann man aber ein Muster erkennen. Die Verzögerung der Pulse fällt immer wieder über einen Zeitraum ca. fünf Minuten von durchschnittlich grob 52ms auf etwa 44ms ab, um dann wieder sprunghaft anzusteigen. Eine Idee, woran diese starken Schwankungen liegen könnten, ist, dass der PPS nicht mit den Abtastintervallen synchron ist. Bei einer Abtastrate von 44,1kHz hat ein Abtastintervall mit 256 Werten allerdings eine Länge von 5,8ms. Da die Schwankun-

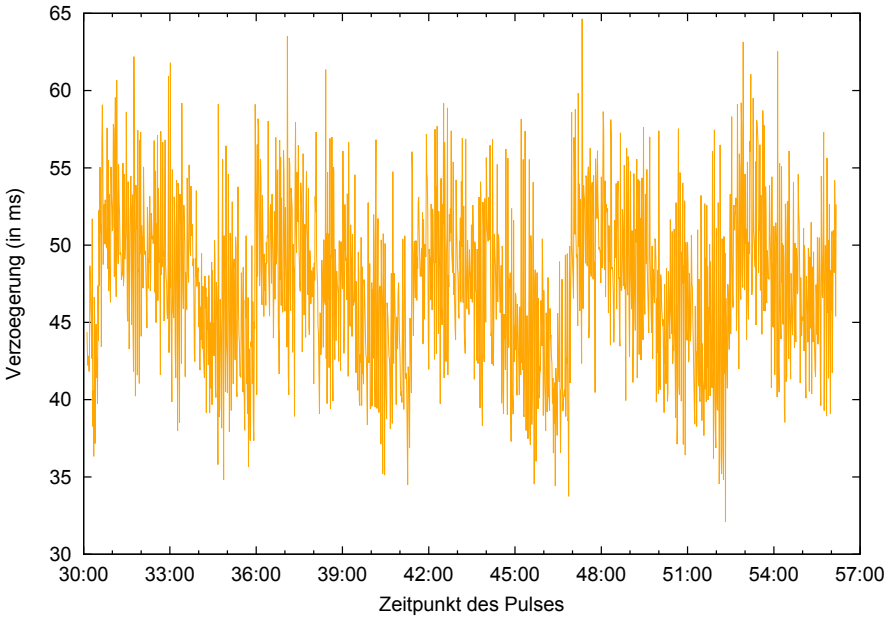

Abbildung 4.1: PPS-Verzögerungen in Messung 3

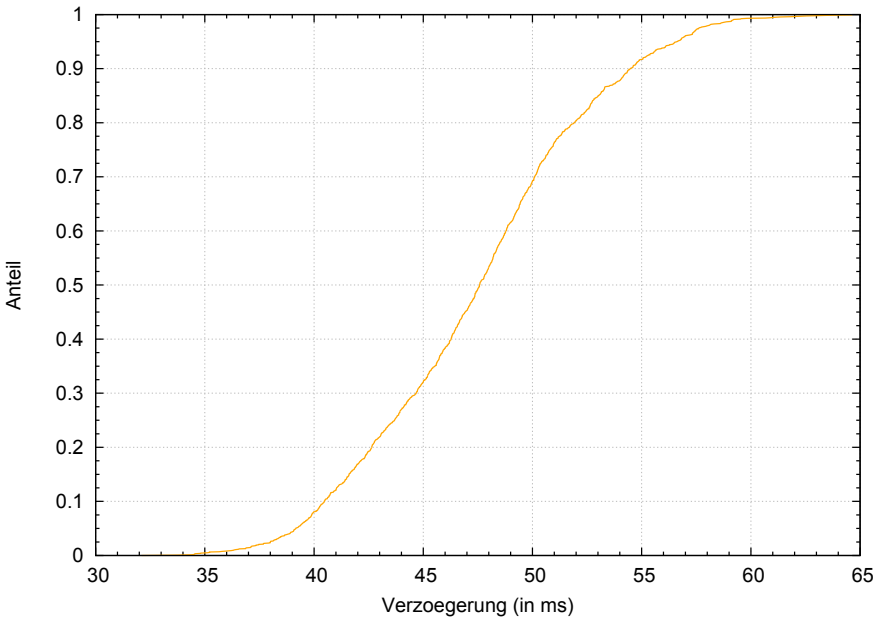

Abbildung 4.2: Verteilung der PPS-Verzögerungen in Messung 3

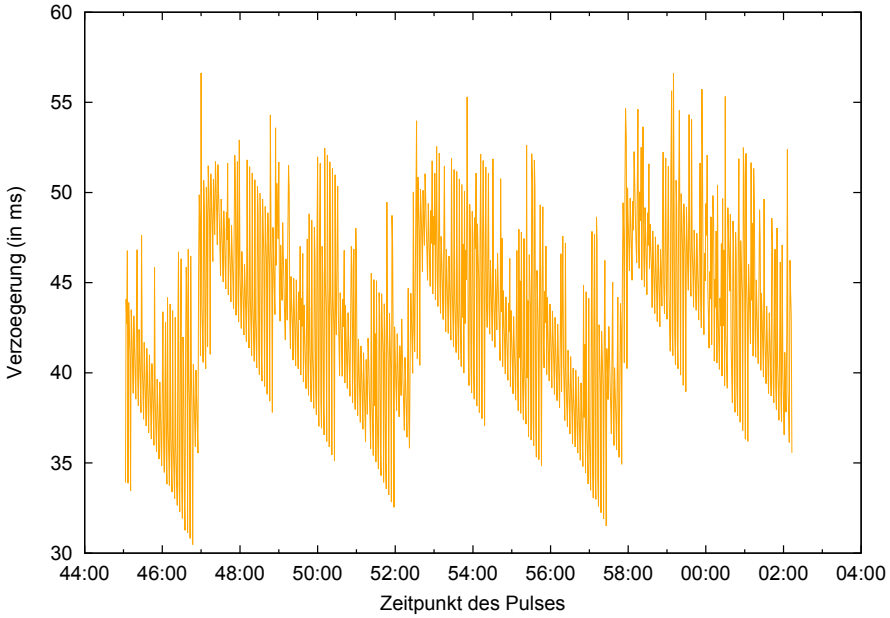

Abbildung 4.3: PPS-Verzögerungen in Messung 4

gen aber deutlich größer sind, kann dies auf keinen Fall der einzige Faktor sein.
Für Messung 4 wurde eine Synchronisation der Decodierung mit dem PortAudio-Thread implementiert. Zuvor wartete der Decodierungs-Thread einfach nur eine gewisse Zeit, wenn keine Daten zur Analyse verfügbar waren.

Der Unterschied ist in Abbildung 4.3 gut zu sehen. Hier wurden wieder die Verzögerungen, bis der PPS von toa_rcv erkannt wurde, gegen die Zeit aufgetragen. Diesmal wirken die Werte deutlich geordneter. Auch bei dieser Messung fallen die Werte über ein Intervall von ungefähr 5 Minuten um dann sprunghaft wieder anzusteigen. Nun wird allerdings sichtbar, dass sich diese Intervalle selbst noch einmal in 3 Teilintervalle aufteilen, die das gleiche, in der Höhe verschobene, Bild zeigen.

Abbildung 4.4 zeigt die gleichen Daten wie Abbildung 4.3 nur mit Punkten statt Linien. Hier wird deutlich, dass sich die Werte in jedem Teilintervall hauptsächlich auf 4 Linien verteilen, zwischen denen die Verzögerungen wechseln. Diese Linien sind in der Abbildung für das erste Teilintervall hervorgehoben.

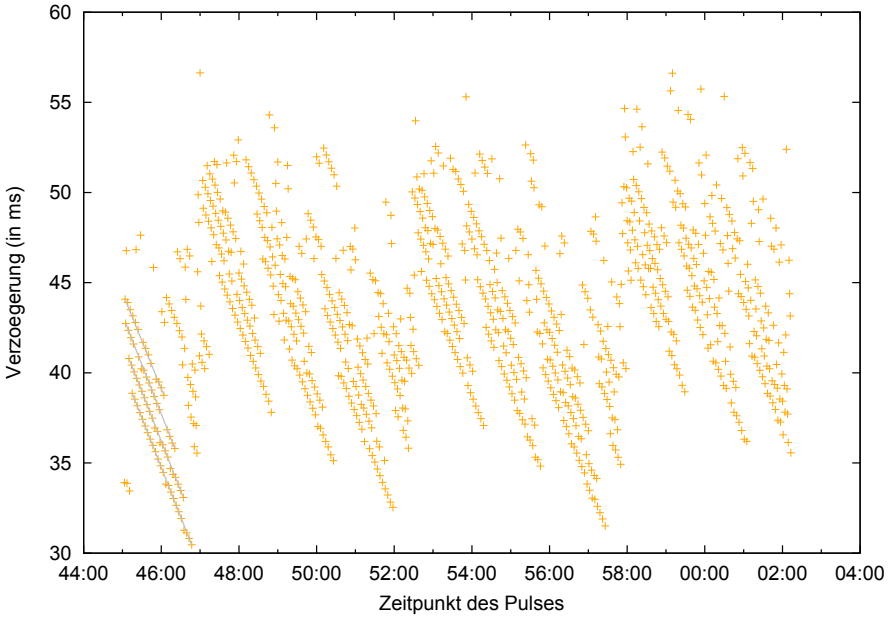

Abbildung 4.4: PPS-Verzögerungen in Messung 4 dargestellt als Punktewolke

Das Verteilungsdiagramm in Abbildung 4.5 zeigt schließlich, dass durch die Synchronisation der Threads die durchschnittliche Verzögerung um etwa 3ms abgesenkt und die Streuung leicht vermindert werden konnte. In Messung 4 lagen fast 90% aller Werte in dem 15ms-Intervall zwischen 36ms und 51ms.

Während beiden Zeitmessungen blieb die Abweichung zwischen der Systemzeit und dem NTP-Server auf dem Raspberry Pi deutlich unter 0,15ms. Folglich spielt diese für die Messergebnisse keine Rolle. Der Verlauf der Abweichung ist für Messung 3 bzw. 4 in Abbildung 4.6 links bzw. rechts dargestellt. Die Werte wurden durch regelmäßige Abfragen mit dem Programm *ntpq* bestimmt.

Eine mögliche Ursache für die Schwankungen der Latenz sind von PortAudio erzeugte Verzögerungen. In der Dokumentation der PortAudio-Bibliothek wird erwähnt [pao], dass das Festlegen des Parameters *framesPerBuffer* bei manchen Audioschnittstellen zusätzliche Latenzen erzeugen kann. In toa_send und toa_rcv ist dieser Wert, der der Länge des Abtastintervalls entspricht, festgelegt, da er mit der Eingabegröße der FFT korrespondiert. Wird diese Zahl nicht festgelegt, wird automatisch ein (teilweise wechselnder) optimaler Wert gewählt. Das ist zwar schwieriger zu handhaben, könnte aber die Erken-

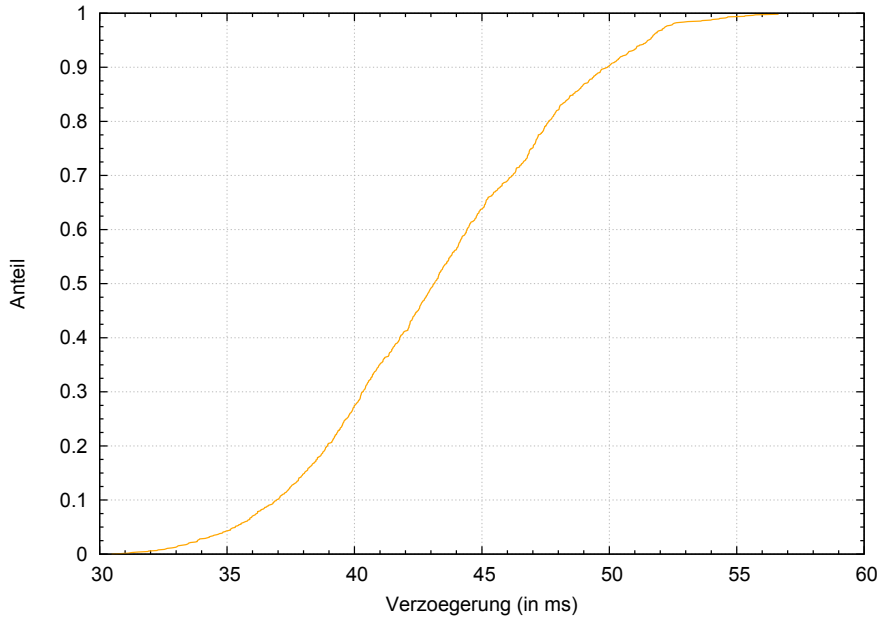

Abbildung 4.5: Verteilung der PPS-Verzögerungen in Messung 4

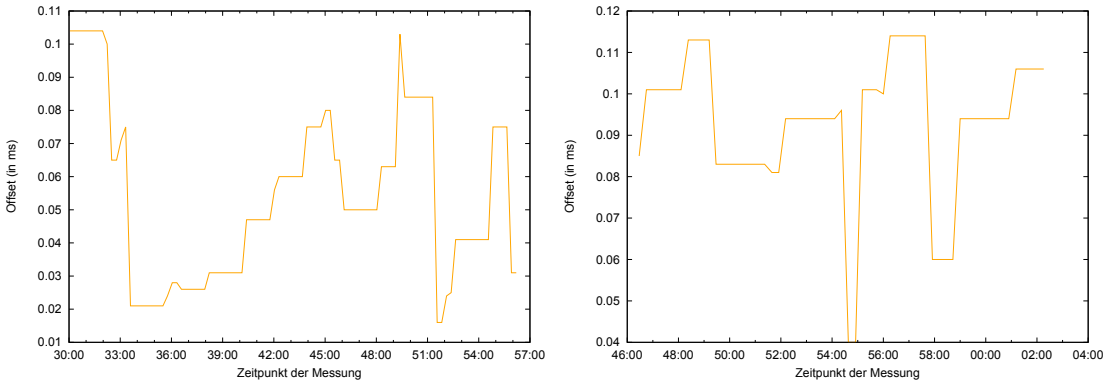

Abbildung 4.6: Verlauf des Offsets während Messung 3 und 4

nungslatenzen vermutlich deutlich senken (vor allem auch in Verbindung mit kleineren FFT-Arrays, siehe Kapitel 5). Das Implementieren und Testen dieser Möglichkeit konnte aus Zeitmangel leider nicht mehr umgesetzt werden.

Kapitel 5

Fazit

In dieser Arbeit wurde prototypisch eine Lösung zur Übertragung des PPS-Signals und der RMC-Datensätze eines GPS-Empfängers über Audio entwickelt. Damit wurde es ermöglicht zu überprüfen, ob eine Zeitsynchronisation für Latenzmessungen im Mobilfunknetz möglich ist. Zunächst wurde dafür eine Hardwareplattform ausgewählt, wobei die Wahl auf den Raspberry Pi fiel. Der Grund dafür war, dass dieser relativ hohe Leistungsreserven bietet. Anschließend wurde für den Raspberry Pi das Programm toa_send entwickelt, das die Codierung von PPS und RMC in Audio und die Ausgabe des entstehenden Audiostroms umsetzt. Entsprechend wurde für die Gegenseite, die auf einem per Audiokabel mit dem Raspberry Pi verbundenen PC läuft, das Programm toa_rcv geschrieben. Dieses übernimmt das Einlesen des Audiostroms und analysiert mit Hilfe der schnellen Fouriertransformation die Audiodaten um darin enthaltene Daten zu decodieren.
Damit konnte gezeigt werden, dass Übertragung und Erkennung von PPS und RMC über Audio sehr gut machbar ist. Die Erkennungsrate für den RMC lag in Messungen über 99% und für den PPS sogar bei 100%. Die Erkennungslatenz für den PPS schwankte in den Messungen allerdings zu stark, als dass sich ein, mit der aktuellen Version der Programme übertragener, Puls als Grundlage für eine Zeitsynchronisation für Latenzmessungen eignen würde. Eine mögliche Ursache für die Schwankungen wurde aber schon gefunden. Für eine Überprüfung fehlte leider die Zeit.

Ausblick

Während dieser Arbeit kamen einige Ideen auf, wie sich die entwickelten Programme möglicherweise weiter verbessern lassen könnten. Diese Ideen sollen hier noch kurz vorgestellt werden.

Für den PPS gibt es im Linuxkernel spezielle Einstellungen, die in dieser Arbeit nicht genutzt wurden. Möglicherweise könnte damit die Latenz und vor allem die Schwankung derselben für den Interrupt in toa_send verringert werden.
Außerdem könnte auf Empfängerseite eine Abtastintervalllänge gewählt werden, die ein Teiler der Abtastfrequenz ist, sodass die PPS-Schwingung theoretisch immer an der gleichen Stelle in einem Abtastintervall beginnt. Dies würde wahrscheinlich die Schwankung der Erkennungslatenz verringern. Allerdings könnte dann die FFT nicht mehr optimal arbeiten, was zu einer - vermutlich relativ geringen - Vergrößerung der Latenz führen würde.
Die Latenz, die nach der Aufzeichnung des Audiostroms durch die Analsyse entsteht, lässt sich herausrechnen. PortAudio stellt der Callback-Funktion den Zeitpunkt der Aufnahme des ersten Samples eines aktuellen Inputbuffers zur Verfügung. Wenn man diesen Zeitpunkt als Erkennungszeitpunkt nutzen würde, würde das die Erkennungslatenz wahrscheinlich deutlich vermindern.
Desweiteren könnte man zur PPS-Erkennung kleinere Intervalle für die FFT wählen, sodass diese früher und schneller berechnet werden könnte. Ein Idee wäre zum Beispiel, die dem PPS zugeordnete Frequenz so zu wählen, dass sie in der aktuell verwendeten FFT einen Ausschlag in Frequenzbin 96 erzeugt. Dann würden theoretisch schon 8 Werte für die FFT reichen um die Frequenz zu erkennen. In diesem Fall wäre der Ausschlag genau in Bin $\frac{96}{256} \cdot 8 = 3$.
Alternativ könnte man statt der Fouriertransformation auch den Goertzel-Algorithmus für die PPS-Erkennung verwenden. Dieser Algorithmus arbeitet schneller als die FFT, solange nur maximal 15% der Frequenzbins errechnet werden müssen. Für die RMC-Erkennung wäre er deshalb nicht von Interesse. Er könnte aber in Verbindung mit den oben angesprochenen kürzeren Abtastintervallen vermutlich einen deutlichen Geschwindigkeitsschub bringen.

Literaturverzeichnis

[ard] *Arduino Website*. http://arduino.cc/en/Guide/Introduction, Abruf: 28.10.2014.

[aud] *Pressemitteilung von Audi zum fahrerlosen Rennwagen*. https://www.audi-mediaservices.com/publish/ms/content/de/public/pressemitteilungen/2014/10/19/audi_rs_7_concept.html, Abruf: 23.10.2014.

[blo] *Blockierende Lese- und Schreibfunktionen in PortAudio*. http://portaudio.com/docs/v19-doxydocs/blocking_read_write.html, Abruf: 26.10.2014.

[fft] *FFTSS*. http://www.ssisc.org/fftss/, Abruf: 21.10.2014.

[FGJ] FRIGO, Matteo; G. JOHNSON, Steven: *FFTW3 Manual*. http://www.fftw.org/fftw3.pdf, Abruf: 20.08.2014.

[fmo] *FMOD Website*. http://www.fmod.org/, Abruf: 28.10.2014.

[gar] *GPS 18x Technical Specifications*. http://static.garmincdn.com/pumac/GPS_18x_Tech_Specs.pdf, Abruf: 30.10.2014.

[Gur14] GURSKI, Frank: *Effiziente Algorithmen*. http://www.acs.uni-duesseldorf.de/~gurski/ealgo, SS2014. Vorlesungsskript

[Hof11] HOFFMANN, R.: *Grundlagen der Frequenzanalyse*. Dritte Auflage. Expert-Verlag, 2011 (Kontakt & Studium). ISBN 9783816931089

[kis] *Kiss FFT*. http://sourceforge.net/projects/kissfft/, Abruf: 21.10.2014.

[ntp] *NTP Website*. http://ntp.org/, Abruf: 2.11.2014.

[Nuk06] NUKADA, Akira: *FFTSS: A high performance fast fourier transform library*. www.ssisc.org/fftss/ICASSP2006.pdf. Version: 2006, Abruf: 21.10.2014.

[OS92] OPPENHEIM, Alan V.; SCHAFER, Ronald W.: *Zeitdiskete Signalverarbeitung*. Erste Auflage. Oldenburg, 1992 (Grundlagen der Schaltungstechnik). ISBN 3486215442

[pao] *PortAudio Dokumentation mit Hinweis zur framesPerBuffer-Einstellung*. http://www.portaudio.com/docs/v19-doxydocs/portaudio_8h.html#a443ad16338191af364e3be988014cbbe, Abruf: 2.11.2014.

[ras] *Raspberry Pi Website*. http://www.raspberrypi.org/, Abruf: 28.10.2014.

[rpi] *rpi_gpio_ntp Website*. http://vanheusden.com/time/rpi_gpio_ntp/, Abruf: 2.11.2014.

[sat] *Satsignal Website*. http://www.satsignal.eu/ntp/Raspberry-Pi-NTP.html#no-soldering, Abruf: 28.10.2014.

[spe] *Speed Tests on Pentium 4*. http://www.fftw.org/speed/Pentium4-2.4GHz-gcc/, Abruf: 21.10.2014.

[ubla] *u-blox 7 receiver description and protocol specification v14*. http://www.u-blox.de/images/downloads/Product_Docs/u-blox7-V14_ReceiverDescriptionProtocolSpec_Public_%28GPS.G7-SW-12001%29.pdf, Abruf: 18.09.2014.

[ublb] *u-blox Max-7 Datasheet*. `http://www.u-blox.com/images/downloads/Product_Docs/MAX-7_DataSheet_(GPS.G7-HW-12012).pdf`, Abruf: 29.10.2014.

[wPi] *WiringPi Website*. `http://wiringpi.com/`, Abruf: 28.10.2014.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 3.November 2014 Felix Hein

Diese CD enthält:

- eine *pdf*-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Bachelorarbeit erstellten Programme toa_send und toa_rcv, sowie calibrate_send und calibrate_rcv
- die zur Evaluation verwendeten Datensätze messung1.txt bis messung4.txt
- die Websites der verwendeten Internetquellen