Entwicklung einer Softwarelösung zur hochgenauen Zeit- und Positionsbestimmung auf GPS-Basis für Car2X-Szenarien

Masterarbeit

im Studiengang Informatik

Benedikt Kleinmeier

Entwicklung einer Softwarelösung zur hochgenauen Zeit- und Positionsbestimmung auf GPS-Basis für Car2X-Szenarien

Masterarbeit

im Studiengang Informatik

Vorgelegt von:	Benedikt Kleinmeier
Fakultät:	Informatik und Mathematik
Matrikelnummer:	03 05 98 08
Erstprüfer:	Prof. Dr.-Ing. Lars Wischhof
Zweitprüfer:	Prof. Dr. Max Fischer

17. Dezember 2014

Development of a Software Solution for High-Precision GPS-based Time and Position Data in Car2X Scenarios

Master's Thesis

in Computer Science

Author:	Benedikt Kleinmeier
Department:	Department of Computer Science and Mathematics
Matriculation Number:	03 05 98 08
First Examiner:	Prof. Dr.-Ing. Lars Wischhof
Second Examiner:	Prof. Dr. Max Fischer

December 17, 2014

Zusammenfassung

Das Institut für Eingebettete Systeme und Kommunikationstechnik (ESK) der Fraunhofer-Gesellschaft forscht im Geschäftsfeld Automotive am Thema „Car2X-Kommunikation“. Auf Basis dieser Forschung wird ein Software-Framework entwickelt, um Anwendungsfälle abzudecken, die durch Car2X-Kommunikation einen intelligenten Verkehrsfluss ermöglichen. Ein Großteil der Anwendungsfälle verfolgt ein Ziel: Die Sicherheit im Straßenverkehr zu erhöhen, zum Beispiel indem vor möglichen Kollisionen gewarnt wird. Das Framework ist hierbei auf hochgenaue Zeit- und Positionsdaten angewiesen. Das ESK hat bereits ein eingebettetes System mit GPS-Empfänger im Einsatz, welches mit einem Linux-Betriebssystem betrieben wird. Dieses System bildet die Grundlage, um im Rahmen dieser Arbeit Zeit- und Positionsdaten zu akquirieren.

Die Arbeit beschäftigt sich zu Beginn mit technischen Grundlagen, darunter Satellitennavigation, GPS und GPS-Empfängern. Es wird deren Funktionsweise erläutert sowie die erreichbare Genauigkeit beleuchtet. Des Weiteren wird eine Technik vorgestellt, wie mit Hilfe eines Interrupts durch den GPS-Empfänger Verzögerungen bei der Verarbeitung von GPS-Daten ermittelt werden können, um die Genauigkeit der Zeitinformationen zu erhöhen. Im weiteren Verlauf werden Grundlagen erläutert, die von einer Softwarelösung berücksichtigt werden müssen, um eine größtmögliche Genauigkeit bei Zeit- und Positionsinformationen sicherzustellen. So werden unter anderem die Auswirkungen von verschiedenen Scheduling-Strategien auf das Zeitverhalten unter Linux betrachtet.

Nach den technischen Grundlagen wird das eingebettete System vorgestellt das beim ESK im Einsatz ist. Es werden Hard- und Softwarekomponenten betrachtet.

Im Implementierungsteil werden zwei Softwarelösungen vorgestellt. Eine findet sich auf Anwendungsebene wider und baut auf den Linux Daemon `gpsd` auf. Eine zweite alternative Lösung ist auf Systemebene als Linux-Treiber implementiert. In einem weiteren Schritt wird das Zeitverhalten des Linux-Treibers detaillierter untersucht, indem der Treiber mit einem echtzeitfähigen Scheduling-Verfahren betrieben wird.

Themengebiete: Zeit- und Positionsbestimmung, GPS, Linux, Daemon, Kernel, USB, Treiber

Abstract

The Fraunhofer Institute for Embedded Systems and Communication Technologies is doing research in the field of Car2X communication in its automotive unit. Based on this research, the institute develops a software framework for intelligent transport systems with the aid of Car2X communication. Most of the scenarios handled by the framework should improve the road safety, for instance to warn of possible collisions on junctions. For this reason, the framework needs high-precision time and position data. The Fraunhofer Insitute already uses an embedded system running a Linux operating system. Also a GPS receiver is connected to the embedded system. Based on this system, the task is to acquire high-precision time and position data.

First, this thesis provides some basic information, for instance about satellite navigation, GPS and GPS receivers, how they work, their limitations and their precision. Furthermore, the thesis introduces a technique called "Pulse per Second" allowing to meassure processing time of GPS data. This time can be used to correct the time information acquired by GPS to ensure best accuracy. The thesis also discusses software aspects which are necessary to ensure high-precision time and position data. For instance, the effects of different scheduling strategies in Linux on response time behavior are analyzed.

Beside basic information, the thesis sheds light on the embedded system which is used at Fraunhofer Insitute and provides information about its hardware and software components.

Next, the thesis focuses on the implementation of two software solutions. The first solution is based on the Linux daemon `gpsd` and runs in user space. The second and alternative solution is implemented in kernel space as Linux driver. In another step, the response time behavior of the Linux driver is optimized by changing the scheduling strategy of the driver.

Keywords: time determination and positioning, GPS, Linux, daemon, kernel, USB, driver

Vorwort

Die Masterarbeit wurde am Institut für Eingebettete Systeme und Kommunikationstechnik (ESK) der Fraunhofer-Gesellschaft verfasst. Warum? Das Informatik-Studium begann auf einem sehr hohen Abstraktionsniveau, zum Beispiel durch objektorientierte Programmierung an Hand von Java. Die eigentliche Hardware — das wohl grundlegendste Werkzeug der digitalen Informationsverarbeitung — wurde aber fast vollkommen ausgespart. Aus diesem Grund arbeitete ich rund 1,5 Jahre als Werkstudent bei der Infineon Technologies AG, um ein besseren Einblick in die Hardware zu bekommen. Dort beschäftigte ich mich überwiegend mit der Programmierung von Mikrocontrollern — also der reinen Hardware. Die Tätigkeit bei der Infineon Technologies AG gipfelte in der Bachelorarbeit mit dem Thema „Entwicklung eines Evaluierungswerkzeugs für barometrische Luftdruck-Sensoren“. Nun hatte ich zwei Extrema kennengelernt: reine Software und reine Hardware. Nun ist es Zeit sich die verbindende Schicht, das Betriebssystem genauer anzusehen. Grundlegende Aufgaben wie Prozess- und Speicherverwaltung sowie Ein-/Ausgabe wurden bereits im Bachelorstudium vermittelt. Konkrete Implementierungen wurden aber bisher nicht betrachtet.

Eine Stellenausschreibung des ESKs bot nun diese Möglichkeit. Das ESK möchte einen über USB angebundenen GPS-Empfänger in eine Linux-Umgebung integrieren und mögliche Latenzen bei der Verarbeitung von GPS-Daten ermitteln. Für die Integration ist unter anderem ein Linux-Kernel-Modul zu entwickeln, welches den GPS-Empfänger über USB anspricht. Diese Aufgabe macht es mir möglich den Linux-Kernel und dessen Funktionen genauer zu inspizieren. Ein Stück Software das mittlerweile auf Millionen verschiedenster Geräte verwendet wird. Von eingebetteten Systemen mit starker Ressourcenbeschränkung bis hin zu Highend-Server-Systemen — und alle nutzen die gleiche Codebasis. Ein Stück Ingenieurskunst. Ich möchte mich daher beim ESK für diese großartige Möglichkeit bedanken. Mein größter Dank gilt aber meinem guten Freund Norbert, der mich in der Zeit der Masterarbeit persönlich wie fachlich immer wieder unterstützte und auch nicht vor Kritik zurückschreckte. Ich danke auch Herrn Prof. Dr.-Ing. Lars Wischhof für die Betreuung dieser Arbeit.

Vielen Dank
Benedikt Kleinmeier

Inhaltsverzeichnis

Abkürzungen

AGPS Assisted GPS

DGPS Differential GPS

ESK Eingebettete Systeme und Kommunikationstechnik

ETSI European Telecommunications Standards Institute

GPS Global Positioning System

HSGPS High Sensivity GPS

IMU Inertial Measurement Unit

ITS Intelligent Transport System

NTP Network Time Protocol

PCI Peripheral Component Interconnect

PPS Pulse Per Second

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

Seit der Erfindung des Automobils im 19. Jahrhundert hat der Individualverkehr stetig zugenommen. Auf der einen Seite führte dies dazu das sich jeder Einzelne schnell und bequem fortbewegen kann. Die Kehrseite sind überfüllte Straßen und eine Vielzahl von Verkehrsunfällen. Technische Innovationen wie das Anti-Blockier-System oder das Elektronische Stabilitätsprogramm unterstützen den Fahrer und haben geholfen Verkehrsunfälle zu vermeiden. Diese Techniken zielen darauf ab den Fahrer in Gefahrensituationen zu unterstützen. Ein besserer Ansatz ist vor Gefahrensituationen zu warnen. Dazu ist es notwendig, dass die unmittelbare Umgebung des Autos bekannt ist, um zum Beispiel mögliche Kollisionsgefahren erkennen zu können. Eine Möglichkeit dies zu erreichen, ist es Autos untereinander zu vernetzen, sodass Informationen ausgetauscht werden können. Diese Informationen können sehr vielfältig sein: Zum Beispiel Positionsdaten zur Vermeidung von Kollisionen oder auch Daten um bei fließendem Verkehr Mautgebühren abzurechnen.

Werden Kraftfahrzeuge miteinander vernetzt und tauschen Informationen aus, spricht man von Car2Car-Kommunikation. Sind bei dieser Kommunikation nicht nur Fahrzeuge beteiligt, sondern auch Infrastruktureinrichtungen, wie zum Beispiel Ampeln, so ist von Car2X-Kommunikation die Rede. Allgemein wird auch von Intelligent Transport System (ITS) gesprochen, wenn verkehrsbezogene Daten automatisiert ausgetauscht werden, um den Verkehrsfluss zu steuern. Die große Stärke dieser Kommunikation liegt darin, dass Gefahren erkannt werden noch bevor sie für den Fahrer wahrnehmbar sind. Folgendes Szenario ist zum Beispiel häufig auf Autobahnen anzutreffen: Die Sicht für einen Autofahrer wird durch einen vorausfahrenden LKW verdeckt. Durch ein plötzliches Bremsmanöver durch ein Auto vor dem LKW ergibt sich eine große Gefahr für einen Auffahrunfall. Abbildung 1.1 veranschaulicht dieses Szenario.

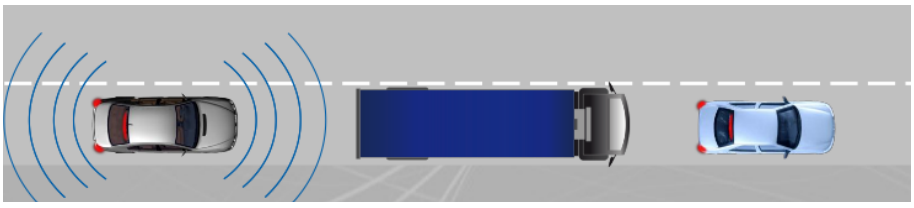


Abbildung 1.1.: Warnung vor Auffahrunfall auf Autobahn [1, S. 15]

Der folgende Abschnitt 1.1 zeigt auf, dass das Institut für Eingebettete Systeme und Kommunikationstechnik (ESK) der Fraunhofer-Gesellschaft im Themenfeld Car2X-Kommunikation forscht und ein Software-Framework implementiert, um intelligenten Verkehrsfluss zu ermöglichen. Das Software-Framework ist auf Zeit- und Positionsdaten angewiesen, um zum Beispiel Kollisionsvorhersagen zu treffen. Diese Daten hochgenau zu akquirieren ist Ziel der Masterarbeit und wird in Abschnitt 1.2 ausgeführt. Abschnitt 1.3 gibt einen Überblick über die Struktur der gesamten Ausarbeitung.

1.1. Car2X-Kommunikation

Das Institut für ESK der Fraunhofer-Gesellschaft forscht mit Partnern aus der Industrie am Thema Car2X-Kommunikation. Das ESK entwickelt hierbei ein Software-Framework, um Projektpartnern einen schnellen Einstieg in das Thema Car2X-Kommunikation zu ermöglichen. Es nutzt für die Implementierung des Frameworks Spezifikationen für ITS die vom European Telecommunications Standards Institute (ETSI) ausgearbeitet werden. Die Spezifikationen sind auf mehrere Dokumente aufgeteilt, wobei einige Teile erst als Entwurf vorliegen[1]. Ein Teil der Spezifikationen ist der Technical Report 102638 [8].

Der Technical Report beschreibt eine Architektur für ITS-Anwendungen. Die Architektur sieht drei Arten von Netzwerkteilnehmern vor, Kraftfahrzeuge, stationäre Kommunikationsknoten und eine zentrale Informationsstelle. Das Kraftfahrzeug ist Teil eines hochdynamischen Netzwerks und wird in der Architektur als „Vehicular Stations“ bezeichnet. Neben Fahrzeugen gibt es noch stationäre Kommunikationsknoten, welche am Straßenrand postiert sind. Sie werden „Roadside Stations“ genannt. Über diese Roadside Stations können Car2X-Nachrichten an eine zentrale Stelle geschickt werden bzw. Nachrichten von dort bezogen werden. Diese zentrale Stelle heißt „ITS Service Centre“. Das ITS Service Centre kann so ein Gesamtbild der Verkehrslage erstellen. Dadurch können an Autos zum Beispiel rechtzeitig Stauwarnungen übermittelt werden, sodass der Stau weiträumig umfahren werden kann. Abbildung 1.2 veranschaulicht diese Architektur.

[1] Eine vollständige Liste von ITS-Standards findet sich unter `http://www.etsi.org/index.php/technologies-clusters/technologies/intelligent-transport`.

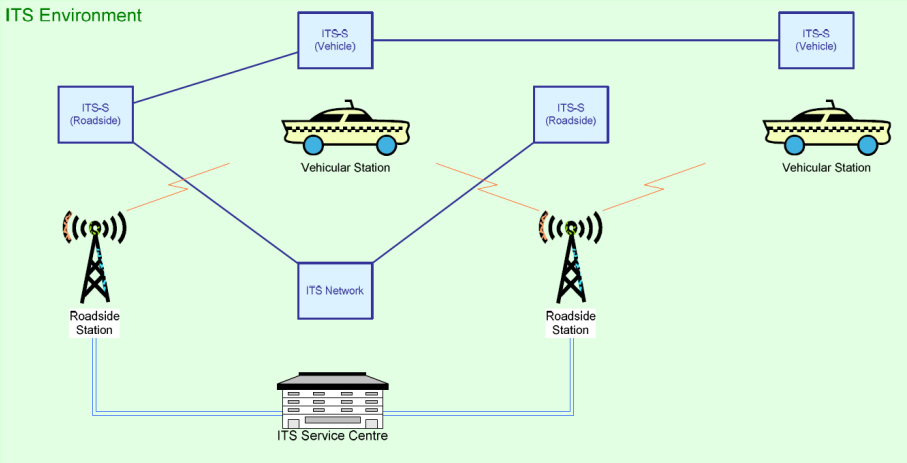


Abbildung 1.2.: Architektur für ITS-Anwendungen nach [8, S. 11]

Neben einer Architektur beschreibt [8] auch eine Menge von Anwendungsfällen, welche durch ein ITS-Framework abgedeckt werden sollen. Das ESK möchte mit seinem Framework im ersten Schritt unter anderem drei Anwendungsfälle abdecken. Das sind „Emergency Electronic Brake Lights“, „Intersection Collision Warning“, und „Co-Operative Forward Collision Warning“. Diese Anwendungsfälle werden im Folgenden kurz erläutert.

Beim Anwendungsfall Emergency Electronic Brake Lights sollen durch Car2X-Nachrichten nachfolgende Autos vor einem starken Bremsmanöver des eigenen Autos gewarnt werden. Beim Tritt auf die Bremse werden neben dem üblichen Bremslicht auch Nachrichten an nachfolgende Autos verschickt. Diese Nachrichten enthalten unter anderem die Uhrzeit und Position des sendenden Fahrzeugs. Fahrzeuge, welche diese Nachrichten empfangen, haben dann die Möglichkeit den Fahrer zu warnen. Entweder visuell, zum Beispiel durch eine Warnung in einem Head-Up-Display oder akustisch, zum Beispiel durch einen Warnton. Vorteile der Car2X-Nachrichten gegenüber dem normalen Bremslicht ergeben sich vor allem bei schlechten Sichtverhältnis. Zum Beispiel bei Nebel oder wenn sich zwischen bremsenden und eigenem Auto weitere Fahrzeuge befinden. Abbildung 1.3 illustriert diesen Anwendungsfall.

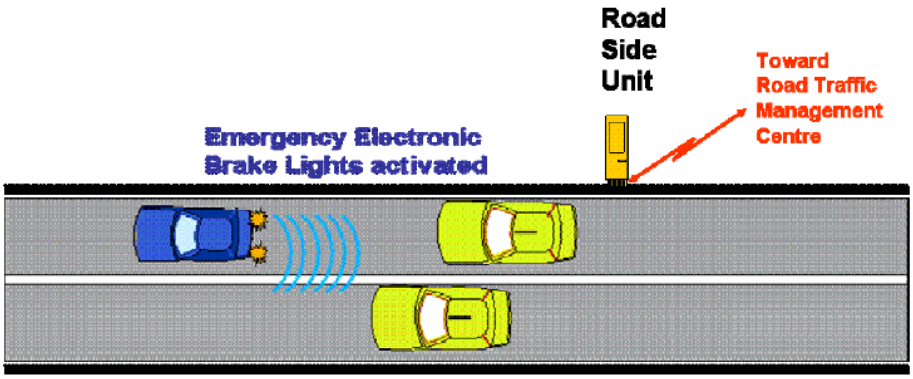


Abbildung 1.3.: Anwendungsfall Emergency Electronic Brake Lights [8, S. 26]

Im Anwendungsfall Intersection Collision Warning sollen Fahrzeuge im Bereich einer Kreuzung durch Car2X-Nachrichten vor Kollisionen gewarnt werden. Wartet ein Fahrzeug zum Beispiel an einem Stoppschild und möchte die Straße überqueren, so können Car2X-Nachrichten genutzt werden, um das wartende Fahrzeug vor herannahenden Autos zu warnen. Vergleiche hierzu Abbildung 1.4.

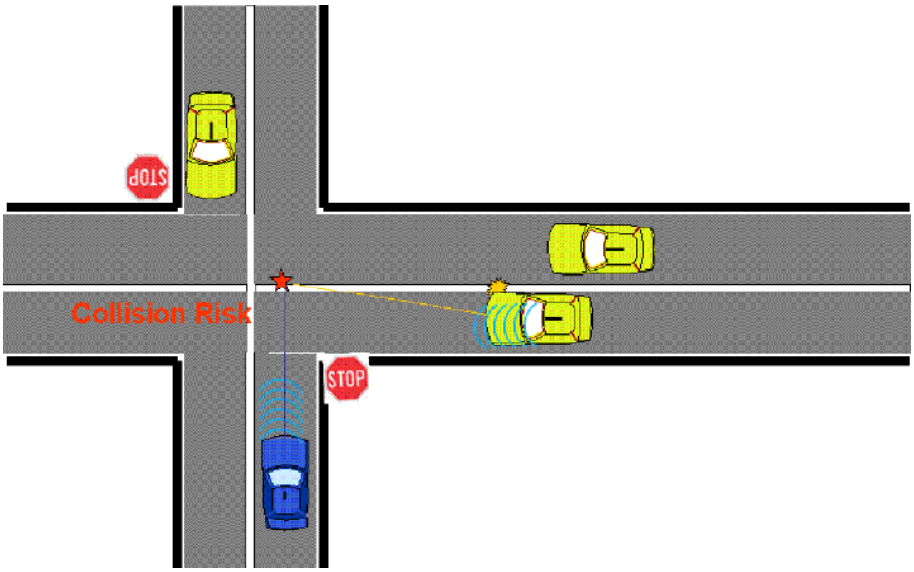


Abbildung 1.4.: Anwendungsfall Intersection Collision Warning [8, S. 46]

Der Anwendungsfall Co-Operative Forward Collision Warning beschreibt ein typisches Szenario auf einer Autobahn: Mehrere Autos fahren dicht hintereinander in einer Kolonne. Durch Austausch von Nachrichten soll erreicht werden, dass der Abstand zwischen den Autos immer ausreichend groß ist und es nicht zu Auffahrunfällen kommt. Im Gegensatz zu den beiden ersten Anwendungsfällen ist bei Co-Operative Forward Collision Warning angedacht nicht nur den Fahrer zu warnen,

sondern direkt das Auto automatisch abzubremsen. Siehe für diesen Anwendungsfall Abbildung 1.5

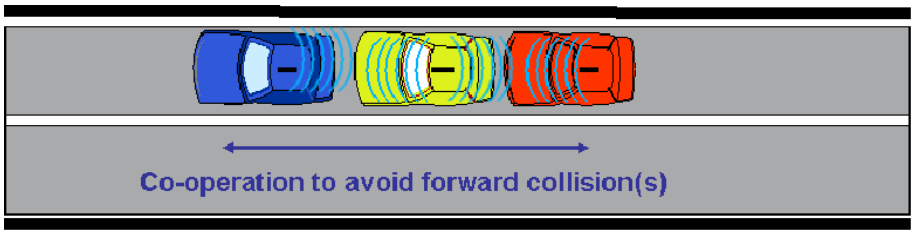


Abbildung 1.5.: Anwendungsfall Co-Operative Forward Collision Warning [8, S. 47]

1.2. Ziele der Masterarbeit

All die unter Abschnitt 1.1 genannten Anwendungsfälle verlangen die Lokalisation des eigenen Autos aber auch der umgebenden. Für die Lokalisation wird primär das Global Positioning System (GPS) verwendet. Um eine genaue Positionsvorhersage für ein fahrendes Fahrzeug machen zu können, ist es notwendig, dass die GPS-Daten möglichst verzögerungsfrei verarbeitet werden.

Wird beispielsweise der Anwendungsfall Intersection Collision Warning aus Abbildung 1.4 genauer betrachtet, so gibt es dort ein wartendes Auto an einer Kreuzung (blau) und ein fahrendes Fahrzeug (gelb). Hierbei können folgende Verzögerungen die Lokalisation beeinträchtigen, t_{GPS}, t_{send} und t_{process}. t_{GPS} beschreibt die Zeit vom Eintreffen der GPS-Daten an der Antenne des fahrenden Autos bis zur Verarbeitung der Daten durch eine CPU. Des Weiteren tritt zwischen Empfang und Senden der Car2X-Nachrichten die Verzögerung t_{send} auf. Am wartenden Auto muss die empfangene Car2X-Nachricht auch noch verarbeitet werden bis tatsächlich eine Aktion ausgeführt werden kann. Dabei tritt die Verzögerung t_{process} auf. Nähert sich zum Beispiel das fahrende Fahrzeug mit einer Geschwindigkeit von 100 km/h, so würde eine Verzögerung von $t_{\text{GPS}} = 100$ ms dazu führen, dass eine Position geschickt wird, welche $2,8$ m von der tatsächlichen Position abweicht ($s = v \cdot t = 100$ km/h $\cdot\ 100$ ms $= 0,028$ m/ms $\cdot\ 100$ ms $= 2,8$ m).

Das ESK hat bereits in Fahrzeugen von Projektpartnern ein eingebettetes System im Einsatz, um weniger zeit- und positionskritische Anwendungen abzuwickeln. Das eingebettete System wird ARTiS PC genannt und verfügt über einen GPS-Empfänger. Auf dem eingebetteten System wird ein Linux-basiertes Betriebssystem eingesetzt. Im Rahmen der Masterarbeit soll eine Softwarelösung entwickelt werden, die hochgenaue Zeit- und Positionsdaten liefert.

1.3. Struktur der Arbeit

Die Struktur dieser Arbeit gliedert sich wie folgt: Von Seite 23 bis 49 werden theoretische Grundlagen erläutert. Unter anderem wird der Stand der Technik zur Zeit- und Positionsbestimmung im Automobilbereich vorgestellt. Des Weiteren wird auf Satellitennavigation und das GPS näher eingegangen, welches als Basistechnologie für die Implementierung dient. Zusätzlich werden noch Grundlagen vermittelt, die wichtig sind, um den Implementierungsteil bewerten zu können, darunter Aspekte zur Zeitmessung in Software und Auswirkungen von Scheduling-Strategien auf das Zeitverhalten. Die Seiten 49 bis 52 beschreiben das eingebettete System, das beim ESK im Einsatz ist und für die Implementierung genutzt wird. Auf den Seiten 53 bis 89 wird die Implementierung beschrieben, wie Zeit- und Positionsinformationen über GPS möglichst verzögerungsfrei bezogen werden. Es werden zwei Lösungsansätze vorgestellt. Eine auf Anwendungsebene auf Basis des Linux-Daemons `gpsd` und eine weitere auf Betriebssystemebene durch die Neuentwicklung eines Linux-Kernel-Moduls. Die Seiten 90 bis 92 fassen die Ergebnisse der Arbeit zusammen und geben einen Ausblick über zukünftige Möglichkeiten.

2. Grundlagen

Für die Implementierung wird ein bestehendes eingebettetes System genutzt. Um dessen Leistungsfähigkeit und Einschränkungen einschätzen zu können, ist es wichtig die zu Grunde liegenden Techniken zu verstehen. Dieses Kapitel zielt darauf ab diese Grundlagen zu vermitteln. Zu Beginn in Abschnitt 2.1 wird ein genereller Überblick gegeben, wie Autohersteller planen Car2X-Kommunikation in Serienfahrzeugen einzusetzen. Das vorliegende eingebettete System ARTiS PC nutzt zur Zeit- und Positionsbestimmung GPS. Abschnitt 2.2 geht daher näher auf Satellitennavigation und GPS und deren Limitierungen ein. Der Abschnitt 2.3 stellt eine Möglichkeit vor die Zeitgenauigkeit von GPS-Empfängern durch Auslösen eines Interrupts zu erhöhen. Die letzten vier Abschnitte vermitteln Grundlagen, welche für die Implementierung nützlich sind. Der Unterpunkt 2.4 zeigt welche Probleme sich durch Einsatz des Network Time Protocol (NTP) lösen lassen. Abschnitt 2.5 erläutert wie während der Implementierung die Zeitmessung durchgeführt wird und welche Genauigkeit hierbei zu erwarten ist. Das vorhandene eingebettete System und Betriebssystem ist nicht daraufhin optimiert echtzeitkritische Anwendungen zu bedienen. Aus diesem Grund müssen die Auswirkungen von Scheduling-Strategien auf das Zeitverhalten berücksichtigt werden. Abschnitt 2.6 widmet sich diesem Thema. Das Kapitel endet mit der Erläuterung zweier Software-Themen. Zuerst wird der Linux-Daemon `gpsd` vorgestellt. Mit diesem Daemon können GPS-Daten akquiriert werden. Der Daemon wird genutzt, um das bestehende System in Betrieb zu nehmen und bildet auch die Grundlage für einen ersten Implementierungsansatz. Um eine Verbesserung der Implementierung zu erreichen, wird ein eigener Treiber in Linux entwickelt, um GPS-Daten zu beziehen. Der letzte Abschnitt schafft daher ein grundsätzliches Verständnis für die Treiberentwicklung in Linux.

2.1. Stand der Technik

Nicht nur das ESK forscht am Thema Car2X-Kommunikation. Auch Firmen wie Audi, BMW, Daimler, Volkswagen und Bosch forschen in diesem Bereich. Diese und weitere Einrichtungen aus Industrie und Wissenschaft haben sich zusammengeschlossen und den bisher größten Feldtest zum Thema Car2X-Kommunikation in Deutschland im Jahr 2013 abgeschlossen. Hierfür wurden Konzepte und Techniken entwickelt, um verlässliche Car2X-Kommunikation zu ermöglichen. Diese Konzepte sollen auch zukünftig in Serienfahrzeugen verwendet werden. Die folgenden Absätze

umreißen wie nach heutigem Stand der Technik Car2X-Kommunikation betrieben wird. Vergleiche [16] und [22] für eine vollständige Systemübersicht.

2.1.1. Zeit- und Positionsbestimmung durch GPS

Ein wesentlicher Aspekt der Car2X-Kommunikation sind exakte Zeit- und Positionsdaten. Zudem muss die Zeit bei allen Teilnehmern synchron laufen, andernfalls ist eine korrekte Auswertung der Daten nicht möglich. Zeit- und Positionsinformationen werden über ein Satellitennavigationssystem bezogen. Auf Grund der weltweiten Verfügbarkeit wird dazu GPS verwendet. Jedes Fahrzeug ist mit einem GPS-Empfänger ausgestattet. Um die Positionsgenauigkeit bei GPS zu erhöhen werden Korrekturinformationen aus dem Differential GPS genutzt[1]. Die Zeitinformationen aus GPS lösen auch das Problem der Zeitsynchronität unter allen Teilnehmern. Alle Teilnehmer erhalten durch die Satellitensignale die selben Zeitinformationen.

2.1.2. Fusionierung von GPS- und Fahrzeugdaten

Durch den Einsatz von GPS ergeben sich zwei Probleme. Ein GPS-Signal ist nicht überall verfügbar. Für einen guten Empfang braucht es eine Sichtverbindung zum Himmel, welche in Tunneln und tiefen Hochhausschluchten nicht oder nur unzureichend gegeben ist. Zudem wird das GPS-Signal nur in einem diskreten Intervall vom GPS-Empfänger ausgewertet. Übliche Auswertungsintervalle liegen zwischen 1 und 10 Hz. Zwischen den Auswertungen sind keine Positionsinformationen bekannt. Aus diesem Grund werden die GPS-Daten mit Sensordaten des eigenen Fahrzeugs fusioniert. Ein modernes Fahrzeug verfügt über eine Vielzahl von Sensoren. Mit Hilfe der aktuellen Geschwindigkeit des Fahrzeugs und des Lenkradwinkels kann, ausgehend von der letzten bekannten Position laut GPS, der aktuelle Standort des Fahrzeugs bestimmt werden. Diese „Kopplung “ von verschiedenen Daten ist auch als „Dead Reckoning“ bekannt.

2.1.3. Car2X-Kommunikation durch WLAN und Mobilfunk

Die Position des eigenen Fahrzeugs wird bei der Car2X-Kommunikation kontinuierlich an andere Fahrzeuge geschickt. Diese können dann entscheiden, ob es sich um eine Gefahrensituation handelt und der Fahrer gewarnt werden muss oder nicht. Für die Übertragung der Nachrichten auf physikalischer Ebene wird WLAN oder Mobilfunk verwendet. Für WLAN sind die Standards 802.11b/g/p vorgesehen. Der Standard 802.11p ist eine Erweiterung von 802.11, die speziell auf die Bedürfnisse in Personenkraftwagen zugeschnitten ist. Auf Grund der Dynamik von Fahrzeugen

[1] Eine ausführliche Diskussion zu Satellitennavigation und GPS findet sich in Abschnitt 2.2 ab Seite 25.

in Car2X-Szenarien wird das WLAN nicht im Infrastruktur- sondern im Ad-hoc-Modus betrieben. Das heißt eine zentrale Basisstation über welche die Nachrichten gesendet werden fehlt. Stattdessen werden die Nachrichten direkt von Fahrzeug zu Fahrzeug weitergereicht. Um größere Distanzen als mit WLAN überbrücken zu können, werden für die Datenübertragung die Mobilfunkstandards GPRS und UMTS verwendet. Zur Reduzierung der Komplexität wird Netzwerkverkehr üblicherweise nach dem OSI-Modell abgearbeitet. Bei der Car2X-Kommunikation sind Schicht 1 und 2 durch WLAN und Mobilfunk abgedeckt. Für die Vermittlungsschicht ist das Internet Protocol (IP) vorgesehen. Auf der Transportschicht wird TCP und UDP verwendet.

2.2. Satellitennavigation und GPS

Mit Hilfe der Satellitennavigation lassen sich weltweit Zeit- und Positionsbestimmungen durchführen. Das Militär sah dafür bereits in den 1970er Jahren Bedarf und begann mit der Forschung in diesem Bereich. Ein großes Ziel war eine einfachere und genauere Navigation der eigenen Truppen, als dies mit Kompass und Karte möglich ist. Heute hat sich die Satellitennavigation auch im zivilen Bereich etabliert. Als Beispiele sind Navigationsgeräte in Fahrzeugen oder GPS-Empfänger in Smartphones zu nennen.

Die Satellitennavigation beruht auf der Laufzeitmessung von Signalen. Als anschauliches Beispiel dient die Entfernungsbestimmung zu einem Gewitter durch einen Menschen. Der Mensch ist in der Lage ohne technische Hilfsmittel die Entfernung zu einen Gewitter abzuschätzen. Er zählt die Zeit, die zwischen Blitz und Donner liegt. Dazu ermittelt das Auge den Startzeitpunkt t_s. Nun wird Laufzeit des Donners — der sich mit Schallgeschwindigkeit c_{Luft} ausbreitet — gemessen. Dazu wird gewartet bis die Schallwellen des Donners das menschliche Ohr erreichen. Dies markiert den Endzeitpunkt t_e. Durch Laufzeitmessung des „Signals“ (Schall) und Wissen über dessen Geschwindigkeit lässt sich die Entfernung zum Gewitter bestimmen, vergleiche Gleichungen 2.1 und 2.2.

$$\text{Laufzeit} = t_e - t_s \tag{2.1}$$

$$\text{Entfernung} = \text{Laufzeit} \cdot c_{\text{Luft}} \tag{2.2}$$

Das Prinzip bei der Satellitennavigation ist ähnlich. Hierbei senden Satelliten kontinuierlich Nachrichten zur Erde. Diese Nachrichten enthalten die Position des Satelliten und die exakte Zeit laut Atomuhr. Diese Zeit entspricht dem Startzeitpunkt t_s aus dem vorherigen Beispiel. Die Nachrichten werden als elektromagnetische Signale übermittelt und breiten sich mit Lichtgeschwindigkeit c aus. Ein elektronischer

Empfänger erhält nun diese Nachrichten und kann mit Hilfe des Empfangszeitpunkts t_e und den zwei Größen t_s und c die Entfernung zu einem einzelnen Satelliten bestimmen.

Ziel ist es aber die Lage im dreidimensionalem Raum zu bestimmen. Dieses Problem lässt sich verallgemeinern auf die Frage wie viele Satelliten in n Dimensionen zur Positionsbestimmung benötigt werden? Diese Verallgemeinerung erlaubt es das Problem herunter zu skalieren und einfache Fälle in kleinen Dimensionen zu betrachten. Mathematisch lässt sich der zweidimensionale Fall durch Abbildung 2.1 darstellen.

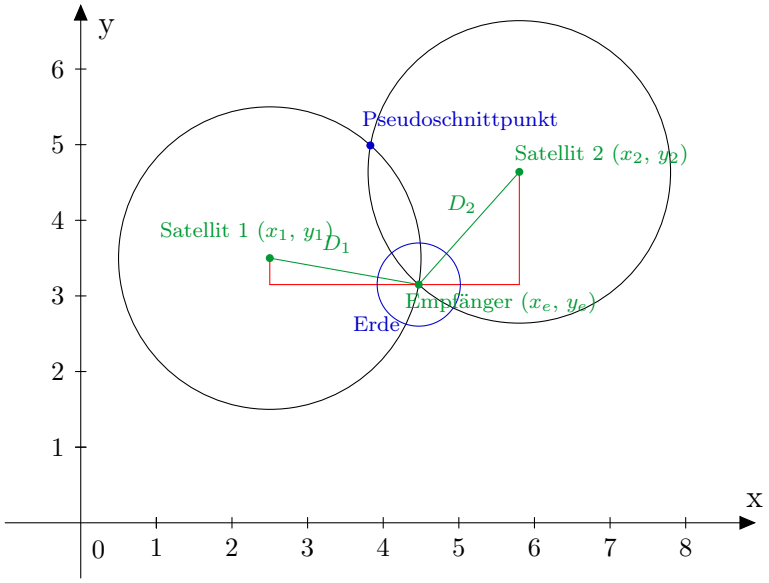


Abbildung 2.1.: Positionsbestimmung durch Signale von zwei Satelliten

Aus den Entfernungen zu den beiden Satelliten lassen sich zwei Kreise konstruieren. Diese beiden Kreise schneiden sich. Mathematisch lässt sich das Problem durch zwei Gleichungen formulieren, siehe 2.3 .

$$\begin{aligned}(x_e - x_1)^2 + (y_e - y_1)^2 &= D_1^2 \\ (x_e - x_2)^2 + (y_e - y_2)^2 &= D_2^2\end{aligned} \tag{2.3}$$

Insgesamt ergeben sich zwei Schnittpunkte. Im zweidimensionalen Fall kann durch Prüfung der y-Koordinate der zweite Schnittpunkt („Pseudoschnittpunkt“ genannt in Abbildung 2.1) verworfen werden. Die Distanz D zu einem Satelliten wird ermittelt durch

$$D = t_{\text{Empfänger}} - t_{\text{Sender}} \cdot c = \Delta t \cdot c \tag{2.4}$$

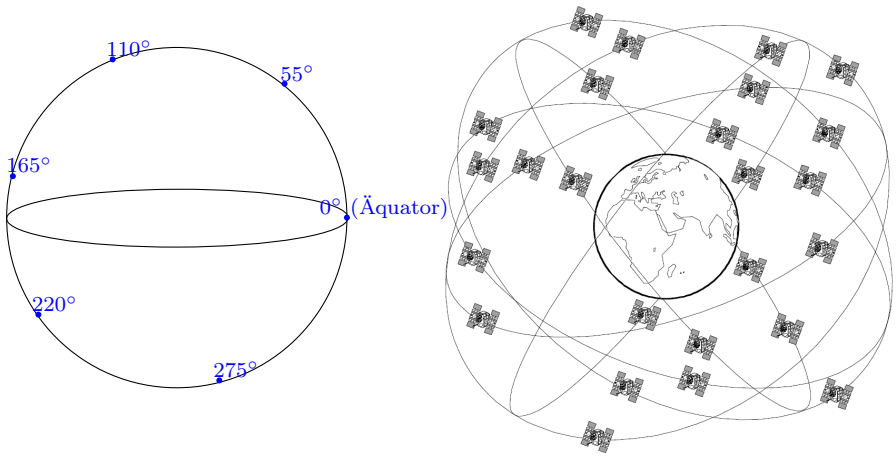
Die Satelliten senden die exakte Zeit t_{Sender} laut Atomuhr zur Erde. Damit $t_{\text{Empfänger}}$ auch wirklich exakt wäre, müsste der Empfänger ebenfalls eine Atomuhr mitführen. Da es sich in der Regel um einen kleinen, mobilen Empfänger handelt, ist dies nicht möglich. Dadurch ergibt sich bei der Auswertung der einzelnen Satellitensignale ein konstanter Fehler e. Dieser Fehler würde sich auf die Positionsgenauigkeit auswirken. e lässt sich durch die Hinzunahme eines weiteren Satelliten und damit einer weiteren Gleichung bestimmen. Für die Satellitennavigation im dreidimensionalem Raum sind deshalb vier Satelliten erforderlich, um die vier unbekannten Größen x, y, z (Position des Empfängers) und e (Zeitfehler) zu ermitteln, vergleiche Gleichung 2.5.

$$(x - x_n)^2 + (y - y_n)^2 + (z - z_n)^2 = [(\Delta t_n + e) \cdot c]^2 \text{ mit n} = [1, 4] \tag{2.5}$$

2.2.1. GPS

Um Satellitennavigation zu ermöglichen braucht es Satelliten. Zur Zeit stehen fünf Satellitensysteme zur Verfügung, welche von unterschiedlichen Nationen entwickelt wurden: GPS (USA), GLONASS (Russland), Galileo (Europa), QZSS (Japan) und Beidou (China).

GPS wurde unter dem Namen „Navigational Satellite Timing and Ranging Global Positioning System“ (NAVSTAR GPS) vom US-amerikanischen Verteidigungsministerium ab den 1970er Jahren entwickelt und ging 1995 offiziell in Betrieb. GPS ist im Jahr 2014 das einzige System, welches eine weltweite Abdeckung erreicht. Dies wird durch 31 Satelliten sichergestellt. Diese kreisen in rund 20.200 km Höhe über der Erde und benötigen für eine Erdumrundung 11 Stunden und 58 Minuten. Die Satelliten kreisen auf sechs verschiedenen Bahnen, die jeweils um 55° zum Äquator geneigt sind, siehe Abbildung 2.2. Dies stellt sicher, dass zu jedem Zeitpunkt ein Punkt auf der Erde durch mindestens vier Satelliten abgedeckt ist ([23, S. 14]).



(a) 2D (b) 3D [23, S. 14]

Abbildung 2.2.: GPS-Satellitenbahnen

Zur exakten Zeitbestimmung führt jeder Satellit bis zu vier Atomuhren mit. Anfangs wurden die Satellitensignale auf einer Frequenz von 1.575,42 MHz, genannt L1, zur Erde gesendet. Die Ionospähre bewirkt eine Verlangsamung der elektromagnetischen Wellen. Dadurch darf die Ausbreitungsgeschwindigkeit der Wellen nicht mehr als konstant angenommen werden. Um die Auswirkung der Ionospähere bestimmen zu können, wird seit 1998 eine zweite Frequenz verwendet. Diese Frequenz nutzt eine Wellenlänge von 1227,60 MHz und wird L2 genannt. Um die Genauigkeit bei der Positionsbestimmung zu steigern, muss ein GPS-Empfänger in der Lage sein beide Frequenzen zu verarbeiten.

Architektur

GPS wird in drei Segmente unterteilt, Weltraum-, Benutzer- und Kontrollsegment. Das Weltraumsegment umfasst die Satelliten in der Erdumlaufbahn. Das Benutzersegment beschreibt alle Empfangsgeräte auf der Erde. Ergänzt werden diese beiden Segmente durch das Kontrollsegment. Hierbei überwachen verschiedene Bodenstationen die Funktionen der Satelliten und schicken Korrekturinformationen an die Satelliten. Die Kontrollinformationen umfassen zum Beispiel die exakten Bahndaten eines Satelliten. Diese werden Ephemeriden genannt. Die genäherten Bahndaten hingegen werden als Almanachen bezeichnet. Jeder Satellit sendet seine Ephimeriden aus und zusätzlich die Almanachen aller Satelliten. Die Almanachen erlauben einem Empfangsgerät schnell zu ermitteln, welche Satelliten „sichtbar“ sind. Die Ephimeriden werden zur exakten Positionsbestimmung genutzt. Illustriert wird die Architektur durch Abbildung 2.3.

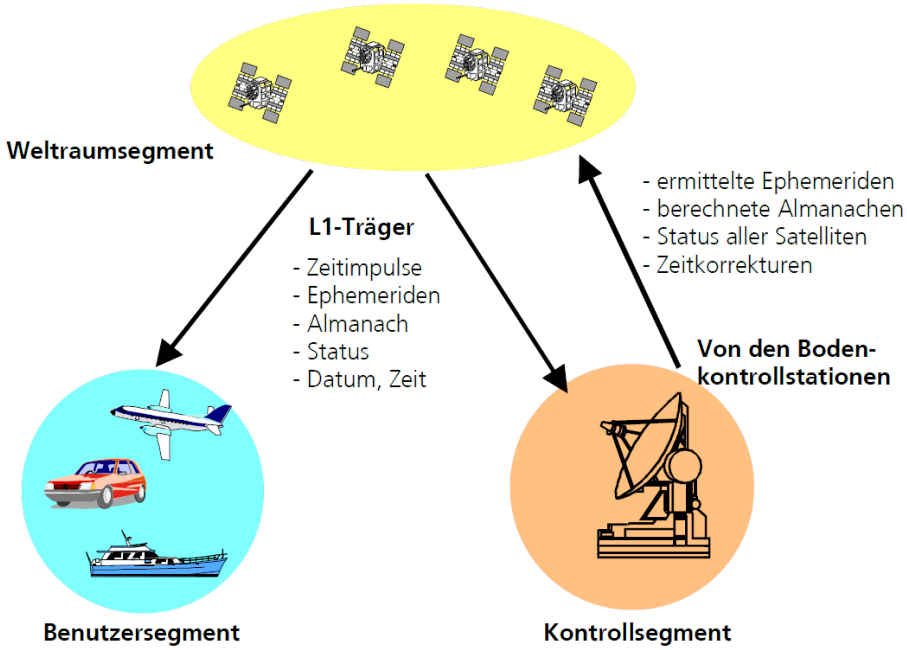


Abbildung 2.3.: GPS-Architektur [23, S. 45]

Genauigkeit

Je nach Leistungsfähigkeit des Empfängers variiert die Genauigkeit der ermittelten Position und Zeit. Tabelle 2.1 liefert einer Übersicht über die Genauigkeiten, welche mit GPS erreicht werden können. Die Angaben gelten für ein Konfidenzniveau von 95%.

Tabelle 2.1.: Genauigkeit von GPS (vergleiche [23, S. 81]

Horizontale Genauigkeit	**Vertikale Genauigkeit**	**Zeitgenauigkeit**
≤ 9 m	≤ 15 m	≤ 40 ns

Verschiedene Faktoren Tragen zur Ungenauigkeit bei GPS bei, so zum Beispiele ungenaue Ephimeriden eines Satelliten. Es gibt verschiedene Ansätze die Genauigkeit zu steigern. Drei Ansätze werden im Folgenden näher betrachtet.

Differential GPS (DGPS) nutzt Bodenstationen deren exakte Positionen bekannt sind. Diese Stationen vergleichen die Ist-Position laut Satellitensignalen mit der Soll-Position und errechnen dadurch Korrekturinformationen. Für die Übertragung der Korrekturinformationen wird oftmals der Standard RTCM SC-104[1] verwendet

[1] Abkürzung für Radio Technical Commission for Maritime Services Special Committee 104.

[23, S. 110]. Die RTCM-fähigen mobilen GPS-Empfänger können Korrekturdaten über Kurz-, Mittel- und Langwelle erhalten sowie über Mobilfunk und über zusätzliche Satelliten[1]. Durch DGPS lässt sich die Genauigkeit auf 3,6 m (Konfidenzniveau 95%) bzw. 1,8 m (68%) steigern [23, S. 118].

Assisted GPS (AGPS) ist eine weitere Möglichkeit um die Genauigkeit eine Empfängers zu steigern. Diese Technik ist für Innenbereiche gedacht und greift auf Daten aus Mobilfunk- und WLAN-Netzen zurück.

High Sensivity GPS (HSGPS) hat allgemein das Ziel die Leistung von GPS-Empfängern zu erhöhen. Hierbei wird versucht stabilere Oszillatoren und leistungsfähigere Antennen mit rauschärmeren Eingangsstufen in den Empfangsgeräten zu verwenden.

2.2.2. GPS-Empfänger

Um die Funktionsweise eines GPS-Empfängers besser verstehen zu können, wird zuerst dessen genereller Hardware-Aufbau skizziert. Im weiteren Verlauf wird ein standardisiertes Austauschformat vorgestellt, mit welchem Daten von einem GPS-Empfänger akquiriert werden können. Das bestehende eingebettete System nutzt als GPS-Empfänger den NEO-7 des Herstellers u-blox . Um dessen Leistungsfähigkeit besser einschätzen zu können, schließt dieser Abschnitt mit einem Vergleich mit anderen gängigen GPS-Empfängern.

Hardware-Aufbau

Jedes Empfangsgerät verfügt über eine Antenne über welche die Satellitensignale empfangen werden. Des Weiteren ist es damit möglich DGPS-Signale zu empfangen, um so die Positionsgenauigkeit zu erhöhen. Die empfangenen Signale werden von einer CPU verarbeitet. Diese extrahiert aus den Signalen unter anderem die Position des Empfängers. Die extrahierten Daten werden über eine Hardware-Schnittstelle wie UART oder USB zur Verfügung gestellt. Die Daten können an der Hardware-Schnittstelle im standardisierten NMEA-Format[2] oder in einem herstellerspezifischen Format abgerufen werden. Abbildung 2.4 gibt einen Überblick über den Hardware-Aufbau eines GPS-Empfängers.

[1] Genannt Satellite Based Augmentation System (SBAS).
[2] NMEA steht für National Marine Electronics Association.

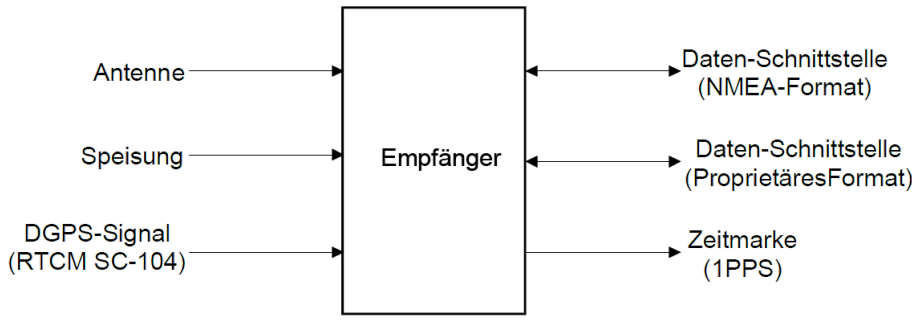


Abbildung 2.4.: Aufbau GPS-Empfänger [23, S. 45]

Im Car2X-Umfeld werden hochgenaue Zeit- und Positionsdaten benötigt, um zum Beispiel Kollisionen vermeiden zu können. In diesem Themenfeld ist mit hohen Geschwindigkeiten zu rechnen und es sollte häufig eine Positionsänderung detektiert werden. Aus diesem Grund ist die Update-Rate eines Empfängers ein entscheidendes Kriterium für die Positionsgenauigkeit. Die Update-Rate eines Empfängers entscheidet, wie oft vom Empfänger die Satellitensignale ausgewertet werden. Bei vielen Empfängern ist die Update-Rate fest auf 1 Hz eingestellt. Der GPS-Empfänger des verwendeten eingebetteten Systems hat eine maximale Update-Rate von 10 Hz. Ein hohe Update-Rate verspricht zwar eine häufige Auswertung der Position aber hat auch einen hohen CPU-Auslastung zur Folge.

GPS-Empfänger können über verschiedene Hardware-Schnittstellen angebunden werden. Um die Daten vom Empfänger möglichst latenzfrei verarbeiten zu können, sollte die Schnittstelle eine separate Interrupt-Leitung besitzen und damit nicht auf Polling angewiesen sein. Tabelle 2.2 bietet eine Übersicht über mögliche Schnittstellen.

Tabelle 2.2.: Übersicht Schnittstellen

Schnittstelle	**Max. Übertragungsrate**	**ISR-Leitung**
CAN	1 Mbit/s	Nein
I^2C	5 Mbit/s	Nein
SPI	-[1]	Nein
UART (RS232)	115,2 kbit/s	Ja
USB	5 Gbit/s	Nein

NMEA

Damit unterschiedliche Navigationsgeräte wie zum Beispiel GPS-Empfänger oder Echolot Daten austauschen können, wurde ein Standard geschaffen. Dieser Standard

wurde von der National Marine Electronics Association (NMEA) ausgearbeitet. Die NMEA ist die US-amerikanische Vereinigung von Elektronikherstellern der Schifffahrtsindustrie. Im Jahr 1980 wurde die erste Version unter dem Namen NMEA 0180 veröffentlicht. Im Jahr 1983 folgte NMEA 0183, wobei diese im Jahr 2012 überarbeitet und als NMEA 0183 Version 4.1 veröffentlicht wurde. Der zuletzt ausgearbeitet Standard NMEA 2000 wird bisher nur von wenigen Geräten unterstützt.

NMEA 0183 sieht für den Datenaustausch eine RS422-Schnittstelle vor. Die Daten, die ein Navigationsgerät anbietet, werden dabei in Sätze eingeteilt und im ASCII-Format übertragen. Ein Satz beginnt mit „$“ und endet mit „Carriage Return“ plus „Line Feed“. Die Datenfelder eines Satzes werden durch Komma getrennt. Jeder Satz transportiert unterschiedliche Informationen. Der Satz GLL (Geographic Position Latitude/Longitude) transportiert beispielsweise nur den aktuellen Längen- und Breitengrad sowie die Uhrzeit. Die meisten GPS-Empfänger unterstützen nur einen Bruchteil der spezifizierten Sätze. Der Satz RMC (Recommended Minimum Sentence C) soll von allen GPS-Empfängern unterstützt werden. Tabelle 2.3 beschreibt daher den folgenden RMC-Satz genauer.

$GPRMC,123005.00,A,4807.99950,N,01131.66952,E,2.027,,130814,,,A*76

Tabelle 2.3.: RMC-Datensatz

Feld	**Erklärung**
$	Anfang Datensatz
GP	Satz aus GPS
RMC	Typ des Satzes
123005.00	Uhrzeit gemäß UTC (HHMMSS.SS)
A	Status-Flag (A = gültig, V = ungültig)
4807.99950	Breitengrad
N	Ausrichtung (N = North, S = South)
01131.66952	Längengrad
E	Ausrichtung (E = East, W = West)
2.027	Geschwindigkeit in Knoten
,<fehlt>,	Kurs in Grad bezüglich Norden
130814	Datum (DDMMYY)
,<fehlt>,	Deklination[1]
,<fehlt>,	Richtung der Deklination (E = East, W = West)
A	Signalintegrität (A = Autonomous, D = Differential, ...)
76	Prüfsumme (XOR aller Daten zwischen $ und *)

[1]Bezeichnet den Winkel zwischen magnetischem und geografischen Norden.

Neben NMEA unterstützen viele GPS-Empfänger ein herstellerspezifisches — üblicherweise binäres — Protokoll. Gegenüber NMEA bietet ein binäres Protokoll einige Vorteile. Es können mehr und genauere Informationen übertragen werden, so wird zum Beispiel die Zeit bei NMEA-Sätzen auf hundertstel Sekunden genau übertragen. Das binäre Protokoll des Herstellers u-blox hingegen liefert im Paket „NAV-TIMEUTC“ die Zeit auf Nanosekunden genau ([20, S. 170]. Ein binäres Protokoll erlaubt zudem eine höhere Datendichte gegenüber ASCII-basiertem Nachrichten und ermöglicht somit eine höhere Ausgabe-Frequenz bei gleichbleibender Datenrate.

Vergleich verschiedener GPS-Empfänger

Um die Leistungsfähigkeit des vorliegenden GPS-Empfängers u-blox NEO-7N besser beurteilen zu können, bietet Tabelle 2.4 einen Vergleich mit anderen gängigen GPS-Empfängern. Die Daten wurden den Datenblättern [11], [14], [18], [21] und [20] entnommen. Für die Angaben zur horizontalen Genauigkeit gilt oft nur ein Konfidenzniveau von 68% oder darunter.

Tabelle 2.4.: Vergleich verschiedener GPS-Empfänger

Hersteller	**Modell**	**Genauigkeit**			**Update-**
		Horizontal [m][1]	**Geschw. [m/s]**	**PPS [ns]**	**Rate**
MediaTek	MT3332	-/-	-	10	5
Navilock	EM-506	2,5/-	0,01	-	1
Trimble	BD982	0,008[2]/0,25	0,007	-	50
u-blox	NEO-7N	2,5/2,0	0,1	60	10

2.3. Pulse per Second zu Verbesserung der Zeitgenauigkeit

Empfänger sorgen für eine Auswertung der Satellitensignale auf der Erde. Nach der Auswertung sind Position und Zeit des Empfängers bekannt. Diese und weitere empfängerspezifischen Informationen können dann durch einen Computer ausgelesen werden.

Zwischen der Zeit- und Positionsermittlung im Empfänger und der Verarbeitung

[1] Ohne/Mit DGPS.

[2] Wird erreicht indem mehrere Empfänger genutzt werden. Für eine genauere Erläuterung dieser Technik, genannt Network Real-Time Kinematic (NRTK), siehe [5].

der Informationen am Computer vergeht eine unbekannte Zeit t. Um diese Zeit ermitteln zu können bieten einige Empfänger einen sogenannten Pulse Per Second (PPS) an. Der PPS wird durch eine Flanke an einem Ausgangspin des Empfängers signalisiert, wenn der Empfänger eine gültige Zeit mit Hilfe der Satellitensignale ermittelt hat. D.h. kann keine Zeit ermittelt werden, z.B. bei schlechter Sichtverbindung zu den Satelliten, wird kein PPS erzeugt.

Ein PPS kann auch mehrmals pro Sekunde ausgelöst werden. Der Puls richtet sich nach der Update-Rate des Empfängers. Wird zum Beispiel alle 0,5 Sekunden die Zeit an Hand von Satellitensignalen ermittelt, so würde alle 0,5 Sekunden eine Flanke am Ausgangspin erzeugt. Ein Puls der nicht an eine Frequenz von 1 Hz gebunden ist, wird allgemein als Zeitpuls bezeichnet. Abbildung 2.5 veranschaulicht das Signalverhalten eines GPS-Empfängers. Der Pin „TIMEPULSE“ wird für die Generierung des PPS verwendet. Über „Serial Data Out“ werden die Daten von einem Computer ausgelesen.

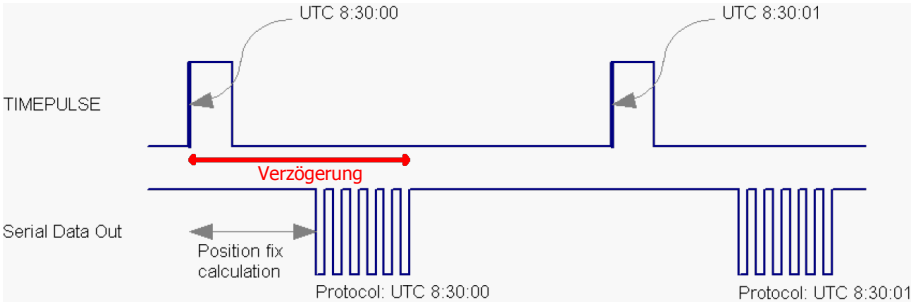


Abbildung 2.5.: Signalverhalten PPS [20, S. 29]

Aus Software-Sicht lässt sich die Verzögerungszeit t in zwei Schritten bestimmen:

1. Erzeuge Zeitstempel t_1 bei auftreten des PPS-Interrupts.
2. Erzeuge Zeitstempel t_2, wenn Computer Daten von Empfänger verarbeitet.

Die Verzögerung t beträgt dann

$$t = t_2 - t_1 \tag{2.6}$$

Durch ein PPS-Signal kann ausschließlich die Zeitgenauigkeit erhöht werden, indem auf den Zeitstempel der GPS-Nachricht die Verzögerungszeit t addiert wird. Eine mögliche Positionsänderung im Zeitraum t bleibt unberücksichtigt.

2.4. Zeitsynchronisation in Netzwerken durch Network Time Protocol

Die bisherigen Abschnitte 2.2 und 2.3 haben die technischen Gegebenheiten beschrieben, welche durch das bestehende eingebettete System des ESK erfüllt werden. Die folgenden Abschnitte beleuchten Aspekte, die für die Implementierung einer Software zur hochgenauen Zeit- und Positionsbestimmung wichtig sind.

In der Architektur die das ESK verfolgt, sollen alle Fahrzeuge mit einem eingebetteten System ausgestattet werden, welches über GPS mit aktuellen Zeit- und Positionsdaten versorgt wird. Diese Zeit- und Positionsdaten werden unter den Fahrzeugen ausgetauscht, um Warnungen in Gefahrensituation zu erzeugen. Durch Einsatz von GPS wird sichergestellt, dass alle Kommunikationsteilnehmer über die gleiche Zeitbasis verfügen. Die Zeitinformationen aus GPS werden genutzt, um den lokalen Zeitgeber des eingebetteten Systems anzupassen. Würde nun der lokale Zeitgeber einfach mit den Informationen aus GPS überschrieben werden, könnte es dazu führen, dass die Zeit des eingebetteten Systems „rückwärts“ läuft. Steht der lokale Zeitgeber zum Beispiel bei 08:00:00 und bekommt über die GPS die Zeit 07:59:55, so würde einfach die neue Zeit auf 07:59:55 gesetzt was zu Anwendungsfehlern führen könnte. Zum Beispiel würde der Empfang eines Netzwerkpakets mit 08:00:00 protokolliert und bei der Auswertung des Pakets wäre die Uhr wieder auf 07:59:55 zurückgestellt. Die Verarbeitung eines Pakets aus der Zukunft wäre die Folge. Das NTP stellt sicher, dass die Zeit bei allen Teilnehmern monoton steigt. Eine umfassende Beschreibung von NTP findet sich in RFC 5905. Es folgt ein kurzer Abriss zu dessen grundsätzlicher Funktionsweise.

2.4.1. NTP-Architektur

NTP nutzt eine Client/Server-Architektur. Server dienen als Zeitquelle, wobei Server wiederum mit verschiedenen (exakten) Zeitgebern kommunizieren, z.B. direkt mit einer Atomuhr oder einem GPS-Empfänger. Auf Client-Seite läuft ein Daemon, der periodisch die aktuelle Zeit vom Server abruft und dabei auftretende Latenzen im Netzwerk berücksichtigt. Im NTP wird eine Hierarchie von Servern aufgebaut. Von einem exakten Zeitgeber (genannt Stratum 0) holt sich ein Server (Stratum 1) seinerseits die exakte Zeit. Jede weitere Hierarchie-Stufe erhöht das Stratum-Level und die möglichen Ungenauigkeiten auf Grund von Netzwerkverzögerungen. In der folgenden Abbildung 2.6 sind im linken Bereich verschiedene NTP-Server skizziert. Gestrichelt dargestellt sind Komponenten die auf Client-Seite zu finden sind.

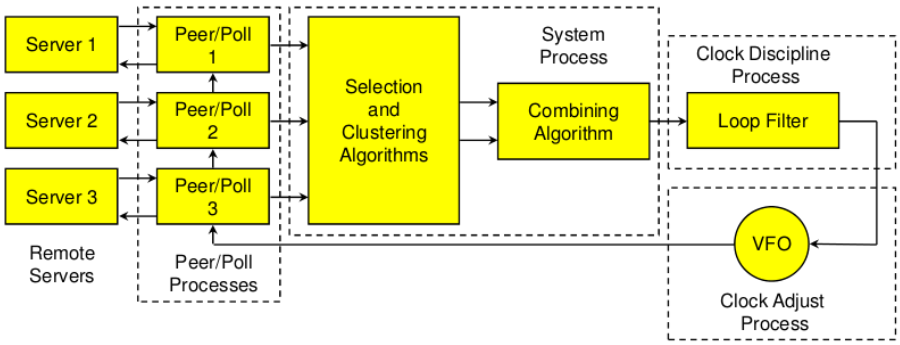


Abbildung 2.6.: NTP-Architektur [13, S. 2]

Weltweit verteilt steht eine Vielzahl von „Remote Servern“ zur Verfügung und bieten den Dienst zur Zeitsynchronisation an. Durch die weltweite Verteilung der NTP-Server werden zwei Ziele erreicht:

1. Redundanz um Ausfälle von mehreren Servern kompensieren zu können.
2. Durch die geografische Verteilung wird sichergestellt, dass ein Client sich mit einem nahegelegenen Server synchronisieren kann. Dadurch werden Netzwerkverzögerungen gering gehalten.

Durch „Peer/Poll Processes“ fragen Clients in Intervallen von 8 Sekunden bis 36 Stunden Server nach der aktuellen Zeit. Der Poll-Prozess ermittelt das Intervall als Kompromiss aus maximaler Genauigkeit und minimaler Netzwerklast. Durch vier Zeitstempel T_n wird die Uhrenabweichung zwischen Client und Server sowie die Netzwerkverzögerung (Round Trip Delay) ermittelt. T_1 ist der Sendezeitpunkt der Client-Anfrage, T_2 ist der Empfangszeitpunkt der Anfrage am Server, T_3 ist der Sendezeitpunkt der Server-Antwort und T_4 ist der Empfangszeitpunkt der Antwort am Client.

$$\text{Uhrenabweichung} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2} \tag{2.7}$$

$$\text{Round Trip Delay} = (T_4 - T_1) - (T_3 - T_2) \tag{2.8}$$

Im „System Process“ werden, die in Peer/Poll Processes gewonnenen, Statistiken zu Uhrenabweichung und Netzwerkverzögerung für einen Server genutzt, um den bestmöglichen auszuwählen. Ein „Clock Select“-Algorithums teilt hierbei Server in korrekt und falsch ein („Truechimers“ und „Falsetickers“). Aus den „Truechimers“ wählt ein „Clock Cluster“-Algorithmus die bestmöglichen Server aus. Diese Server werden dann durch den „Combining“-Algorithmus nochmal unter anderem an Hand der Uhrenabweichung gewichtet. In den Folgeprozessen werden diese Server dann zur Justierung des Client-Zeitgebers genutzt.

Im „Clock Discipline/Adjust Process“ werden aus dem Unterschied zwischen Server- und Client-Zeitstempel Parameter für eine Regelschleife ermittelt. Diese Regelschleife besteht aus einer Phase-Lock Loop (PLL) und einer Frequency-Lock Loop (FLL). Diese halten den Zeitgeber des Clients mit dem des Servers synchron.

2.4.2. Implementierungen und Genauigkeit

Für NTP existieren zwei Implementierungen: `ntpd` (Network Time Protocol Daemon) und `chrony`. Beide erlauben es die Zeitinformationen direkt von einem GPS-Empfänger zu beziehen. GPS-Empfänger werden als Stratum 0 angesehen. Weitere Information hierzu finden sich im Abschnitt 4.4.2 ab Seite 65. Greift NTP direkt auf einen präzisen Zeitgeber wie eine Atomuhr oder einen GPS-Empfänger zurück, kann der lokale Zeitgeber bis auf wenige Mikrosekunden genau justiert werden, da hierbei auch kein Netzwerkverkehr stattfinden muss. In LAN-Umgebungen wird eine Genauigkeit im Millisekundenbereich erreicht. Die NTP-Pakete müssen hierbei in der Regel über mehrere Netzwerkknoten laufen, was zu Ungenauigkeiten bei der Zeitbestimmung führt. In WAN-Umgebungen ist eine Genauigkeit von wenigen zehntel Millisekunden zu erwarten. Vergleiche hierzu [12, S. 2].

2.5. Zeitmessung in Software

Im Verlaufe der Implementierung muss an verschiedenen Stellen Zeit gemessen werden. So zum Beispiel beim Auftreten des PPS-Interrupts und vor der Verarbeitung der eigentlichen GPS-Daten (vergleiche hierzu Abschnitt 2.3 auf Seite 33), um diese Verzögerung bei den Zeitinformationen berücksichtigen zu können. Die Anforderungen sprechen von einer hochgenauen Zeitbestimmung. Daher muss geklärt werden wie genau die Zeit in Software wirklich gemessen werden kann. Jede Software stützt sich grundsätzlich auf Hardware und deren Eigenschaften. Das beim ESK verwendete eingebettete System basiert auf der 80x86-Architektur. Aus diesem Grund wird im Folgenden beleuchtet, welche Hardware bei dieser Architektur für die Zeitmessung zur Verfügung steht.

2.5.1. Hardware-Timer

Auf der 80x86-Architektur stehen folgende Timer immer zur Verfügung:

- Real Time Clock (RTC): Ein batteriegepufferter Timer, der unabhängig von der CPU läuft. Erzeugt periodisch Interrupts zwischen 2 Hz und 8.192 Hz. Die RTC wird vom Betriebssystem nur genutzt, um Datum und Zeit beim Starten abzuleiten.

- Programmable Interval Timer (PIT): Jeder IBM-kompatible PC besitzt mindestens einen PIT, der Interrupts mit einer Frequenz zwischen 100 und 1000 Hz erzeugt. Dieser kann vom Betriebssystem nach dem Start genutzt werden, um die verstrichene Zeit zu ermitteln.

Unter anderem folgende Timer stellen eine höhere Auflösung bereit, werden aber nicht von allen Herstellern angeboten:

- High Precision Event Timer (HPET): HPET stellt bis zu acht unabhängige 32- oder 64-Bit-Timer zur Verfügung. Die Spezifikation für HPET wurde von Intel und Microsoft ausgearbeitet und verlangt, dass die Interrupt-Frequenz mindestens 10 MHz betragen muss.
- ACPI Power Management Timer (ACPI PMT): Der ACPI PMT läuft mit einer unveränderlichen Frequenz von 3,58 MHz.[1]
- Time Stamp Counter (TSC): Dieses, in der CPU befindliche Register, wird bei jedem Taktsignal erhöht. Bei einer Taktfrequenz der CPU von beispielsweise 1 GHz wird es jede Nanosekunde erhöht. Hierdurch wird zwar die höchstmögliche Genauigkeit erreicht, jedoch kann sich die Taktrate der CPU im laufenden Betrieb durch Stromspartechniken ständig ändern. Die Zeitermittlung durch TSC ist deshalb fehleranfällig. Das Paper [15] beschreibt die notwendigen Schritte, um mit Hilfe des TSC-Registers die Ausführungszeit von C-Code in einer Linux-Umgebung zu messen.

2.5.2. Nutzung von Hardware-Timern durch Systemaufrufe

Das Betriebssystem nutzt je nach Verfügbarkeit die oben beschriebenen Timer zur internen Zeitverwaltung und bietet verschiedene Systemaufrufe an, um die aktuelle Zeit an Anwendungen zu geben. Auf dem eingebetteten System des ESK wird ein Linux-basiertes Betriebssystem eingesetzt. Deshalb stehen unter anderem folgende Linux-Systemaufrufe zur Verfügung:

- `time()`: Gibt Anzahl der Sekunden seit Mitternacht 1. Januar 1970 zurück.
- `gettimeofday()`: Gibt die Struktur `timeval` zurück. Sie enthält die Anzahl an Sekunden und Mikrosekunden seit Mitternacht 1. Januar 1970. Wie genau die Anzahl der Mikrosekunden bestimmt werden kann, hängt von den verfügbaren Timern ab. Linux nutzt in absteigender Reihenfolge HPET, ACPI PMT und TSC. Sollte keiner dieser Timer verfügbar sein, wird auf PIT zurückgegriffen (vergleiche [2, S. 253]).

[1] ACPI steht für Advanced Configuration and Power Interface und ist ein plattformunabhängiger und offener Standard für die Geräte- und Energieverwaltung durch das Betriebssystems ohne Beteiligung durch das BIOS.

Das eingebettete System verfügt über HPET. Daher ist bei der Zeitmessung mit einer Genauigkeit von einer Mikrosekunde zu rechnen.

2.6. Auswirkung von Scheduling-Strategien auf das Zeitverhalten

Auf dem eingebetteten System des ESK werden bereits Programme ausgeführt, um Anwendungsfälle abzuwickeln, welche keine harten Zeitanforderungen besitzen. Die unter Abschnitt 1.1 auf Seite 18 vorgestellten Anwendungsfälle hingegen verlangen nach harten Antwortzeiten. Zum Beispiel verlangt der Anwendungsfall Intersection Collision Warning nach einer zeitnahen Kollisionswarnung. Alle Programme des ESK laufen auf einem Linux-basierten Betriebssystem. Aus diesem Grund wird im folgenden Abschnitt 2.6.1 das Scheduling in Linux untersucht. Der Abschnitt 2.6.2 stellt eine Möglichkeit vor, Linux so anzupassen, sodass auch harte Echtzeitbedingungen eingehalten werden können.

2.6.1. Scheduling in Linux

Der Linux-Kernel wurde — und wird — als „General Purpose Operating System“ (GPOS) entwickelt und hat damit zum Ziel viele Prozesse möglichst fair zu bedienen und zu verwalten. Im Gegensatz dazu existieren „Real-Time Operating Systems“ (RTOS), welche daraus ausgerichtet sind Prozesse so zu bedienen, dass bestimmte Zeitanforderungen eingehalten werden.

Die Komponente eines Betriebssystemkerns, welche ermöglicht das mehrere Prozesse die Hardware nutzen, heißt Scheduler. Der Scheduler entscheidet wann ein Prozesswechsel stattfindet und welcher Prozess als nächstes die CPU verwenden darf. Die grundlegende Funktionsweise des Schedulings lässt sich in zwei Schritten darstellen:

1. Ein Timer mit ausreichend hoher Frequenz erzeugt kontinuierlich Interrupts, zum Beispiel mit einer Frequenz von mindestens 1 kHz. Diese Interrupts erlauben dem Kernel die globale Zeit zu verwalten, indem bei jedem Interrupt eine Variable hochgezählt wird.

2. Bei jedem Timer-Interrupt entscheidet der Scheduler, ob der aktive Prozess die CPU weiter benutzen darf oder ein Prozesswechsel stattfinden soll.

Scheduling-Strategien

In Linux wird für jeden Prozess eine Scheduling-Strategie und eine statische Priorität vergeben. Der Scheduler entscheidet mit Hilfe dieser Faktoren, welcher Prozess

als nächstes die CPU nutzen darf. Linux bietet drei „normale“ (nicht echtzeitfähige) Scheduling-Strategien an: SCHED_OTHER, SCHED_BATCH und SCHED_IDLE. Zusätzlich werden mit SCHED_FIFO und SCHED_RR zwei echtzeitfähige Scheduling-Strategien angeboten.

Linux erlaubt unterschiedlichen Prozessen verschiedene Scheduling-Strategien zuzuordnen. Durch die Angabe eines Verhältnisses wird gesteuert, wie mit Prozessen mit „normaler“ Strategie umzugehen ist, wenn echtzeitfähige Prozesse existieren. Eine Angabe von 95% bedeutet zum Beispiel, dass 5% der CPU-Zeit für „normale“ Prozesse verbleiben. Die restliche CPU-Zeit ist für echtzeitfähige Prozesse gedacht[1]. Dies unterstreicht die Tatsache, dass Linux als GPOS konzipiert ist und nicht als reines RTOS. Die unterschiedlichen Scheduling-Strategien werden im Folgenden näher erläutert.

SCHED_OTHER kann nur mit einer statischen Priorität von 0 verwendet werden. Bei SCHED_OTHER darf ein Prozess ein bestimmtes Zeitquantum rechnen. Ist dieses Zeitquantum aufgebraucht, wird zu einem anderen Prozess gewechselt. Das Zeitquantum bestimmt der Scheduler. Bei SCHED_OTHER wird versucht die Rechenzeit fair zwischen den Prozessen aufzuteilen. Das Zeitquantum kann durch die Angabe eines nice-Wertes beeinflusst werden. Der nice-Wert drückt aus wie „nett“ ein Prozess anderen Prozessen gegenüber ist. Durch negative nice-Werte wird anderen Prozessen Rechenzeit genommen. Durch positive nice-Werte erhalten andere Prozesse zusätzliche Rechenzeit. Gültige nice-Werte reichen von -20 bis +19 und können durch das Programm `renice` verändert werden.

SCHED_BATCH verhält sich wie SCHED_OTHER mit einem Unterschied: Prozesse welche die Strategie SCHED_BATCH nutzen werden vom Scheduler als CPU-intensiv angesehen und geben ungern Rechenzeit ab. I/O-intensive Prozesse hingegen geben immer Rechenzeit ab, wenn sie auf eine Ein- oder Ausgabe durch Peripheriekomponenten warten. SCHED_BATCH-Prozesse werden deshalb vom Scheduler leicht bestraft und nicht so häufig aufgeweckt wie SCHED_OTHER-Prozesse.

SCHED_IDLE ist für Prozesse mit extrem niedriger Priorität gedacht. Die statische Priorität muss 0 betragen und ein nice-Wert wird nicht berücksichtigt.

SCHED_FIFO kann nur mit statischen Prioritäten höher als 0 genutzt werden. Die Echtzeitprioritäten reichen von 1 (niedrig) bis 99 (hoch). Prozesse mit echtzeitfähigen Scheduling-Strategien wie SCHED_FIFO oder SCHED_RR erhalten vom Scheduler den Vorzug gegenüber den „normalen“ Scheduling-Strategien. Ein SCHED_FIFO-Prozess wird nur unterbrochen, wenn ein anderer Prozess mit höhe-

[1] Siehe `https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt`.

rer Priorität aufkommt, der Prozess durch einen I/O-Aufruf blockiert wird oder der Prozess selbst Rechenzeit abgibt, zum Beispiel durch den Systemaufruf `sched_yield()`.

SCHED_RR folgt dem gleichen Prinzip wie SCHED_FIFO: Nur der Prozess mit der höchsten Priorität darf laufen. SCHED_RR führt jedoch noch ein Zeitquantum für einen Prozess ein. Ist dieses abgelaufen darf ein Prozess gleicher Priorität rechnen. Es wird also zwischen Prozessen gleicher Priorität gewechselt („Round Robin“). Existiert kein Prozess gleicher Priorität, so verbleibt die Rechenzeit beim aktuellen Prozess wobei dessen Zeitquantum wieder erneuert wird.

Nur mit den echtzeitfähigen Scheduling-Strategien SCHED_FIFO und SCHED_RR kann erzwungen werden, dass ein Prozess Rechenzeit bekommt. Ein bestimmtes Zeitverhalten für einen Prozess kann daher nur mit diesem beiden Scheduling-Strategien erreicht werden.

Completely Fair Scheduler

All die zuvor beschriebenen Scheduling-Strategien müssen durch einen konkreten Scheduler abgearbeitet werden. Der folgende Abschnitt beschreibt den aktuellen Scheduler des Linux-Kernels.

Seit Linux 2.6.23 nutzt der Kernel den „Completely Fair Scheduler“ (CFS). Der vorhergehende Scheduler $\mathcal{O}(1)$ arbeitete mit einem festen Zeitquantum für einen Prozess. Ist das Zeitquantum abgelaufen oder blockiert der Prozess, kommt es zu einem Prozesswechsel. Dieses feste Zeitquantum wurde dynamisch angepasst. So wurden I/O-intensive Prozesse belohnt und das Zeitquantum erhöhte sich. CPU-intensive Prozess dagegen wurden bestraft indem das Zeitquantum verringert wurde. Dies erfordert aber, dass über das Verhalten jedes einzelnen Prozesses genaue Statistiken geführt werden müssen. CFS vermeidet diesen Aufwand und möchte zudem ein faires Scheduling ermöglichen. Das heißt auf einem Desktop-System sollen alle Prozesse die CPU ungefähr gleich lange benutzen dürfen.

Bei CFS wird die Laufzeit für jeden Prozess in einer Variable `vruntime` (in Nanosekunden) protokolliert. CFS wählt den Prozess mit dem kleinsten `vruntime`-Wert (also derjenige Prozess der bisher am wenigsten die CPU genutzt hat). CFS sortiert hierzu alle lauffähigen Prozesse in einen Rot-Schwarz-Baum[1] ein. Für Suchen, Einfügen und Löschen ist somit eine Zeitkomplexität von $\mathcal{O}(\log\ n)$ gegeben. Als Sortierkriterium wird `vruntime` verwendet. Am weitesten links steht der Prozess mit der kleinsten `vruntime`. Dieser wird vom Scheduler als aktiver Prozess ausgewählt. Im Laufe der Zeit steigt `vruntime` dieses Prozesses an und ein anderer Prozess

[1] Der Name für diesen binären Suchbaum beruht auf den beiden Informatikern Leonidas J. Guibas und Robert Sedgewick.

wandert im Baum ganz nach links und initiiert so einen Prozesswechsel. Abbildung 2.7 veranschaulicht einen Rot-Schwarz-Baum

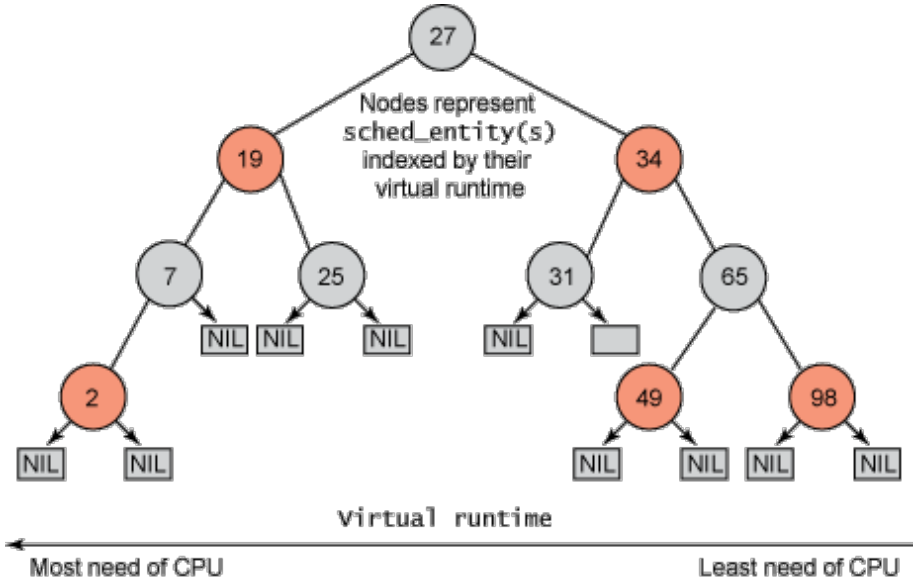


Abbildung 2.7.: Rot-Schwarz-Baum bei CFS ([9, S. 3])

Zwei Parameter erlauben es bei CFS das Scheduling zu beeinflussen. Der Parameter „Granularity“ verhindert dass zu viele Prozesswechsel stattfinden. Der Parameter garantiert somit eine minimale Laufzeit für einen Prozess. Über den „nice-Wert“ kann ein Benutzer die Laufzeit eines Prozesses beeinflussen. Dieses Konzept wurde aus dem alten $\mathcal{O}(1)$-Scheduler übernommen.

CFS unterstützt die unter 2.6.1 vorgestellten Scheduling-Strategien durch die drei Scheduling-Klassen „rt_sched_class“, „fair_sched_class“ und „idle_sched_class“. Die Klasse rt_sched_class verwaltet Prozesse aus SCHED_FIFO und SCHED_RR. Sind in dieser Klasse keine Prozesse vorhanden, so greift die Klasse fair_sched_class. Diese Klasse verwaltet Prozesse aus SCHED_OTHER. Sind gar keine Prozesse abzuarbeiten, greift die Klasse idle_sched_class. Diese Klasse verwaltet den „Idle-Task“.

2.6.2. Echtzeitfähigkeit in Linux durch CONFIG_PREEMPT_RT-Patch

Linux ist als GPOS konzipiert und versucht aus diesem Grund möglichst viele Prozesse gleichzeitig und fair zu bedienen. Das heißt die CPU-Zeit soll auf die vorhandenen Prozesse gleichmäßig verteilt werden, wie im vorhergehenden Abschnitt

dargestellt. Dies erlaubt aber keinerlei Determinismus bezüglich wann welcher Prozess rechnen darf. Durch die Scheduling-Strategien SCHED_FIFO und SCHED_RR wird die Echtzeitfähigkeit erhöht. Eine bestmögliche Echtzeitfähigkeit in einem Betriebssystem wird aber nur erreicht, wenn alle Prozesse bzw. Aufgaben eines Betriebssystems unterbrechbar sind. Das heißt zum Beispiel auch, dass bestimmte Hardware-Interrupts von einem Prozess unterbrochen werden dürfen. Der Standard-Kernel erlaubt nur in drei Situationen einen Prozesswechsel: Die CPU wird von einer Anwendung genutzt, welche die CPU-Zeit aufgebraucht hat. Kernel-Code kehrt von einem Systemaufruf oder einer Interrupt-Service-Routine zurück. Kernel-Code wartet darauf bis ein Mutex freigegeben wird. Möchte zum Beispiel ein Prozess mit höchster Priorität rechnen und befindet sich der Kernel noch in einer Interrupt-Service-Routine, so muss der Prozess warten. Diese Wartezeiten sind in vielen Szenarien mit Echtzeitanforderungen nicht akzeptabel.

Aus diesem Grund wird der CONFIG_PREEMPT_RT-Patch für den Linux-Kernel gepflegt[1]. Der Patch erreicht eine granularere Unterbrechung unter anderem durch die drei folgenden Veränderungen (siehe [19]):

1. Interrupt-Service-Routinen werden zu unterbrechbaren Kernel-Threads konvertiert. Diese besitzen eine `struct task_struct` und können dann wie gewöhnliche Prozesse unterbrochen werden.

2. Priority Inheritance für Semaphoren. Das Problem der Prioritätinversion ist dadurch gelöst.

3. Auf „Busy Waiting“ basierende Synchronisationsmechanismen wie Spinlocks werden durch unterbrechbare Mechanismen wie Mutexe ersetzt.

Der Patch schafft damit die theoretischen Voraussetzungen, um auch harte Echtzeitanforderungen zu erfüllen. Das Paper [10] prüft die Echtzeitfähigkeit des CONFIG_PREEMPT_RT-Patches durch Benchmarking und vergleicht dies mit einem Standard-Linux-Kernel. Durch Tests konnte eine deutlich verbesserte Unterbrechbarkeit des Betriebssystemroutinen festgestellt werden. Diese bessere Unterbrechbarkeit führt dazu, dass Latenzen durch Betriebssystemaktivitäten signifikant reduziert werden konnten. Auf der anderen Seite führt diese verbesserte Unterbrechbarkeit aber zu einer Verschlechterung des Gesamtdurchsatzes des Betriebssystems.

2.7. Software

Der GPS-Empfänger des bestehenden Systems wurde bei der Inbetriebnahme mit dem Linux-Daemon `gpsd` getestet. Für einen ersten Implementierungsansatz wur-

[1] Thomas Gleixner, einer der Hauptentwickler des RT-Patches, gab am 16.10.2014 bekannt, den Patch nur noch als Hobbyprojekt zu pflegen, da es an einer Finanzierung fehlt. Siehe [6]

de auch gpsd verwendet. Aus diesem Grund wird in Abschnitt 2.7.1 der Daemon gpsd genauer betrachtet. Eine maßgeschneiderte Lösung für das ESK soll durch die Implementierung eines eigenen Linux-Treibers erreicht werden. Abschnitt 2.7.2 gibt einen Überblick über wesentliche Eigenschaften der Treiberentwicklung in Linux.

2.7.1. Der Linux-Daemon gspd

Beim gpsd-Projekt handelt es sich um eine Sammlung von Programmen zur Handhabung von GPS-Geräten. Der Daemon mit dem Namen gpsd liest Daten von einem oder mehreren GPS-Empfängern aus, parst die GPS-Daten und stellt diese über einen Software-Socket auf TCP-Port 2947 im JSON-Format zur Verfügung. Damit vereinheitlicht gpsd den Zugriff auf unterschiedliche GPS-Empfänger. GPS-Empfänger können über RS232, USB und Bluetooth angeschlossen sein. Neben dem gängigen NMEA-Format werden über 20 verschiedene Übertragungsformate erkannt, darunter auch das binäre Protokoll UBX des Herstellers u-blox. Funktionalitäten wie das Parsen von GPS-Daten werden im gpsd-Projekt gekapselt und als Software-Bibliotheken anderen Anwendungsprogrammen zur Verfügung gestellt. Es existieren Bibliotheken für C, C++, Python, Java und Perl. Zudem ist der Daemon auch fester Bestandteil des Android-Betriebssystems.

Neben dem Daemon werden zahlreiche Anwendungen mitgeliefert, so zum Beispiel gpsmon, gpsprof und gpsctl. Mit gpsmon können die GPS-Daten eines Empfängers in Echtzeit angezeigt werden. gpsprof ist eine Profiler-Anwendung, um die Genauigkeit des GPS-Empfängers zu bestimmen und Latenzen bei der Verarbeitung aufzuspüren. Mit gpsctl lassen sich Konfigurationsnachrichten an den Empfänger schicken. gpsd eignet sich auch als präziser Zeitgeber, indem es als Zeitquelle für ntpd dient und PPS-Signale eines GPS-Empfängers berücksichtigt.

Das gpsd-Projekt umfasst ca. 52.000 Zeilen Code, wobei der Großteil in C umgesetzt ist. Zum Testen des Codes wird kein Unit-Testing-Framework. Stattdessen werden eigene Test-Routinen verwendet und zur statischen Code-Analyse die Werkzeuge splint und valgrind verwendet.

Der Daemon gpsd besteht aus vier logischen Schichten: Packet Sniffer, Drivers, Core Library und Multiplexer, siehe Abbildung 2.8.

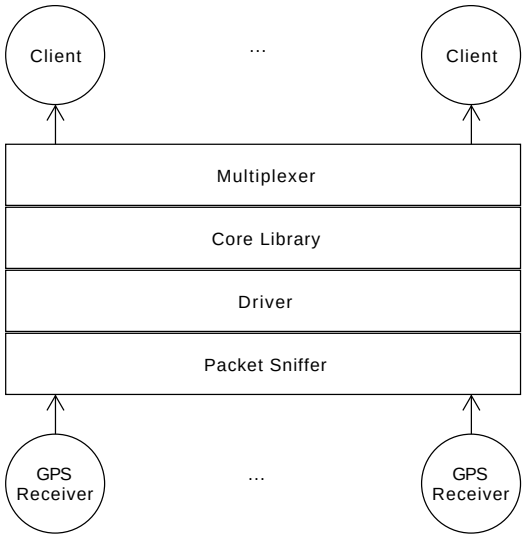


Abbildung 2.8.: Datenfluss im gpsd (nach [3])

Der Datenfluss im `gpsd` wird im Folgenden gemäß dem Bottom-Up-Ansatz erläutert. Die Daten entspringen einem GPS-Empfänger und werden durch die verschiedenen Software-Schichten nach oben bis zu einer Anwendung gereicht.

- Der Packet Sniffer parst die Ausgabe-Daten des GPS-Empfängers und ist in der Lage unterschiedliche Übertragungsformate zu erkennen.
- Auf der „Driver“-Schicht wird für jeden unterstützten Empfänger ein „Treiber“ zur Verfügung gestellt. Die Hauptaufgabe eines Treiber ist es, aus den übertragenen Paketen die relevanten GPS-Informationen zu filtern, darunter zum Beispiel Zeit, Position und Geschwindigkeit. Diese Treiber sind im Gegensatz zu klassischen Hardware-Treibern nicht auf Systemebene implementiert, sondern auf Anwendungsebene. Die Treiber orientierten sich jedoch an der Treiberimplementierung unter Unix. So wird eine generische Struktur bestehend aus Variablen und Funktions-Pointern verwendet, um den Zugriff auf alle Treiber zu vereinheitlichen. Wird zum Beispiel vom Packet Sniffer ein komplettes Paket erkannt, so kann aus der generischen Struktur die Methode zum Parsen aus dem Feld `.parse_packet` verwendet werden, vergleiche Listing 2.1.

Listing 2.1: Generische Treiberstruktur und konkrete Implementierung

```
/* Generische Treiberstruktur in drivers.c. */
const struct gps_type_t driver_unknown = {
  .type_name      = "Unknown",  /* full name of type */
  ...
  .parse_packet   = generic_parse_input,  /* how to interpret a
      packet */
  ...
};

/* Konkrete Implementierung in driver_ubx.c. */
const struct gps_type_t driver_ubx = {
    .type_name        = "u-blox",    /* Full name of type */
    ...
    .parse_packet     = parse_input,    /* Parse message
       packets */
    ...
};
```

- Die Core Library initiiert die Kommunikation mit einem GPS-Empfänger. Dazu liest es Daten vom Empfänger mit verschiedenen Baudraten und verschiedenen Kombinationen von Start-, Stopp- und Paritätsbit. Dies ist notwendig, da `gpsd` im Voraus keinerlei Kenntnisse über den GPS-Empfänger hat und Empfänger, die über RS232 angebunden sind verschiedene Konfigurationen aufweisen können. Die gelesenen Daten werden dem Packet Sniffer übergeben. Meldet der Sniffer ein korrektes Paket, so wurde die passende Empfänger-Konfiguration ermittelt. Im weiteren Verlauf pollt die Core Library den GPS-Empfänger kontinuierlich nach neuen Informationen und übergibt die Daten der Driver-Schicht. Innerhalb der Core Library wird der Kommunikationsvorgang mit einem einzelnen GPS-Empfänger als Session bezeichnet. Die Core Library ist in der Lage verschiedene Sessions simultan zu verwalten.

- Auf oberster Schicht findet sich der Multiplexer. Diese Schicht stellt den von außen sichtbaren Daemon dar. Er nutzt die darunter liegenden Schichten, um GPS-Daten von verschiedenen Empfänger zu akquirieren. Sogenannte „Exporter“ stellen die GPS-Daten Anwendungsprogrammen zur Verfügung. Es existiert ein Exporter, um die Daten in einen TCP-Socket zu schreiben. Von dort können Anwendungen die GPS-Daten im JSON-Format lesen. Ein weiterer Exporter ist in der Lage die Zeitinformationen aus GPS dem `ntpd` zu übergeben. Dazu schreibt der Exporter in ein Shared-Memory-Segment, welches kontinuierlich vom `ntpd` gelesen wird. Der Daemon legt zusätzlich einen Unix-Domain-Socket an. Über diesen können Kommandos an den Daemon geschickt werden, zum Beispiel um die Konfiguration des GPS-Empfängers zu ändern.

Die Komponenten Packet Sniffer, Driver und Core Library werden in der Bibliothek

`libgpsd` zusammengefasst und können unabhängig vom eigentlichen Daemon `gpsd` genutzt werden. Der Multiplexer ist auch in der Lage das PPS-Signal eines GPS-Empfängers zu verarbeiten. Voraussetzung für die automatisierte Verarbeitung eines PPS ist, dass das PPS-Signal über eine eigene Interrupt-Leitung mit dem Computer verbunden ist. Üblicherweise wird das Signal über die Data-Carrier-Detect-Leitung einer RS232-Verbindung an den Computer übermittelt.

2.7.2. Treiberentwicklung in Linux

Eine umfassende Diskussion zur Treiberentwicklung in Linux findet sich in [4]. Eine kurze Übersicht über wesentliche Eigenschaften wird im Folgenden gegeben.

Der Linux-Kernel erfüllt eine Vielzahl von Aufgaben, die heute von einem Betriebssystem erwartet werden, darunter Prozess- und Speicherverwaltung sowie Ein- und Ausgabe von Daten durch entsprechende Hardware, vergleiche Abbildung 2.9. Um diese Aufgaben strukturiert erledigen zu können, besitzt Linux einen modularen Aufbau. Viele Aufgaben werden in Module — separate Objektdateien — gekapselt und können entweder beim Bauvorgang des Kernels fest in das Kernel-Image integriert werden, oder aber erst zur Laufzeit durch Programme wie `insmod` oder `modprobe` geladen werden.

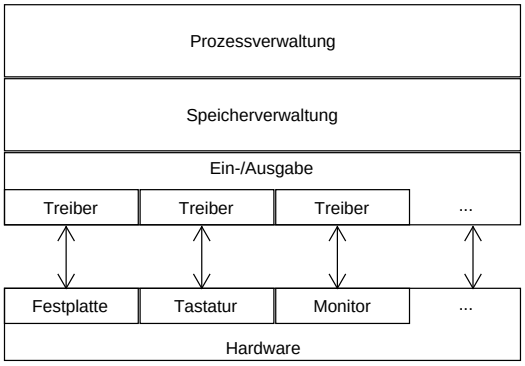


Abbildung 2.9.: Aufgaben eines Betriebssystems

Auch Treiber werden als Module gekapselt. Sie regeln den Zugriff auf konkrete Hardware und sind daher maßgeblich für den Bereich Ein- und Ausgabe verantwortlich. Ein- und Ausgabe im Linux-Kernel wird in verschiedene Subsysteme unterteilt. So gibt es zum Beispiel das USB-Subsystem, welches wiederkehrende Aufgaben der USB-Kommunikation kapselt und bereitstellt. Ein anderes Subsystem stell zum Beispiel Peripheral Component Interconnect (PCI) dar, für den Zugriff auf den PCI-Bus. Der Linux-Kernel stellt Anwendungen die Hardware in Form von Dateien im Verzeichnis `/dev` zur Verfügung und folgt damit dem Unix-Grundsatz „everything is

a file“. Auf diese Dateien können definierte Operationen durchgeführt werden, indem Systemaufrufe abgesetzt werden. Darunter zum Beispiel der `read()`-Systemaufruf zum Lesen von Daten von einem Gerät. Anwendungen können somit einheitlich kommunizieren, egal ob es sich um eine reguläre Datei des Dateisystems handelt oder ein reales Gerät, welches sicher hinter eine Datei unter `/dev` verbirgt. Vergleiche dazu Listing 2.2, welches das Unix-Werkzeug `echo` nutzt, um einen String in ein Ziel zu schreiben. Einmal in eine reguläre Datei und einmal auf Hardware. Die Ausgabe der Hardware erscheint auf der virtuellen Konsole zwei, welche durch die Tastenkombination `Strg` + `Alt` + `F2` erreichbar ist.

Listing 2.2: Schreiben in reguläre Datei und auf Hardware

```
1 echo "Hello, World!" > /tmp/regular_file.txt
2 echo "Hello, World!" > /dev/tty2
```

Für einen Treiber in Linux bestehen grundsätzlich zwei Aufgaben:

1. Registrierung des Treibers am entsprechenden Subsystem. Daraufhin kann für die Hardware ein Eintrag im Verzeichnis `/dev` erzeugt werden. Die Registrierung ermöglicht dem Linux-Kernel eine Zuordnung zwischen der Gerätedatei und dem zu nutzenden Treiber.

2. Implementierung der Methoden, die von der Hardware unterstützt werden und die für die entsprechende Datei im Verzeichnis `/dev` zulässig sind. Linux vereinheitlicht die möglichen Methoden durch Vorgabe der Struktur `struct file_operations`. Diese Struktur erlaubt 28 Operationen, darunter zum Beispiel `read` und `write` und enthält jeweils einen Funktions-Pointer auf die konkrete Implementierung des Treibers. Die Struktur `file_operations` wird üblicherweise bei der Registrierung des Treibers dem Linux-Kernel übergeben.

Abbildung 2.10 veranschaulicht das Zusammenspiel zwischen dem Linux-Kernel und dem konkreten Treiber, am Beispiel eines Lesevorgang mit dem Programm `cat`. Es wird die fiktive Hardware „my_hardware“ und der fiktive Treiber „my_driver “ benutzt.

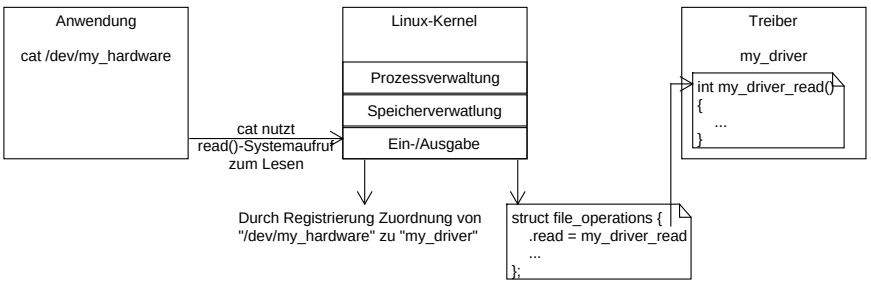


Abbildung 2.10.: Zusammenspiel Linux-Kernel und Treiber

3. Bestehendes System

Das ESK hat bereits ein System bestehend aus Hard- und Software bei Projektpartnern im Einsatz. Dieses System bildet die Grundlage für die technische Umsetzung aus Abschnitt 4 und wird verwendet um auch Anwendungsfälle aus anderen Projekten des ESKs umzusetzen. Um die Anwendungsfälle aus Abschnitt 1.1 umzusetzen, soll das System in Fahrzeugen verbaut werden. Über einen integrierten GPS-Empfänger werden Zeit- und Position des Fahrzeugs ermittelt. Neben dieser Datenakquirierung soll auch das Senden und Empfangen von Car2X-Nachrichten von diesem System durchgeführt werden. Langfristig sollen auch noch Fahrassistenzsysteme auf dem System betrieben werden. Diese werten die Car2X-Nachrichten aus und wirken in Gefahrensituationen unterstützend auf den Fahrer ein. Zum Beispiel durch eine visuelle Warnung auf einem Display oder akustisch durch einen Warnton. Im Folgenden wird Hard- und Software des Systems vorgestellt.

3.1. Hardware

Bei der Hardware handelt es sich um den ARTiS PC (Automotive Real-Time Prototyping System). Der ARTiS PC ist darauf ausgelegt Infotainment- und Car2X-Anwendungen abzuarbeiten und enthält folgende Ausstattungsmerkmale:

- Intel Atom Z520PT mit 1.33 GHz und integriertem Graphic Controller Intel GMA 500
- 1 GB RAM
- 1x DVI
- 2x miniPCI (opt. 1x miniPCIe)
- 2x USB
- 1x Ethernet 10/100/1000 MBit/s
- 1x microSD
- 1x HD Audio in/out
- 1x u-blox NEO-7 GPS-Empfänger

Die Firma embedded brains ist für die Fertigung des ARTiS PC zuständig. Eine eigens entworfene Platine wird mit den obigen Bauteilen bestückt und in einem Metallgehäuse untergebracht, siehe Abbildung 3.1.

Abbildung 3.1.: ARTiS PC in Metallgehäuse[1]

3.1.1. USB-Anbindung des GPS-Empfängers

Der GPS-Empfänger u-blox NEO-7 ist über USB angebunden und neue Daten müssen deshalb durch Polling bezogen werden. Der GPS-Empfänger verfügt über einen Pin zur PPS-Generierung. Für Details zu PPS siehe Abschnitt 2.3 ab Seite 33) und trägt den Namen „timepulse“.

3.1.2. PPS-Detektion durch FPGA

Auf Grund von Beschränkungen im Platinenlayout wird der timepulse-Pin nicht direkt mit dem Interrupt-Controller der CPU verbunden. Auf direktem Wege kann daher nicht der Zeitpunkt ermittelt werden, wann an der Antenne des GPS-Empfängers gültige GPS-Daten anliegen. Um dies trotzdem zu ermöglichen wird ein FPGA (Field Programmable Gate Array) verwendet. Das FPGA verfügt hierzu über eine Möglichkeit Flanken — und damit einen Interrupt-Request — am timepulse-Pin zu erkennen. An Hand von zwei 20-Bit-Registern[1] kann die Zeit seit Auftreten einer Flanke ermittelt werden. Das Register „Counter“ wird bei jedem Taktsignal des FPGAs erhöht. Das FPGA nutzt das Taktsignal des PCI-Busses, welcher mit 33 MHz getaktet ist. Der Takt kann durch einen 1-zu-64-Prescaler auf 515.625 Hz reduziert werden. Tritt ein Flanke am timepulse-Pin auf, wird der aktuelle Wert von Register Counter in Register „Capture“ kopiert. Werden die dann folgenden GPS-

[1] Quelle: `http://www.esk.fraunhofer.de/de/projekte/ARTiS-PC/_jcr_content/stage/image.img.jpg/RS3497_ARTiS-PC_transparent_mitSchatten.1381411478094.jpg`

[1] Die eigentlichen Register sind 24 Bit breit, jedoch werden nur 20 Bit als Zähler verwendet.

Daten von einer CPU im Computer verarbeitet, kann der aktuelle Counter-Wert mit dem gespeicherten Capture-Wert verglichen werden. Durch Berücksichtigung der Taktfrequenz des FPGAs kann die Zeit in Sekunden seit Auftreten des PPS ermittelt werden, vergleiche Gleichung 3.1.

$$t_{\mathrm{PPS}} = \frac{\mathrm{Counter} - \mathrm{Capture}}{f_{\mathrm{FPGA}}} \tag{3.1}$$

t_{PPS} ist die Zeit seit Auftreten von PPS in Sekunden. Counter und Capture enthalten Zählerstände gemäß der FPGA-Taktfrequenz f_{FPGA}. Hinweis: Überläufe der Register Counter und Capture sind hierbei nicht berücksichtigt.

Vier Register des FPGAs werden an den Low-Pin-Count-Bus (LPC) der ARTiS-Platine angebunden. Der LPC-Bus wurde von Intel entwickelt, um Hardware aus der ISA-Ära, wie zum Beispiel einen Disketten-Controller, auch in neueren Computersystemen verwenden zu können ohne den veralteten ISA-Bus mitführen zu müssen. Über Adressen am LPC-Bus hat die CPU die Möglichkeit die Register zu lesen und zu schreiben. Tabelle 3.1 bietet einen Überblick über diese vier Register.

Tabelle 3.1.: Übersicht der FPGA-Register [7, S. 13]

Addresse	**Zugriff**	**Breite (Bit)**	**Register**
0x0300	R/W	8	Control
0x0301	R/-	8	Status
0x0304	R/-	24	Counter
0x0308	R/-	24	Capture

Über das Control-Register kann unter anderem der Prescaler aktiviert und deaktiviert werden. Das Status-Register zeigt an, ob am PPS-Pin des GPS-Empfängers ein Interrupt erzeugt wurde. Die Register Counter und Capture erlauben die Zeit seit Auftreten des Pulses zu messen. Abbildung 3.2 visualisiert die PPS-Detektion durch das FPGA.

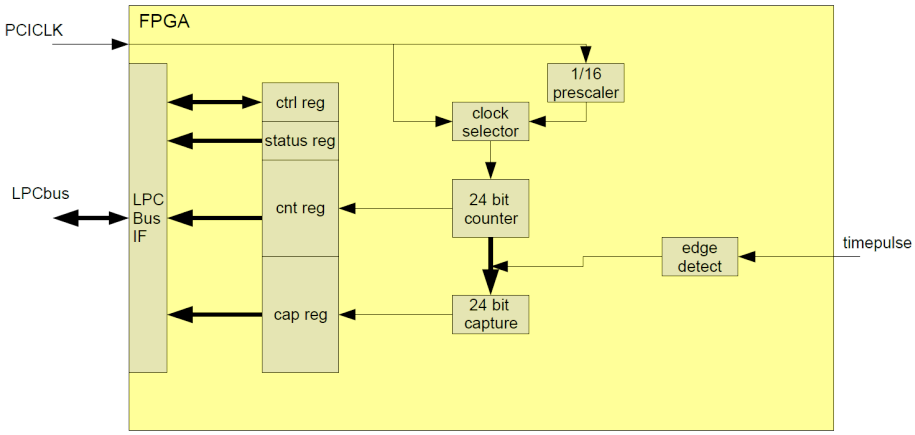


Abbildung 3.2.: Blockschaltbild FPGA [7, S. 13]

3.2. Software

Auf dem ARTiS PC wird ein Linux-basiertes Betriebssystem eingesetzt. Es handelt sich um Debian GNU/Linux 7.5 (Codename „Wheezy“) mit dem Linux-Kernel 3.2.0-4 in der 32-Bit-Ausführung. Im späteren Verlauf soll auf dem ARTiS PC das Software-Framework zur Car2X-Kommunikation ausgeführt werden. Das Framework soll die eigentliche Kommunikation über WLAN und Mobilfunk abwickeln sowie die Anwendungsfälle aus Abschnitt 1.1 abdecken.

4. Implementierung

Ausgehend vom System aus Abschnitt 3 soll eine Software-Lösung entworfen werden, um hochgenaue Zeit- und Positionsdaten zu erhalten. Diese Daten werden von einem Framework genutzt, um die Anwendungsfälle Emergency Electronic Brake Lights, Intersection Collision Warning und Co-Operative Forward Collision Warning umzusetzen (für eine Beschreibung der Anwendungsfälle siehe Abschnitt 1.1 auf Seite 18). Abschnitt 4.1 geht auf Anforderungen ein, die von der Implementierung berücksichtigt werden sollen. Abschnitt 4.2 beschreibt Probleme, die bei der Inbetriebnahme der Hardware auftraten. Ein exaktes Referenzsystem zur Verifikation der bei der Implementierung gewonnen GPS-Daten ist nicht vorhanden. Aus diesem Grund muss eruiert werden wie trotzdem die Richtigkeit der Implementierung sichergestellt werden kann. Abschnitt 4.3 geht auf diese Thematik ein. Die Implementierung folgt einem iterativen Ansatz. Zuerst wird eine Lösung auf Anwendungsebene auf Basis existierender Software entwickelt, um eine schnelle Abschätzung bezüglich der Zeitgenauigkeit zu erhalten. Dieser Ansatz wird in Abschnitt 4.4 ausgeführt. Um eine Verbesserung und leichter wartbaren Code zu erhalten, wird im weiteren Verlauf ein zweiter Lösungsansatz auf Betriebssystemebene als Linux-Kernel-Modul entwickelt, welcher in Abschnitt 4.5 vorgestellt wird. Während der Implementierung zeigte sich eine ungewöhnlich hohe Verzögerung zwischen Eintreffen von GPS-Daten an der Antenne des GPS-Empfängers und deren Verarbeitung in der CPU des ARTiS PC. Durch eine Änderung der Scheduling-Strategie im Kernel-Modul wird versucht diese Verzögerung zu minimieren. Dieses Vorgehen wird in Abschnitt 4.6 erläutert.

4.1. Anforderungen an die Implementierung

Das ESK hat als Anforderung formuliert, dass die bestehende Hardware bestmöglich genutzt werden soll. Durch diese Formulierung lässt sich eine Erfüllung der Anforderung nur schwer messen. Jedoch lässt die Beschreibung der Anwendungsfälle im Technical Report [8] Rückschlüsse auf die geforderte Genauigkeit bezüglich Zeit und Position zu. Die drei Anwendungsfälle Emergency Electronic Brake Lights, Intersection Collision Warning und Co-Operative Forward Collision Warning verlangen, dass Car2X-Nachrichten mit einer minimalen Frequenz von 10 Hz gesendet werden. Das heißt mindestens alle 100 ms muss eine neue Zeit- und Positionsinformation bekannt sein, um eine Entscheidung treffen zu können. Dies muss auch von der Implementierung berücksichtigt werden.

Zudem verlangt der Anwendungsfall Co-Operative Forward Collision Warning, dass die relative Position eines Fahrzeugs bis unter 1 m genau bestimmt werden kann. Die relative Position nimmt dabei Bezug auf ein anderes Fahrzeug innerhalb einer Fahrzeugkolonne. Ein Fahrzeug sendet seine Positionsdaten durch Car2X-Nachrichten an andere Fahrzeuge. Die Empfänger müssen nun auf Basis ihrer Positionsdaten und den empfangenen Positionsdaten entscheiden wie weit sie vom Sender entfernt sind. Um eine relative Positionsgenauigkeit von unter 1 m zu erreichen, ist es erforderlich, dass die die absoluten Positionsinformation mindestens eine ebenso hohe Genauigkeit aufweisen.

Das ESK führt auf dem eingebetteten System ARTiS PC eine Vielzahl von Anwendungen aus. Diese Anwendungen sollen durch die Akquirierung von GPS-Daten nicht negativ beeinflusst werden. Es lässt sich daher die nichtfunktionale Anforderung der Ressourcensparsamkeit bezüglich CPU- und RAM-Auslastung ableiten.

Der verfügbare GPS-Empfänger u-blox NEO-7 besitzt eine maximale Update-Rate von 10 Hz und erfüllt damit die Minimalanforderungen der Anwendungsfälle bezüglich der Häufigkeit eines Positionsupdates. Daraus folgt, dass neue Positionsinformationen nur in 100-ms-Intervallen bekannt sind. Zwischen den „Abtastpunkten“ sind keine Informationen bekannt. Eine Positionsgenauigkeit von unter 1 m ist jedoch nicht zu erfüllen. Laut Datenblatt [21, S. 6] ist mit DGPS eine Genauigkeit von 2 m zu erwarten. Eine höhere Positionsgenauigkeit kann nur erreicht werden, wenn wie in Abschnitt Stand der Technik ab Seite 23 dargestellt, die Daten eines GPS-Empfängers mit Sensordaten des eigenen Fahrzeugs fusioniert werden. Die Fusion von Sensordaten birgt eine weitere Komplexität, welche das ESK vermeiden möchte. Zudem sind für die Datenfusion Sensordaten des eigenen Fahrzeugs notwendig, was zusätzliche Umbauten an Versuchsfahrzeugen des ESKs erforderlich macht. Aus diesem Grund versucht das Institut durch eine hohe Update-Rate des GPS-Empfängers an ausreichende Informationen zu kommen. Auch wenn für den Anwendungsfall Co-Operative Forward Collision Warning die Genauigkeit durch den GPS-Empfänger nicht erreicht wird, soll trotzdem eine Softwarelösung entwickelt werden, um möglichst genaue Zeit- und Positionsdaten zu liefern. Langfristig lässt sich der GPS-Empfänger durch einen leistungsfähigeren ersetzen.

4.2. Probleme durch die Hardware

Der ARTiS PC ist bereits bei Projektpartnern im Einsatz. Für die Masterarbeit wurde eine neue Version mit integriertem GPS-Empfänger genutzt, für welche ein umfassender Abnahmetest noch nicht abgeschlossen war. Aus diesem Grund wurde zuerst ein grundsätzlicher Funktionstest des GPS-Empfängers und auch des FPGAs

durchgeführt. Dabei zeigten sich zwei Fehlverhalten des FPGAs, welche in Abschnitt 4.2.2 näher ausgeführt werden.

4.2.1. Funktionstest GPS-Empfänger mit Linux-Daemon gpsd

Für den Test des GPS-Empfängers kann auf existierende Software zurückgegriffen werden. Es wird der Linux-Daemon `gpsd` genutzt. Eine ausführliche Beschreibung des Linux-Daemons liefert Abschnitt 2.7.1 ab Seite 44.

Der ARTiS PC nutzt als Betriebssystem Debian GNU/Linux 7.5. Über die offiziellen Software-Repositories wird `gpsd` in der Version 3.6 installiert. `gpsd` bringt verschiedene Anwendungsprogramme mit, um die Daten eines GPS-Empfängers zu visualisieren, darunter `gpsmon` und `cgps`. Für den Funktionstest wird `gpsmon` genutzt. Dieses Kommandozeilenwerkzeug zeigt in Echtzeit die empfangenen Daten des GPS-Empfängers an. Abbildung 4.1 visualisiert die Ausgabe zum Testzeitpunkt in der Hansastraße 32, 80686 München. Im oberen Bereich sind das Zeit, Längen- und Breitengrad. Darunter werden alle NMEA-Sätze protokolliert, welche bisher empfangen wurden. Der untere Bereich zeigt den Inhalt der wichtigsten NMEA-Sätze. Abbildung 4.1 zeigt, dass erfolgreich GPS-Daten empfangen werden können.

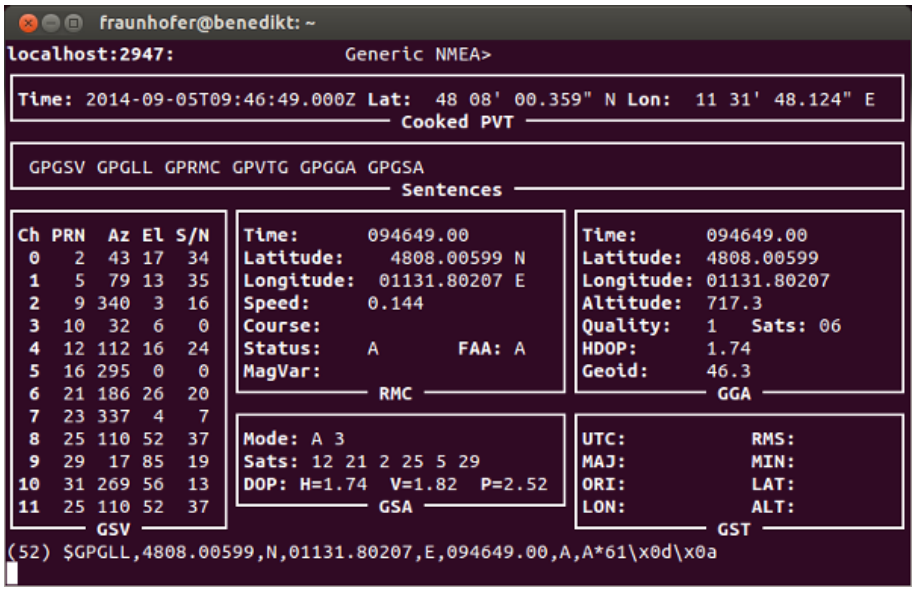


Abbildung 4.1.: Inbetriebnahme GPS-Empfänger mit Programm gpsmon

Mit `gpsctl` kann auch erfolgreich vom NMEA-Format auf das herstellerspezifische, binäre Format „UBX“ umgeschaltet werden, welches auch von `gpsd` unterstützt wird. Eine umfassende Protokollbeschreibung für UBX findet sich in der Empfänger- und Protokollbeschreibung [20]. Die UBX-Aktivierung ist jedoch nicht

mit der installierten Version 3.6 von `gpsd` möglich. Es muss die Version 3.11 von `git://git.savannah.nongnu.org/gpsd.git` bezogen und gebaut werden. Eine erfolgreiche UBX-Aktivierung wird durch `gpsctl -b -t ‘‘u-blox’’ -f /dev/ttyACM0` erreicht. Der Parameter -b aktiviert das binäre Protokoll des GPS-Empfängers, -t spezifiziert den Typ des Empfängers und -f erzwingt den Zugriff auf den Empfänger, der durch die Gerätedatei /dev/ttyACM0 beschrieben wird.

Ein Problem tritt hingegen auf beim Versuch die Update-Rate des GPS-Empfängers von standardmäßig 1 Hz auf 10 Hz zu stellen. Es kann nicht hinreichend verifiziert werden, dass der GPS-Empfänger tatsächlich mit einer Rate von 10 Hz Daten sendet. Eine Konfiguration des GPS-Empfängers ist mit dem Programm `gpsctl` möglich, indem UBX-Nachrichten an den Empfänger gesendet werden. Der Hersteller u-blox garantiert jedoch nur mit dem Programm „u-blox u-center“[1] eine verlässliche Konfiguration. u-blox u-center wird nur von Microsoft Windows unterstützt und kann daher in der Linux-Umgebung des ARTiS PC nicht ausgeführt werden. Daher wird das Werkzeug `gpsctl` folgendermaßen angewendet:

```
./gpsctl -t ‘‘u-blox’’ -x ‘‘\x06 \x08 \x64 \x00 \x01 \x00 \x01 \x00’’ \
-f /dev/ttyACM0
```

Der Parameter -t gibt den Typ des GPS-Empfängers an, -x definiert die UBX-Nachricht für den Empfänger und -f erzwingt die Verwendung des Geräts /dev/ttyACM0. Hinter dieser Gerätedatei verbirgt sich der GPS-Empfänger.

UBX-Nachrichten beginnen mit zwei unveränderlichen Synchronisations-Byte. Dann folgen zwei Bytes, Class und Id, welche einen Befehl beschreiben. Eine Klasse gruppiert ähnliche Befehle. Beispielsweise existiert die Klasse CFG (0x06) zur Konfiguration des Empfängers. Innerhalb dieser Klasse beschreiben Ids konkrete Anweisungen an den GPS-Empfänger. Die Id RATE (0x08) gibt zum Beispiel die Anweisung die Update-Rate des Empfängers zu ändern. Über einen Payload variabler Länge erhalten die Class/Id-Kombinationen Parameter für den Empfänger. Eine Prüfsumme aus zwei Byte komplettiert ein UBX-Paket. UBX-Nachrichten nutzen als Bytereihenfolge Little Endian. Abbildung 4.2 zeigt den Aufbau eines UBX-Pakets.

[1] `http://www.u-blox.com/de/evaluation-tools-a-software/u-center/u-center.html`

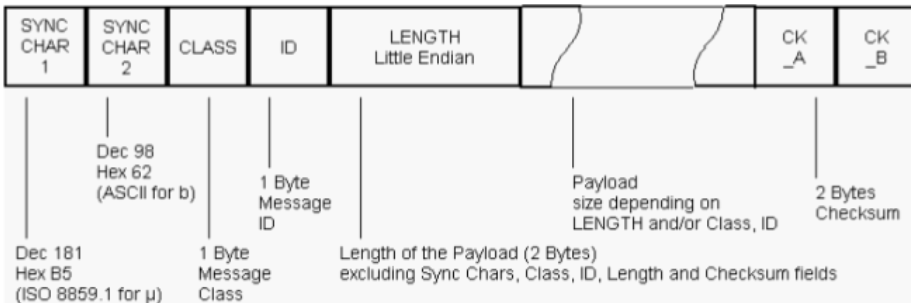


Abbildung 4.2.: Aufbau UBX-Paket [20, S. 86]

Für die Konfiguration mittels `gpsctl` muss ausschließlich der Payload für die entsprechende Class/Id-Kombination angegeben werden. `gpsctl` erzeugt automatisch die Synchronisations-Bytes, die Länge des Payloads und die Checksumme. Der gesendete Payload 0x06 0x08 0x64 0x00 0x01 0x00 0x01 0x00 hat folgende Bedeutung: Sende CFG-RATE (0x06 0x08) mit sechs Byte Payload (0x64 0x00 0x01 0x00 0x01 0x00) an den GPS-Empfänger. Der Payload setzt die Update-Rate auf 100 ms (0x64 0x00), die Navigation-Rate auf 1 (0x01 0x00) und verlangt, das die Zeit im GPS-Format (0x01 0x00) gesendet wird. Ein Übersicht über den Payload für die UBX-Nachricht CFG-RATE liefert Tabelle 4.1.

Tabelle 4.1.: Payload für UBX-Paket CFG-RATE [20, S. 129]

Byte Offset	**Name**	**Einheit**	**Erklärung**
0	measRate	ms	Update-Rate.
2	navRate	cycles	Navigation-Rate. Kann nicht verändert werden.
4	timeRef	-	Sende Zeit im UTC- oder im GPS-Format. 0 = UTC, 1 = GPS

Nach der Konfiguration der Update-Rate mittels `gpsctl` kann im Programm `gpsmon` eine höhere Aktualisierungsfrequenz der angezeigten Daten festgestellt werden. Jedoch fehlt ein Nachweis, dass die GPS-Daten wirklich mit 10 Hz aktualisiert werden. Zudem konnte die Konfiguration nicht verlässlich durchgeführt werden. Einige Versuche die Update-Rate zu ändern bzw. das binäre UBX-Protokoll zu aktivieren scheiterten und der GPS-Empfänger sendete daraufhin keine Daten mehr. Das ursprüngliche Verhalten des GPS-Empfängers konnte durch einen Reset mittels einer kurzen Stromunterbrechung wiederhergestellt werden.

4.2.2. Funktionstest FPGA

Das FGPA wird beim ARTiS PC genutzt, um das PPS-Signal auswerten zu können, vergleiche hierzu Abschnitt 3.1.2 ab Seite 50. Die vier Register des FPGAs sind über den LPC-Bus an die CPU angebunden. Unter Linux lassen sich die FPGA-Register durch I/O-Port-Operationen ansprechen. `ioperm()` reserviert den Zugriff auf einen I/O-Adressbereich für den aktuellen Thread. Lesen und Schreiben der Register erfolgt über Methoden wie `inb()` und `outb()`. In einem ersten Funktionstest wurden die FPGA-Register in einer Endlosschleife gelesen, vergleiche Listing 4.1.

Listing 4.1: Funktionstest FGPA

```
...
#define FPGA_BASE_ADDRESS 0x300
#define FPGA_LENGTH 0x10

#define FPGA_CTRL (FPGA_BASE_ADDRESS + 0x00)
...
int main()
{
  int return_value = 0;
  ...
  return_value = ioperm(FPGA_BASE_ADDRESS, FPGA_LENGTH, 1);
  if (return_value)
    printf("ioperm(0x%x, 0x%x, 1) failed with %d!\n",
        FPGA_BASE_ADDRESS, FPGA_LENGTH, return_value);
  else {
    while (1) {
      fpga_ctrl = inb(FPGA_CTRL);
      ...
      printf("ctrl = 0x%x\n", fpga_ctrl);
      ...
    }
  }

  return return_value;
}
```

Ein Test über mehrere Minuten zeigt, dass sich die Register-Inhalte nicht änderten. Zumindest das Register Counter müsste sich laut Beschreibung im Datenblatt kontinuierlich mit einer Frequenz von 33 Mhz ändern. Dieses offensichtliche Fehlverhalten wurde durch den Hersteller mit einem ersten Firmware-Update behoben. Listing 4.2 und 4.3 visualisieren das Fehlverhalten des FPGAs und das Verhalten nach dem Firmware-Update.

Listing 4.2: Vor Update

```
ctrl = 0xff
stat = 0xff
counter = 0xffffffff
capture = 0xffffffff
ctrl = 0xff
stat = 0xff
counter = 0xffffffff
capture = 0xffffffff
...
```

Listing 4.3: Nach Update

```
ctrl = 0x0
stat = 0x10
counter = 0xff0daca4
capture = 0xff065a98
ctrl = 0x0
stat = 0x10
counter = 0xff0db3a1
capture = 0xff065a98
...
```

Ein zweites Fehlverhalten des FPGAs zeigt sich während der Implementierung. Das Capture-Register wird nicht aktualisiert, obwohl ein PPS-Interrupt aufgetreten ist und dieser auch im Status-Register des FPGAs signalisiert wird. Listing 4.4 zeigt dieses Fehlverhalten. Das Listing zeigt die fünf Spalten timestamp, time_pulse_detected, free_counter, pulse_counter und time_since_pulse. timestamp enthält den Zeitstempel gemäß Unix-Zeit[1] und beschreibt wann die FPGA-Register ausgelesen wurden. Die Spalte time_pulse_detected zeigt, ob ein PPS-Interrupt detektiert wurde. Die beiden folgenden Spalten free_counter und pulse_counter zeigen die aktuellen Zählerstände. Die letzte Spalte zeigt die Zeit in Sekunden, die seit dem letzten PPS vergangen ist. In Zeile fünf des Listings ist zu beobachten, dass zwar ein PPS detektiert wurde (zweite Spalte enthält 1), aber das Capture-Register sich nicht änderte (fünfte Spalte). Dieses Fehlverhalten wurde durch ein zweites Firmware-Update des Herstellers behoben.

Listing 4.4: Fehlverhalten FPGA

```
timestamp; time_pulse_detected; free_counter; pulse_counter;
    time_since_pulse
...
1404222958.063211; 1; 0x98267; 0x8fc18; 0.066682
1404222959.066165; 1; 0x175ce; 0x0e97f; 0.069661
1404222960.062170; 1; 0x95b1b; 0x0e97f; 1.073206
1404222961.066184; 1; 0x150a9; 0x0c44c; 0.069688
...
```

4.3. Korrektheit der Messdaten

Abschnitt 4.1 beschreibt die Anforderungen an die Implementierung. Zeitinformationen sollen mindestens alle 100 ms mit größtmöglicher Genauigkeit geliefert werden. Auch Positionsinformationen sollen mit gleicher Frequenz und unter 1 m genau

[1] Sekunden seit 1. Januar 1970 00:00 Uhr.

bereit stehen. Zudem wird ein geringer Ressourcenverbrauch angestrebt. Vor der Implementierung muss geklärt werden, wie die Korrektheit der Implementierung bezüglich dieser Anforderungen geprüft werden kann.

Um Zeit- und Positionsdaten verifizieren zu können, wird ein Referenzsystem benötigt. Eine Inertial Measurement Unit (IMU) kann beide Daten liefern. Eine IMU verfügt über Sensoren, um Beschleunigung und Drehrate in x- y- und z-Richtung zu messen. Durch zweimalige Integration und Kombination mit anderen Sensordaten für Zeit und Position, um die Integrationskonstanten bestimmen zu können, lassen sich hochgenaue Zeit- und Positionsdaten ableiten. Das ESK verfügte im Zeitraum der Masterarbeit nicht über ein IMU. Aus diesem Grund sind alternative Referenzsysteme für Zeit und Position notwendig.

Um die Zeitinformationen verifizieren zu können, wird NTP genutzt. Dazu wird auf dem ARTiS PC NTP eingerichtet, sodass der Zeitgeber des ARTiS PC mit einem Referenzsystem aus dem Internet synchron läuft. Die bei der Implementierung ermittelte Zeit kann mit dieser Referenzzeit verglichen werden. Für Positionsinformationen kann das ESK kein adäquates Referenzsystem zur Verfügung stellen. Zudem erfüllt der GPS-Empfänger nicht die von Anwendungsfall Co-Operative Forward Collision Warning geforderte Genauigkeit von unter 1 m. Die Positionsinformationen werden deshalb für eine Verifikation nicht betrachtet.

Das Update-Intervall von mindestens 10 Hz soll eingehalten werden und es kann geprüft werden, ob die Implementierung wirklich mindestens alle 100 ms Rechenzeit vom Betriebssystem bekommt, um seine Aufgaben zu erfüllen. Um den geringen Ressourcenverbrauch überprüfen zu können, werden CPU-Auslastung und RAM-Auslastung mit dem Programm `top` in 0,5-Sekunden-Intervallen gemessen.

Die Messergebnisse werden in den einzelnen Abschnitten zur Implementierung besprochen. Alle Daten zu Zeitinformation, Update-Intervall und Ressourcenverbrauch werden über Bash-Skripte gesammelt und gefiltert. Über Python-Skripte erfolgt die Konvertierung in CSV-Dateien. Diese Daten können mittels Microsoft Excel oder einem anderen Programm weiter verarbeitet werden. Dieser automatisierte Prozess erlaubt konsistente Testläufe. Das genaue Vorgehen der Tests wird in den einzelnen Abschnitten zur Implementierung unter 4.4.3 und 4.5.3 besprochen. Die Bash-Skripte können im Anhang eingesehen werden.

4.4. Modifikation des Linux-Daemons gpsd

Bereits zur Inbetriebnahme des GPS-Empfängers wurde der Linux-Daemon `gpsd` verwendet. Bei `gpsd` handelt es sich um ein quelloffenes Projekt und kann daher gut

an eigene Bedürfnisse angepasst werden. Um eine schnelle Abschätzung bezüglich der erwartbaren Zeitgenauigkeit zu erhalten wird auf den `gpsd` zurückgegriffen. Langfristig strebt das ESK jedoch ein eigenes Linux-Kernel-Modul an, um GPS-Daten zu akquirieren und auf Anwendungsebene zur Verfügung zu stellen. Konzeptionell werden für die Implementierung daher zwei Bereiche betrachtet, die Anwendungs- und die Systemebene, vergleiche Abbildung 4.3. Der `gpsd` arbeitet auf Anwendungsebene, das Linux-Kernel-Modul auf Systemebene. Um ein bestehendes Software-Projekt zielgerichtet modifizieren zu können, muss zuerst dessen grundsätzlicher Aufbau verstanden werden. Das `gpsd`-Projekt wird in Abschnitt 2.7.1 ab Seite 44 diskutiert. Abschnitt 4.4.1 beschreibt das Konzept, dass durch die Modifikation des `gpsd`-Codes umgesetzt werden soll. Der Punkt 4.4.2 beschreibt die notwendigen Änderungen am `gpsd`, damit das bestehende System des ESK bestmöglich genutzt werden kann. Abschnitt 4.4.3 bewertet diese Lösung bezüglich der Zeitgenauigkeit und des Ressourcenverbrauchs.

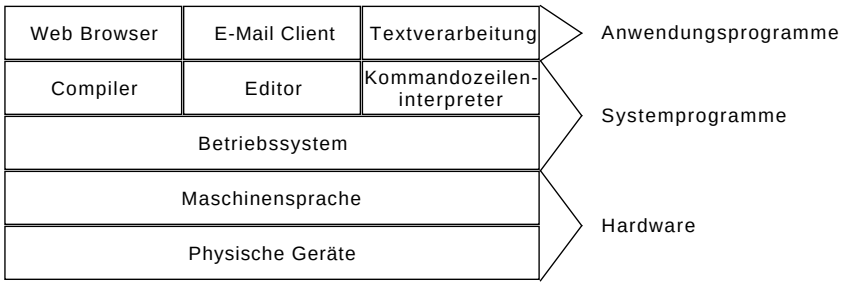


Abbildung 4.3.: Schichten eines Computersystems (nach [17, S. 13])

4.4.1. Konzept

Ziel der Modifikation des `gpsd`-Projekts ist, dass der Zeitgeber des ARTiS PC an Hand der GPS-Zeit möglichst genau angepasst wird. Dazu wird die Zeit über einen GPS-Empfänger bezogen und `gpsd` extrahiert aus dem Datenstrom relevante Informationen, darunter Zeit und Position. Diese Informationen werden in einen TCP-Socket geschrieben. Von dort können Anwendungen die GPS-Informationen im JSON-Format lesen. Zusätzlich wird die über GPS gewonnene Zeit um die Verarbeitungszeit korrigiert, welche mit Hilfe des PPS-Signals ermittelt werden kann (siehe Abschnitt 3.1.2 ab Seite 50). Die korrigierte Zeit wird dem `ntpd` übergeben, welcher den lokalen Zeitgeber der ARTiS PC mit der GPS-Zeit synchronisiert, ohne dass Zeitsprünge, zum Beispiel „Rückwärtslaufen“, beim lokalen Zeitgeber auftreten. Siehe Abbildung 4.4 für eine Visualisierung dieses Konzepts.

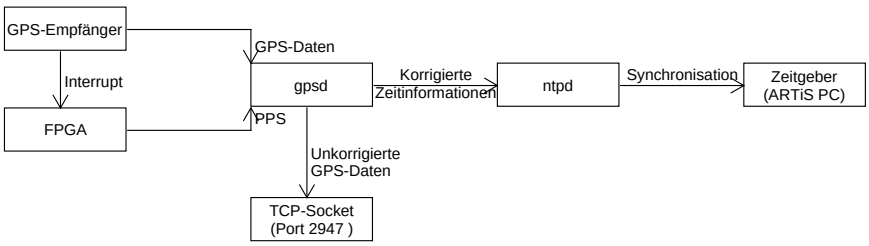


Abbildung 4.4.: Konzept für Modifikation des gpsd

4.4.2. Umsetzung

Die Programme aus Tabelle 4.2 dienen als Grundlage für das Konzept aus dem vorhergehenden Abschnitt. `gpsd` wird aus dem offiziellen Git-Repository unter `git://git.savannah.nongnu.org/gpsd.git` bezogen. `ntpd` wird aus den Debian-Software-Repositories installiert.

Tabelle 4.2.: Verwendete Software

Programm	**Version**
gpsd	3.11
ntpd	4.2.6

Eine Anpassung des `gpsd`-Projekts ist notwendig, da beim ARTiS PC das PPS-Signal, mit welchem die Verarbeitungszeit bestimmt werden kann, nicht direkt über einen Interrupt detektiert werden kann, sondern über ein FPGA zugänglich gemacht wird, vergleiche hierzu die Beschreibung der Hardware ab Seite 50. Die gründliche Analyse des `gpsd`-Projekts in Abschnitt 2.7.1 ab Seite 44 erleichtert die Modifikation des bestehenden `gpsd`-Codes. Die Schichten Packet Sniffer, Driver und Core Library müssen nicht angepasst werden. Einzig die Exporter-Funktion im Multiplexer, die in den Shared Memory von `ntpd` schreibt, bedarf einer Modifikation. Die unveränderten GPS-Informationen des Empfängers können weiterhin über den TCP-Socket gelesen werden. Die korrigierten Zeitinformationen werden von `ntpd` genutzt, um den lokalen Zeitgeber anzupassen. Um dies zu ermöglichen, sind zwei Aufgaben zu erledigen. Der `gpsd` muss angepasst werden und `ntpd` muss so konfiguriert werden, dass Informationen aus dem Shared-Memory-Segment von `gpsd` gelesen werden.

Anpassung des gpsd-Quelltexts

Um die Modularität von `gpsd` zu wahren, wird die PPS-Verarbeitung in eine eigene Datei gekapselt. Die Schnittstelle, um am ARTiS PC das PPS-Signal abzugreifen

ist in der Datei `esk_pps.h` definiert und enthält unter anderem die Methoden aus Klassendiagramm 4.5.

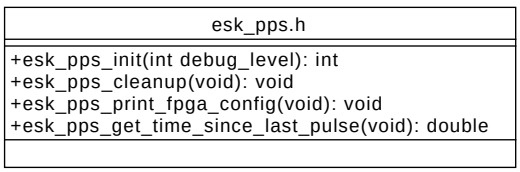


Abbildung 4.5.: Methoden in esk_pps.h

Die Implementierung der Methoden erfolgt in der Datei `esk_pps.c` in der Programmiersprache C. Im Folgenden werden die vier Methoden aus Klassendiagramm 4.5 vorgestellt:

- `esk_pps_init()`: Initialisiert das CTRL-Register des FPGAs. Dazu wird der 1-zu-64-Prescaler und die Interrupt-Erkennung an der PPS-Leitung des GPS-Empfängers aktiviert. Das Counter-Register des FPGAs ist 20 Bit breit, das heißt der Wertebereich umfasst die Werte von 0 bis 1.048.575. Ohne Prescaler beträgt die Taktrate des FPGAs 33 Mhz und das Counter-Register wird bei jedem Taktsignal um eins inkrementiert. Dies führt dazu, dass dieses Register bereits alle 0,03 s überläuft, siehe Gleichung 4.1.

$$\frac{1048575\ [\text{Ticks}]}{\frac{33 \cdot 10^6\ [\text{Ticks}]}{1\ \text{s}}} = 0,03\ \text{s} \tag{4.1}$$

  Durch den 1-zu-64-Prescaler erfolgt eine Inkrement nur bei jedem 64. Taktsignal und das Register läuft alle 2,03 s über.

- `esk_pps_cleanup()`: Macht die Einstellungen der Methode `esk_pps_init()` rückgängig.

- `esk_pps_print_fpga_config()`: Diese Methode gibt die aktuelle FPGA-Konfiguration, darunter zum Beispiel Taktrate, nach `stdout` aus.

- `esk_pps_get_time_since_last_pulse()`: Diese Methode liefert die Zeit seit dem Auftreten des PPS-Interrupts in Sekunden, indem Counter- und Capture-Register des FPGAs verglichen werden. Diese Methode muss Überläufe der beiden Register handhaben. Zudem stellt die Methode sicher, dass ein Aufruf-Intervall von 2,03 s eingehalten wird. Nur dann kann eine korrekte Behandlung der Register-Überläufe gewährleistet werden. Wird das Aufruf-Intervall nicht eingehalten, signalisiert die Methode dies durch einen negativen Rückgabewert.

Durch die Kapselung der PPS-Verarbeitung in eine eigene Datei fallen die Änderungen am bestehenden gpsd-Code minimal aus. Es muss ausschließlich der Daemon in `gpsd.c` erweitert werden. In der Methode `main` findet die Initialisierung des Daemons statt. In dieser Methode erfolgt auch die Initialisierung des FPGAs durch die Methode `esk_pps_init()`. Beendet sich der Daemon werden alle Ressourcen am Ende der `main`-Methode freigegeben. Dort erfolgt auch das Zurücksetzen des FPGAs mittels `esk_pps_cleanup()`. Die statische Methode `all_reports()` enthält die Exporter-Funktionalität des Daemons. An dieser Stelle werden die extrahierten GPS-Daten in den TCP-Socket und den Shared Memory von `ntpd` geschrieben. Vor dem Schreibvorgang in den Shared Memory wird mittels `esk_pps_get_time_since_last_pulse()` die Zeit seit Auftreten des PPS ermittelt und die Zeitinformationen laut GPS korrigiert, vergleiche Listing 4.5.

Listing 4.5: Zeitkorrektur durch PPS-Informationen

```
1  static void all_reports(struct gps_device_t *device, gps_mask_t
       changed)
2  {
3    ...
4    double fix_time;
5    double time_since_last_pulse;
6    ...
7    fix_time = device->newdata.time;
8    ...
9    time_since_last_pulse = esk_pps_get_time_since_last_pulse();
10   if (time_since_last_pulse >= 0)
11     fix_time += time_since_last_pulse;
12   else
13   ...
14   (void)ntpshm_put(device, device->shmIndex, &td);
15   ...
16 }
```

Die starke Kapselung erlaubt dem ESK, dass zukünftig auch aktualisierte Versionen des gpsd eingesetzt werden können ohne eine Vielzahl an Änderungen einpflegen zu müssen. Tabelle 4.3 fasst alle Modifikationen des gpsd-Projekts zusammen.

Tabelle 4.3.: Code-Statistik zu gpsd-Modifikationen

Sprache	**Dateien**	**Leer-zeilen**	**Kommentar-zeilen**	**Code**
C	1	44	86	214
C/C++ Header	1	11	73	20
Summe	2	55	159	234

Konfiguration von ntpd

Um den lokalen Zeitgeber der ARTiS PC an Hand der GPS-Zeit anzupassen, muss `ntpd` installiert und konfiguriert werden. Standardmäßig berücksichtigt `ntpd` nur Zeiten die von NTP-Servern stammen. Aus diesem Grund muss die Konfigurationsdatei `/etc/ntp.conf` angepasst werden. Innerhalb der Konfigurationsdatei beginnen Kommentare mit #, Leerzeichen werden ignoriert und die Konfiguration erfolgt durch Kommandos gefolgt von Argumenten für das jeweilige Kommando, vergleiche hierzu Listing 4.6.

Listing 4.6: ntp.conf

```
1 pool de.pool.ntp.org
2
3 driftfile /var/lib/ntp/ntp.drift
4 logfile /var/log/ntp.log
5
6 # GPS
7 server 127.127.28.0 prefer
8 fudge 127.127.28.0 refid GPS
```

Die erste Zeile definiert einen Pool von NTP-Servern im deutschen Raum als Zeitreferenz. Die dritte Zeile gibt eine Drift-Datei an. Diese enthält die Frequenzabweichung des lokalen Zeitgebers zur Referenzquelle in „Parts Per Million“. Dies beschleunigt die Zeitsynchronisation des lokalen Zeitgebers nach einem `ntpd`-Neustart, da die Frequenzabweichung nicht mehr neu ermittelt werden muss. Durch das Kommando `server` in Zeile sieben wird eine weitere Zeitreferenz angegeben, welches durch das Schlüsselwort `prefer` Vorzug vor anderen Zeitreferenzen erhält. Die Pseudo-IP-Adresse `127.127.28.0` referenziert dabei das von `gpsd` genutzte Shared-Memory-Segment. Das Kommando `fudge` in Zeile acht erlaubt es weitere Informationen zu einer Referenzquelle anzugeben. Das Argument `refid` gibt zum Beispiel einen Namen für die Pseudo-IP-Adresse an. Debugging-Werkzeuge wie `ntpq` zeigen dann zum Beispiel den Namen an Stelle der Pseudo-IP-Adresse an.

4.4.3. Evaluierung

Für die Evaluierung der Implementierung wird die Zeitgenauigkeit und die CPU-/RAM-Auslastung betrachtet. Um die Zeitgenauigkeit bewerten zu können, wird auf dem ARTiS PC `ntpd` so eingerichtet, dass der lokale Zeitgeber mit einem NTP-Server aus dem Internet synchron läuft. Dies stellt die Referenzzeit dar und diese wird mit den Zeitinformationen aus den GPS-Nachrichten verglichen. Die Messungen erfolgen bei einer Update-Rate von 1 Hz über 120 Sekunden. Die erste Messung erfolgt mit dem originalen `gpsd`-Code, wobei das PPS-Signal nicht berücksichtigt wird. Die zweite Messung beinhaltet die Änderungen aus Abschnitt 4.4.2. Während

den Messungen wird CPU- und RAM-Auslastung des `gpsd`-Prozesses durch das Programm `top` protokolliert, welches in 0,5-s-Intervallen die Auslastung prüft. Durch das PPS-Signal wird die Verarbeitungszeit seit Eintreffen der GPS-Daten an der Antenne des GPS-Empfängers gemessen. Es ist zu erwarten, dass die empfangene Zeit laut GPS um diese Verarbeitungszeit von der Referenzzeit abweicht. Die folgenden Abbildungen 4.6 und 4.7 fassen die Messergebnisse zusammen.

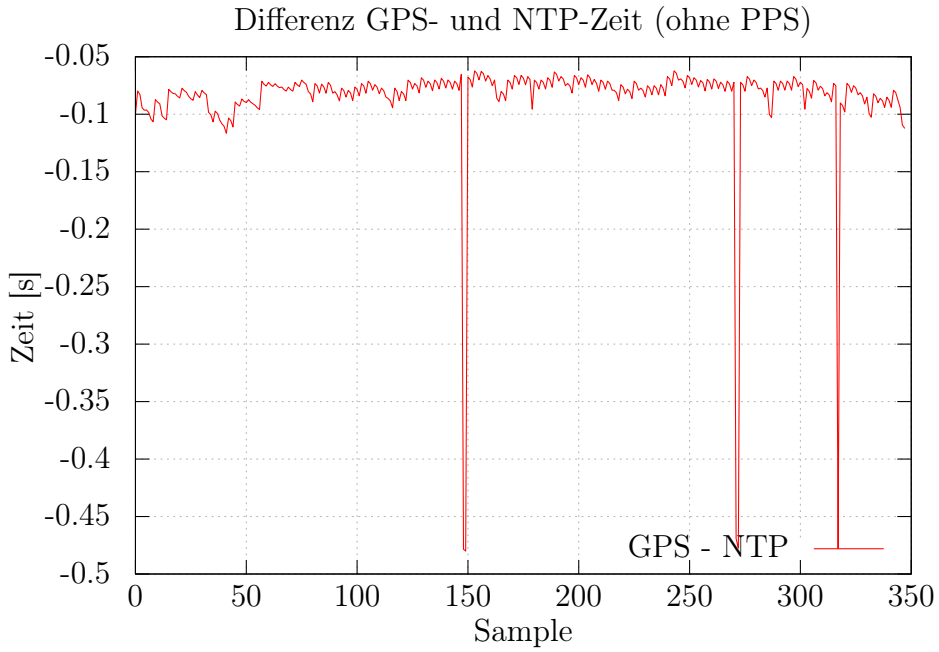


Abbildung 4.6.: gpsd ohne PPS: Differenz GPS- und NTP-Zeit bei 1 Hz Update-Rate über 120 s

Das Messergebnis stimmt mit der Erwartung überein. Die Differenz aus GPS- und NTP-Zeit ist negativ. Zuerst treffen die GPS-Daten an der Antenne das GPS-Empfängers ein, dann folgt eine Verzögerung und zu einem späteren Zeitpunkt werden die Daten von der CPU im ARTiS PC verarbeitet. Die durchschnittliche Zeit seit Auftreten des Pulses beträgt 0,085 s. Bei der Messung zeigten sich drei Ausreißer. Dort zeigte sich eine Verarbeitungszeit von rund 0,45 s. Weitere Messungen zeigten ebenfalls sporadische Ausreißer. Diese könnten sich dadurch erklären, dass die CPU nicht rechtzeitig den `gpsd`-Prozess bearbeitete.

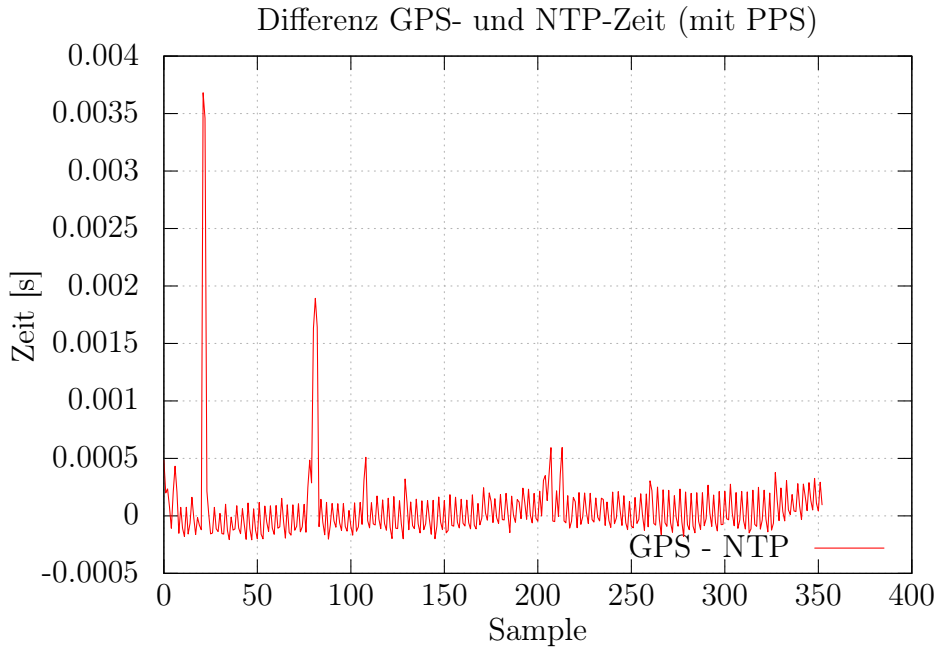


Abbildung 4.7.: gpsd mit PPS: Differenz GPS- und NTP-Zeit bei 1 Hz Update-Rate über 120 s

Wird das PPS-Signal berücksichtigt, ergeben sich deutlich geringere Abweichungen von der angenommenen Referenzzeit. Dies deutet auf eine korrekte Implementierung hin. Jedoch ist anzumerken, dass die angenommene Referenzzeit auf NTP beruht und deshalb auch fehlerbehaftet ist. Tabelle 4.4 liefert eine Statistik zu den obigen Messungen.

Tabelle 4.4.: gpsd ohne/mit PPS: Statistik Zeitinformationen

Abb.	**Erwartungswert**	**Standardabweichung**	**Min**	**Max**
4.6	-0.085251222	0.048296539	-0.480050087	-0.062249899
4.7	6.05718E-05	0.000343651	-0.000210047	0.003679991

Bei der CPU- und RAM-Auslastung ohne und mit PPS zeigten sich nur geringe Unterschiede, welche im Rahmen der Messungenauigkeit liegen. Die Verarbeitung des PPS-Signals im `gpsd`-Code verursacht somit keine deutliche Mehrbelastung der CPU und des RAMs, vergleiche Tabelle 4.5.

Tabelle 4.5.: gpsd ohne/mit PPS: Statistik CPU-/RAM-Auslastung in [%]

		Erwartungswert	Standardabweichung	Min	Max
Ohne PPS	**CPU**	0.707758621	1.054886075	0	3.9
	RAM	0.59137931	0.06508478	0.1	0.6
Mit PPS	**CPU**	0.685714286	1.033932637	0	3.9
	RAM	0.593506494	0.056609077	0.1	0.6

Für den zeitlichen Verlauf des CPU-/RAM-Verbrauchs während der Messung mit PPS siehe Abbildung 4.8. Der Ressourcenverbrauch wird von `top` in 0,5-s-Sekunden-Intervallen ermittelt. An vielen Abtastpunkten hat `gpsd` vom Betriebssystem keine Rechenzeit erhalten. An diesen Abtastpunkten ergibt sich eine CPU-Belastung von 0%. Bekommt `gpsd` Rechenzeit zugesprochen, belastet es die CPU überwiegend mit rund 2%. In einigen Situationen ergibt sich eine Belastung von knapp 4%. Der RAM-Verbrauch ist über den Testzeitraum mit 0,6% konstant.

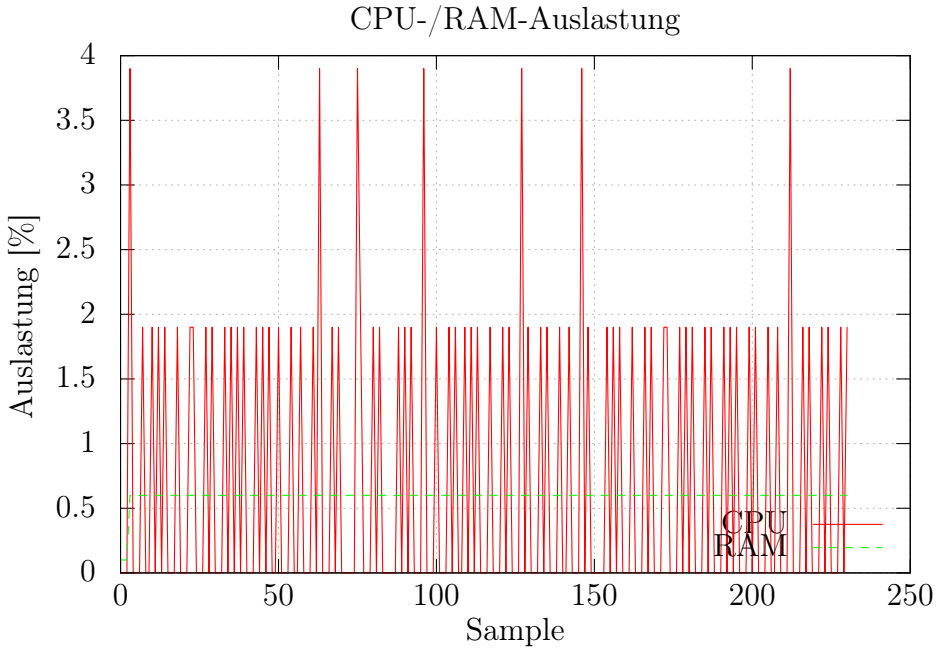


Abbildung 4.8.: gpsd mit PPS: CPU- und RAM-Auslastung bei 1 Hz Update-Rate über 120 s

Alle Messungen wurden mehrmals durchgeführt, wobei sich ein identisches Verhalten zu den oben beschriebenen Messungen zeigte. Langzeitmessungen wurden nicht durchgeführt.

4.5. Neuentwicklung des Linux-Kernel-Moduls ublox

Im ersten Schritt der Implementierung wurde auf Basis des `gpsd`-Codes eine Verzögerung von rund 85 ms ermittelt. Diese Verzögerung tritt zwischen Eintreffen von GPS-Daten an der Antenne des GPS-Empfängers und der Verarbeitung in der CPU im ARTiS PC auf. Ziel der folgenden Implementierung ist es diese Zeit zu reduzieren. Durch Nutzung des PPS-Signals kann im `gpsd`-Code die Zeitinformation korrigiert werden, nicht jedoch die Positionsinformation. Das Institut erhofft sich zudem durch eine maßgeschneiderte Lösung einen geringeren Wartungsaufwand im Vergleich zum Einsatz des `gpsd`-Codes.

4.5.1. Konzept

Das ESK möchte vollständige Kontrolle über die genutzte Hardware — und damit auch über den GPS-Empfänger —, um Verzögerungen, wie oben beschrieben, besser beherrschen zu können. Um Hardware steuern zu können, braucht es einen Treiber. Ziel der folgenden Implementierung ist es daher, einen Treiber zu entwickeln, der kontinuierlich Zeit- und Positionsinformationen liefert und auch Informationen zum PPS-Signal bietet. Gemäß Unix-Philosophie „everything is a file“ werden diese Informationen über eine Gerätedatei im Dateisystem unter `/dev` zugänglich gemacht. Für die Implementierung wird der Name `/dev/ublox` gewählt. Der Linux-Treiber implementiert die Aufrufe, die für den GPS-Empfänger möglich sind, darunter zum Beispiel `read()`. Ein `read()`-Systemaufruf liefert die aktuellen Zeit- und Positionsinformationen vom GPS-Empfänger. Die Zeit seit dem letzten Puls kann durch einen `ioctl()`-Aufruf abgefragt werden. Während `read()` nur erlaubt Daten vom Kernel in den User-Space zu schreiben und `write()` umgekehrt, erlaubt ein `ioctl()`-Aufruf Daten in beide Richtungen auszutauschen. `write()`-Aufrufe und damit eine Konfigurationsmöglichkeit des GPS-Empfängers durch UBX-Nachrichten werden in diesem Implementierungsschritt nicht berücksichtigt. Bereits bei der Inbetriebnahme des GPS-Empfängers kam es beim schreibenden Zugriff auf den Empfänger mit dem existierenden Linux-Treiber `cdc_acm`[1] zu Problemen, siehe Abschnitt 4.2.1 auf Seite 55. Abbildung 4.9 visualisiert das beschriebene Konzept.

[1] Dieser Treiber verwaltet unter Linux den Zugriff auf USB-Geräte, die sich mit der USB-Klasse `Abstract Control Model` identifizieren, so zum Beispiel Modems und der u-blox GPS-Empfänger.

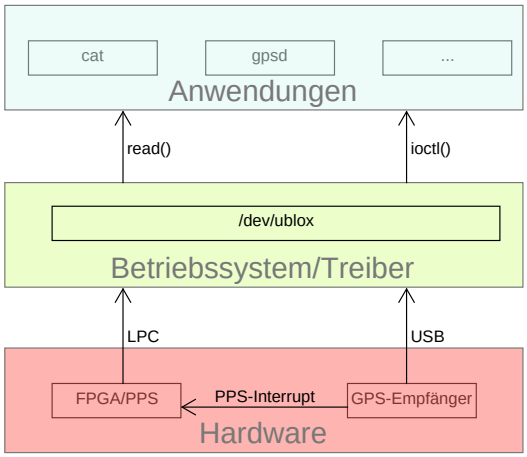


Abbildung 4.9.: Konzept des Linux-Treibers

Eine alternative Implementierung ist auch als User-Space-Treiber möglich. Zugriff auf den GPS-Empfänger ist zum Beispiel durch Nutzung von `libusb`[1] möglich und auch Lesen und Schreiben des LPC-Busses kann durch Operationen wie `inb()` und `outb()` im User-Space erreicht werden. Dadurch ist auch eine Code-Einsparung und damit geringerer Wartungsaufwand gegenüber dem `gpsd`-Projekt möglich, jedoch würde dies einer Kopie des `gpsd`-Projekts gleichkommen. Eine Treiber-Implementierung im Kernel-Space vermeidet im Vergleich zur User-Space-Implementierung zudem viele Kontextwechsel zwischen User- und Kernel-Space und trägt zu einer Reduzierung von Latenzen bei. Durch die Implementierung als Kernel-Space-Treiber, können GPS-Daten auch weiterhin von `gpsd` genutzt werden. Ein `read()`-Aufruf liefert die originalen GPS-Daten des Empfängers zurück. Der `gpsd` und seine Debugging-Werkzeuge können weiterhin verwendet werden.

4.5.2. Umsetzung

Der Linux-Kernel hat einen modularen Aufbau. Viele Komponenten, darunter auch Treiber, lassen sich als Module — das heißt separate Objektdateien — implementieren und entweder fest in das Kernel-Image integrieren oder im laufenden Betrieb durch Werkzeuge wie `insmod` oder `modprobe` nachladen. Während des Bauvorgangs des Kernels lassen sich Module fest in das Kernel-Image integrieren. Das zu entwickelnde Kernel-Modul trägt den Namen `ublox` und wird im ersten Entwicklungsschritt nicht fest in das Kernel-Image integriert. Wie in Abbildung 4.9 zu sehen, hat das Kernel-Modul zwei Schnittstellen. Eine zur Hardware und eine zur Anwendungsschicht.

[1] `http://www.libusb.org/`

Schnittstelle zur Hardware

Über USB wird auf den GPS-Empfänger zugegriffen und über den LPC-Bus auf das FPGA, um das PPS-Signal auswerten zu können. Im Folgenden wird genauer erläutert wie in Linux der Zugriff auf beide Bussysteme ermöglicht wird.

Linux stellt ein USB-Subsystem bereit, welches den Zugriff auf die eigentliche USB-Hardware bewerkstelligt. Ein USB-Treiber in Linux registriert sich mit der Methode `usb_register(struct usb_driver)` am USB-Subsystem. In der Struktur `struct usb_driver` wird unter anderem angegeben, welche Geräte ein USB-Treiber verwalten möchte. Dies geschieht zum Beispiel durch Angabe von Vendor Id und Product Id eines USB-Geräts. Diese Ids werden vom USB-Subsystem am Gerät abgefragt, wenn es mit dem System verbunden wird. Für die eigentliche Kommunikation mit einem USB-Gerät stellt das USB-Subsystem die asynchrone Methode `usb_submit_urb(struct urb *urb, ...)` bereit. Die Struktur `struct urb` (urb = USB Request Block) beschreibt dabei alle Parameter, die für eine USB-Transaktion notwendig sind.

Der Zugriff auf die FPGA-Register, die über den LPC-Bus angebunden sind, erfolgt über Methoden wie `ioperm()`, `inb()` und `outb()`. `ioperm()` reserviert für den aufrufenden Thread den Zugriff auf einen bestimmten Adressbereich. Durch `inb()` wird lesend auf eine Adresse zugegriffen, durch `outb()` schreibend.

Schnittstelle zur Anwendungsschicht

Der Anwendungsschicht sollen Zeit-, Positions- und PPS-Informationen zugänglich gemacht werden. Der GPS-Empfänger soll letztendlich so konfiguriert werden, dass er beliebige UBX-Nachrichten an den ARTiS PC sendet. Um die Schnittstelle zur Anwendung einfach zu halten, werden Zeit und Position in Form der binären UBX-Nachricht NAV-SOL[1] zur Verfügung gestellt. Diese Nachricht enthält Zeit, Position, Geschwindigkeit und weitere Angaben zur Genauigkeit dieser Informationen. Die UBX-Nachricht NAV-SOL enthält damit alle geforderten Informationen und kann zum Beispiel durch „Strukturüberlagerung“ geparst werden. Die Verarbeitung der Nachricht obliegt der Anwendungsschicht. Aufgabe des Treibers ist es den Datenstrom des GPS-Empfängers nach NAV-SOL-Nachrichten zu durchsuchen. Wird eine solche Nachricht detektiert, kann durch Auswertung des PPS-Signals der Zeitpunkt ermittelt werden, wann diese Nachricht an der Antenne des GPS-Empfängers ankam. Über folgende Systemaufrufe können die Daten von Anwendungen abgerufen werden:

- `read()`: Liefert die zuletzt empfangene NAV-SOL-Nachricht zurück.

[1] Vollständige Beschreibung dieser Nachricht findet sich unter [20, S. 163].

- `ioctl()`: Der Aufrufer wählt über einen Parameter welche Informationen gewünscht sind. Folgende Argumente werden angeboten:
  - `READ_PPS`: Liefert die Zeit seit Auftreten des Pulses. Die Zeit wird in Ticks zurückgegeben und nicht in Sekunden wie bei der vorher beschriebenen Methode `esk_pps_get_time_since_last_pulse()` unter 4.4.2 auf Seite 62. Dies ist dem Umstand geschuldet, dass im Linux-Kernel standardmäßig keine Floating-Point-Operationen zulässig sind. Anwendungen können die Ticks durch die Formel $\frac{\text{Ticks}}{f_{\text{FPGA}}}$ in Sekunden umrechnen. Das FPGA wird mit einer Frequenz von 33 Mhz und einem 1-zu-64-Prescaler betrieben. Daraus folgen 515.625 Ticks pro Sekunde ($\frac{33\text{ Mhz}}{64} = 515625$ Hz). Durch $\frac{\text{Ticks}}{515625}$ lässt sich die Zeit seit Auftreten des Pulses in Sekunden ermitteln.
  - `READ_GPS`: Liefert die selben Informationen wie `read()`.
  - `READ_GPS_AND_PPS`: Liefert die aktuelle NAV-SOL-Nachricht und PPS-Informationen in einem einzelnen Aufruf zurück.

Vereinigung der beiden Schnittstellen im Modul ublox

Die Schnittstellen zur Hardware und zur Anwenderschicht werden im Kernel-Modul `ublox` vereint. Die Daten Zeit, Position und PPS sollen dem Anwender schnellstmöglich zur Verfügung gestellt werden. Würde die USB-Kommunikation mit dem GPS-Empfänger erst initiiert, wenn eine Anwendung einen `read()`-Systemaufruf absetzt, so wäre eine zusätzliche Verzögerung vorhanden. Aus diesem Grund nutzt das Kernel-Modul `ublox` einen Thread, der in einem definierbaren Intervall den GPS-Empfänger über USB pollt. Das grundsätzliche Verhalten des Kernel-Moduls beschreibt der Pseudocode aus Listing 4.7.

Listing 4.7: Pseudocode Kernel-Modul

```
1 while (module_is_loaded && no_error) {
2   poll_gps_receiver_via_usb();
3   parse_data();
4   filter_data_for_nav_sol();
5   sleep();
6 }
```

Um die Aufgabe aus Listing 4.7 erledigen zu können, ist das Modul `ublox` modular aufgebaut. Abbildung 4.10 visualisiert die einzelnen Dateien des Kernel-Moduls und deren Abhängigkeiten.

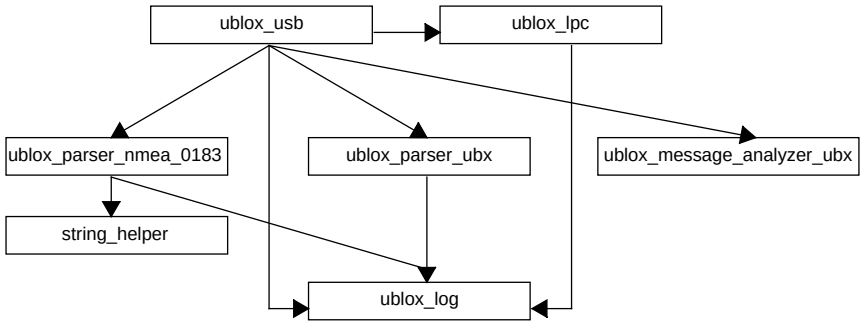


Abbildung 4.10.: Dateien des Kernel-Moduls ublox

Die Dateien werden im Folgenden nach dem Bottom-Up-Ansatz näher erläutert.

- `ublox_log`: In dieser Datei wird ein C-Makro definiert, um Nachrichten mit beliebigem Log-Level in den Kernel-Log-Buffer[1] zu schreiben. Das Makro ist nur aktiv, wenn beim Kompilieren des Moduls das Symbol „DEBUG“ definiert ist. Andernfalls wird das Makro leer ersetzt. Dadurch lassen sich alle Meldungen des Kernel-Moduls unterdrücken und so Verzögerungen durch `printk()`-Ausgaben vermeiden. Dies ist notwendig, um Laufzeitmessungen nicht zu verfälschen.

- `string_helper`: Zu Beginn der Masterarbeit konnte der GPS-Empfänger nur so konfiguriert werden, dass NMEA-Nachrichten gesendet werden. Aus diesem Grund konnte anfangs nur mit NMEA-Nachrichten gearbeitet werden. `string_helper` enthält Hilfsmethoden, um die ASCII-basierten NMEA-Nachrichten zu verarbeiten, zum Beispiel um den Anfang einer NMEA-Nachricht zu detektieren.

- `ublox_parser_nmea_0183`: Der Anwendungsschicht sollen nur vollständige GPS-Informationen aus Zeit und Position zur Verfügung gestellt werden. Der GPS-Empfänger sendet in einer USB-Transaktion maximal 64 Byte an Daten, da der Ausgabepuffer 64 Byte groß ist. Viele NMEA-Sätze und auch UBX-Pakete überschreiten jedoch 64 Byte. Das heißt eine Nachricht muss in mehreren Transaktionen übermittelt werden. Es ist Aufgabe des Parsers die unvollständigen NMEA-Nachrichten zu Puffern und zu melden, wenn eine vollständige Nachricht empfangen wurde.

- `ublox_parser_ubx`: Wie bei `ublox_parser_nmea_0183` werden unvollständige Nachrichten gepuffert und vollständige Nachrichten gemeldet. Dieser Parser verarbeitet nur UBX-Datenströme.

[1] Dieser Ring-Puffer kann zum Beispiel durch das Programm `dmesg` ausgelesen werden.

- `ublox_message_analyzer_ubx`: Der Anwendungsschicht soll nur die letzte UBX-Nachricht vom Typ NAV-SOL zur Verfügung gestellt werden. Diese Datei bietet Methoden, um Typ und weitere Merkmale einer UBX-Nachricht zu bestimmen.

- `ublox_lpc`: Diese Datei kapselt die Auswertung des PPS-Signals, welches über den LPC-Bus abgefragt werden kann.

- `ublox_usb`: In dieser Datei werden mehrere Aufgaben erfüllt. Es wird der Treiber im Linux-USB-Subsystem registriert. Nach erfolgreicher Registrierung wird die Gerätedatei `/dev/ublox` angelegt. Bei der Registrierung wird durch eine Struktur `file_operations` angegeben, welche Operationen auf die Gerätedatei möglich sind, darunter `read()` und `ioctl()`. Kern dieser Datei ist das kontinuierliche Pollen des GPS-Empfängers wie in Listing 4.7 dargestellt. Das Pollen wird durch einen eigenen Thread abgewickelt, der vom Linux-Kernel in Form einer `workqueue` aus `linux/workqueue.h` zur Verfügung gestellt wird. In diese `workqueue` wird in einem definierbaren Intervall eine Task zum Pollen des GPS-Empfängers eingehängt. Nachteil dieser Lösung durch `workqueues` ist, dass das Polling-Intervall nicht vollkommen frei wählbar ist, sondern an die Größe „Jiffy“ gebunden ist. Ein Jiffy beschreibt die Dauer zwischen zwei Timer-Interrupts in Linux und beträgt beim ARTiS PC 4 ms. Ein Polling-Intervall kann daher nur in 4-ms-Schritten justiert werden. Das Intervall kann beim Laden des Kernel-Moduls mittels `insmod` oder `modprobe` angegeben werden. Standardmäßig wird in 20-ms-Intervallen gepollt. Dies stellt einen Kompromiss aus Ressourcenverbrauch und notwendiger Aktualität der GPS-Daten dar. Die empfangenen und geparsten GPS-Daten werden vom Kernel-Modul zwischengespeichert und können durch `read()` und `ioctl()` abgerufen werden.

Für die Implementierung der Dateien `string_helper`, `ublox_parser_nmea_0183`, `ublox _parser_ubx` und `ublox_message_analyzer_ubx` wird folgendes Vorgehen gewählt, um ein fehlerfreies Verhalten des Codes sicherzustellen: Da auf Kernel-Ebene kein klassisches Unit-Testing-Framework existiert, wird der Code zuerst auf Anwendungsebene implementiert. Nach erfolgreichen Tests mit dem Unit-Testing-Framework `CUnit` wird der Code auf Kernel-Ebene portiert. Dieses Vorgehen ist hilfreich, da ein Fehlverhalten auf Kernel-Ebene fatale Folgen haben kann. Ein dereferenzierter `NULL`-Pointer auf Kernel-Ebene kann zum Beispiel einen sofortigen Absturz des Systems bedeuten.

Tabelle 4.6 bietet einen Überblick über den Quelltext des Kernel-Moduls `ublox`. Die Skripte in der Sprache „Bourne Shell“ werden genutzt, um Daten für die in Abschnitt 4.5.3 erläuterten Messungen zu sammeln.

Tabelle 4.6.: Code-Statistik zur Neuentwicklung

Sprache	**Dateien**	**Leer-zeilen**	**Kommentar-zeilen**	**Code**
C	8	443	262	1568
Bourne Shell	11	88	141	226
C/C++ Header	8	54	359	126
make	2	9	3	27
Summe	29	594	765	1947

4.5.3. Evaluierung

Wie bei der Evaluierung der `gpsd`-Modifikation wird die Zeitgenauigkeit und die CPU-/RAM-Auslastung betrachtet, um einen Vergleich ziehen zu können. Dazu wird wieder die Zeit eines entfernten NTP-Servers als Referenz angenommen, wobei die selben Einschränkungen bezüglich Ungenauigkeit der NTP-Zeit gelten. Des Weiteren wird die Zeit seit dem Auftreten des Pulses — und damit die Verarbeitungszeit — genauer betrachtet. Dazu wird nach dem Empfang einer NAV-SOL-Nachricht das Counter- und Capture-Register des FGPAs verglichen. Zudem wird die Aufrufhäufigkeit untersucht, mit welcher der Kernel-Thread den GPS-Empfänger über USB pollt, um sicherzugehen, dass das gewünschte Polling-Intervall eingehalten wird. Dazu wird der Linux-Kernel mit Hilfe der `ftrace`-Funktionalität instrumentalisiert, um die Prozesszeiten eines Threads zu protokollieren. Eine ausführliche Erklärung zu diesem Function Tracer gibt die offizielle Dokumentation unter `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`. Der Vorteil durch die Kernel-Instrumentalisierung liegt darin, dass der `ublox`-Code nicht verändert werden muss. Alle Messungen werden wie beim `gpsd` bei einer Update-Rate von 1 Hz mit einer Laufzeit von 120 Sekunden durchgeführt. Die CPU-/RAM-Auslastung wird wieder mit `top` in 0,5-s-Intervallen protokolliert.

Zeitgenauigkeit und CPU-/RAM-Auslastung

Es ist zu erwarten, dass ohne PPS-Korrektur die Differenz aus GPS- und NTP-Zeit negativ ist. Zudem sollte sich die Zeit seit Auftreten des Pulses reduzieren und maximal die Länge des gewählten Polling-Intervalls von 20 ms aufweisen.

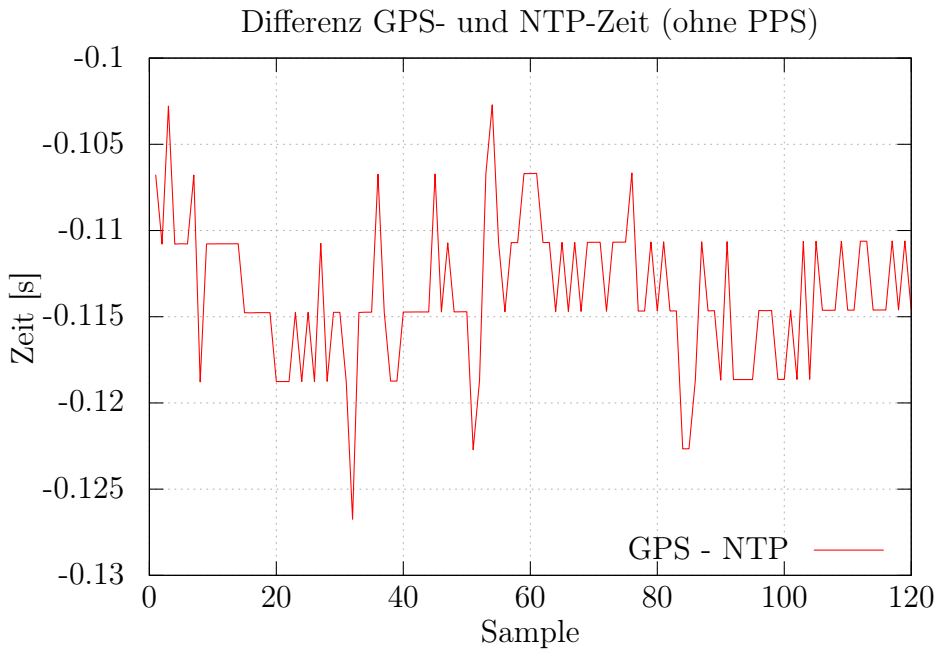


Abbildung 4.11.: ublox ohne PPS: Differenz GPS- und NTP-Zeit bei 1 Hz Update-Rate über 120 s

Gegenüber den Messungen mit `gpsd` fällt auf, dass im gleichen Zeitraum deutlich weniger Samples erzeugt wurden (120 gegenüber 350). Dies erklärt sich dadurch, dass das Kernel-Modul ausschließlich das UBX-Paket NAV-SOL berücksichtigt, und dies bei einer Update-Rate von 1 Hz nur einmal pro Sekunde vom GPS-Empfänger gesendet wird. `gpsd` hingegen berücksichtigt die vier UBX-Nachrichten NAV-SOL, NAV-DOP, NAV-TIMEGPS sowie NAV-SVINFO[1] und wertet deshalb im selben Zeitraum mehr UBX-Pakete aus. Die Erwartungen bezüglich der negativen Differenz aus GPS- und NTP-Zeit haben sich bestätigt. Jedoch hat sich die Zeit seit Auftreten des Pulses von durchschnittlich 85 ms auf durchschnittlich 114 ms erhöht. Diese Verschlechterung wird im folgenden Abschnitt 4.5.3 genauer untersucht.

[1] Vergleiche Quelltext `driver_ubx.c` Zeile 59 ff.

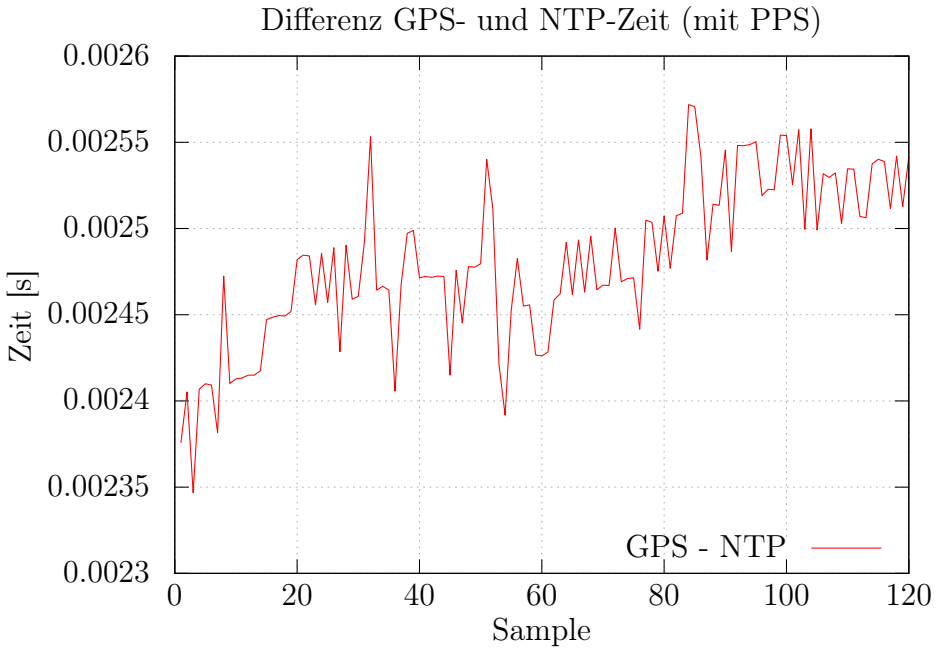


Abbildung 4.12.: ublox mit PPS: Differenz GPS- und NTP-Zeit bei 1 Hz Update-Rate über 120 s

Durch Berücksichtigung des PPS-Signals ergeben sich deutlich geringere Abweichungen zur angenommenen Referenzzeit. Verglichen mit der Messung des `gpsd`-Codes mit PPS hat sich die Abweichung zur angenommenen Referenzzeit von durchschnittlich 0,06 ms auf 2 ms verschlechtert. Da jedoch auch die angenommene Referenzzeit um mehrere Zehntel Millisekunden von der exakten Zeit abweichen kann, muss diese unter dem Gesichtspunkt der Messungenauigkeit betrachtet werden.

Tabelle 4.7.: ublox ohne/mit PPS: Statistik Zeitinformationen

Abb.	**Erwartungswert**	**Standardabweichung**	**Min**	**Max**
4.11	-0,1136979361	0,0040685243	-0,1267399788	-0,1027100086
4.12	0,0024808606	4,71309067332811E-005	0,0023527145	0,0025725365

Die CPU-/RAM-Auslastung im Kernel-Modul `ublox` wird nur bei aktiver PPS-Auswertung betrachtet. Es zeigt sich eine dreimal geringere CPU-Auslastung gegenüber dem `gpsd`-Code, welcher jedoch im selben Zeitraum auch mehr UBX-Pakete auswertet als das Kernel-Modul `ublox`. Das Programm `top` ist nicht in der Lage den RAM-Verbrauch eines Kernel-Moduls zu ermitteln und nimmt deshalb als RAM-Verbrauch 0% an. Tabelle 4.8 bietet eine Zusammenfassung der CPU- und RAM-Auslastung.

Tabelle 4.8.: ublox: Statistik CPU-/RAM-Auslastung in [%]

	Erwartungswert	Standardabweichung	Min	Max
CPU	0,2177966102	0,6307288434	0	3,8
RAM	0	0	0	0

Die geringere CPU-Auslastung gegenüber `gpsd` zeigt sich auch im zeitlichen Verlauf in Abbildung 4.13.

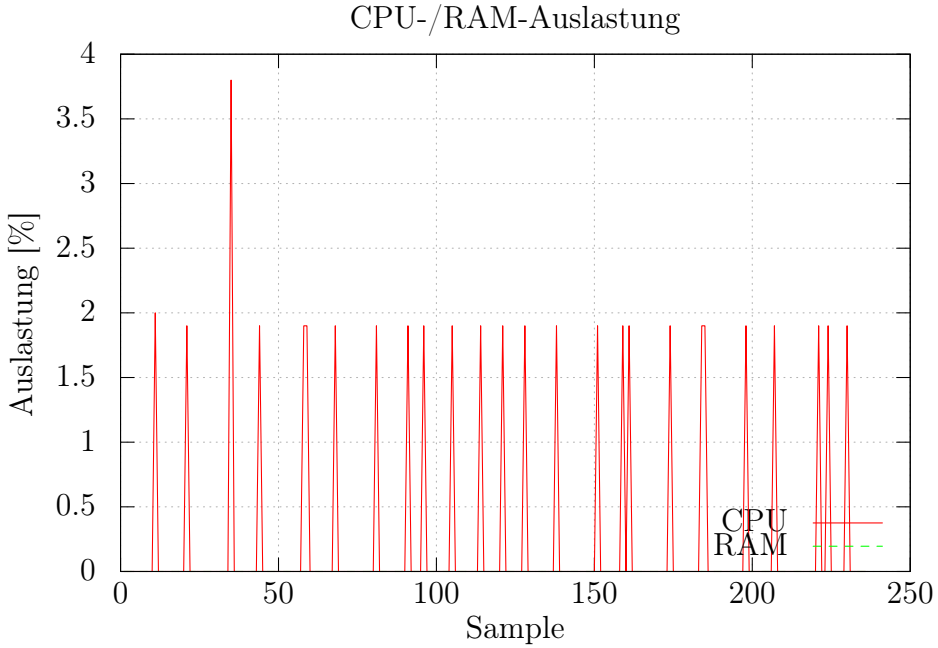


Abbildung 4.13.: ublox mit PPS: CPU- und RAM-Auslastung bei 1 Hz Update-Rate über 120 s

Auch mit dem Kernel-Modul `ublox` wurden die Messungen mehrmals durchgeführt mit jeweils ähnlichen Beobachtungen. Die Zeiten seit Auftreten des Pulses reichen von minimal 70 ms bis maximal 140 ms.

Zeit seit Puls und Einhaltung des Polling-Intervalls

Abbildung 4.14 visualisiert die Zeit seit Auftreten des Pulses während des Testzeitraums von 120 s.

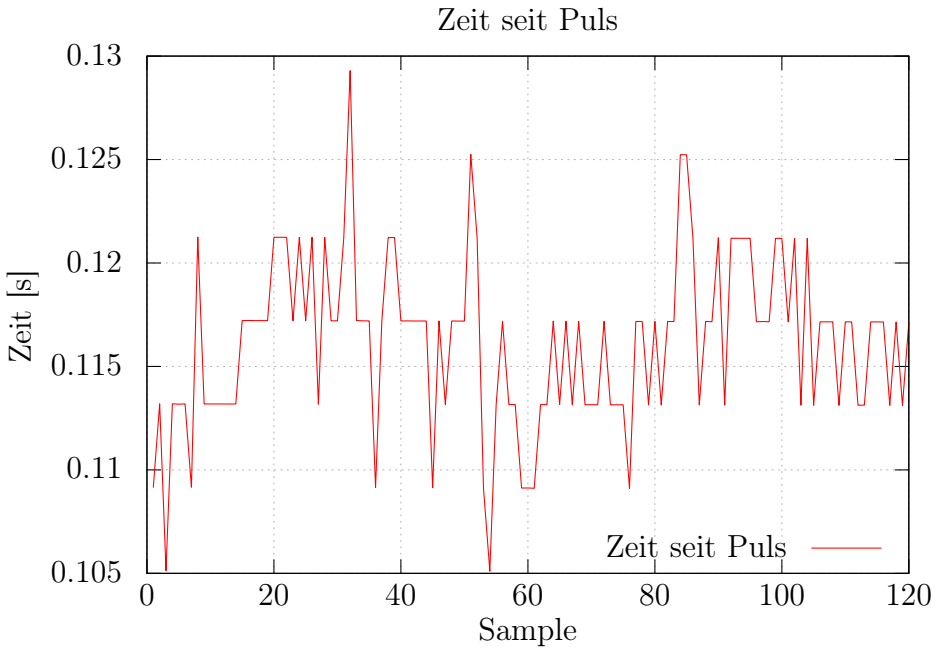


Abbildung 4.14.: ublox: Zeit seit Puls bei 1 Hz Update-Rate über 120 s

Erwartet wird, dass durch ein festes Polling-Intervall von 20 ms die Zeit seit Auftreten des Pulses auf maximal diesen Wert reduziert werden kann. Da diese Erwartung nicht erfüllt wird, muss untersucht werden, ob das Polling-Intervall überhaupt eingehalten wird. Abbildung 4.15 visualisiert die Differenz der Aufwachzeiten des Polling-Threads. Diese sollten konstant 20 ms Sekunden betragen.

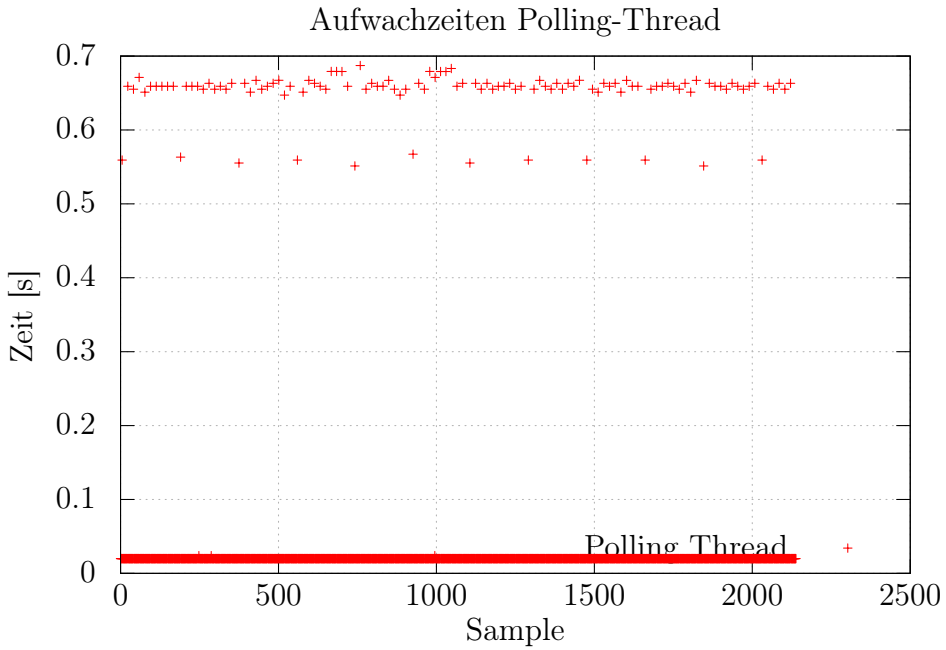


Abbildung 4.15.: ublox: Differenz Aufwachzeiten Polling-Thread bei 1 Hz Update-Rate über 120 s

Die überwiegende Zeit wird das Polling-Intervall von 20 ms eingehalten. Im Plot verbinden sich die unteren Punkte bereits zu einem Linienzug. Jedoch zeigen sich deutliche Ausreißer im Bereich von 0,6 s. Tabelle 4.9 zeigt die statistische Auswertung der Zahlen aus Abbildung 4.15. Sie zeigt, dass die durchschnittliche Aufwachzeit des Polling-Threads nicht bei 20 ms liegt, sondern bei rund 54 ms.

Tabelle 4.9.: ublox: Statistik Aufwachzeiten Polling-Thread

Erwartungswert	**Standardabweichung**	**Min**	**Max**
0,0541671996	0,142956724	0,019938	0,687165

4.6. Minimierung von Latenzen durch Scheduling-Strategie SCHED_FIFO

Die Messungen im vorhergehenden Schritt zeigen auf, dass ein Polling-Intervall von 20 ms im Kernel-Modul `ublox` nicht eingehalten wird. Dies kann verschiedene Ursachen haben. Neben der Software muss auch die Hardware als Problemquelle berücksichtigt werden. Der GPS-Empfänger verhält sich wie eine Blackbox, da nur für Microsoft Windows die Evaluations-Software „u-blox u-center“ zur Verfügung steht,

mit welcher das Verhalten des Empfängers genauer analysiert und verändert werden kann. Beim GPS-Empfänger kann daher nur die Standardkonfiguration, die in der Empfänger- und Protokollbeschreibung [20] beschrieben ist, angenommen werden. So zum Beispiel eine Update-Rate von 1 Hz. Um die Software als Problemquelle auszuschließen, wird im Folgenden die Scheduling-Strategie so anpasst, dass sichergestellt wird, dass der Polling-Thread ausreichend Rechenzeit vom Betriebssystem zugeteilt bekommt.

4.6.1. Konzept

Bisher wird der Polling-Thread des Kernel-Moduls `ublox` mit der Standard-Scheduling-Strategie `SCHED_OTHER` betrieben. Das heißt dem Prozess wird keine Rechenzeit durch das Betriebssystem garantiert. Aus diesem Grund wird die Scheduling-Strategie im Code zur echtzeitfähigen Strategie `SCHED_FIFO` geändert. Zudem wird gleichzeitig der Linux-Kernel mit dem „RT-Patch“ gepatcht, für Details zum RT-Patch siehe Abschnitt 2.6.2 ab Seite 42. Dadurch werden selbst Hardware-Interrupts durch Kernel-Threads bearbeitet. Dies erlaubt für Testzwecke den Polling-Thread des Kernel-Moduls `ublox` mit einer maximalen Priorität von 99 laufen zu lassen. Dies garantiert, dass der Polling-Thread nicht von Hardware-Interrupts unterbrochen wird, welche beim RT-Patch von Threads mit der Priorität 50 bearbeitet werden[1]. Dies soll gewährleisten, dass ein Polling-Intervall von 20 ms eingehalten wird.

4.6.2. Umsetzung

Die Umsetzung des Konzepts besteht aus zwei Schritten, der Änderung des Quelltexts des Kernel-Moduls `ublox` und dem Einspielen eines RT-fähigen Linux-Kernels. Diese beiden Schritte werden im folgenden näher erläutert.

Änderung des Quelltextes

Der pollende Kernel-Thread arbeitet nach dem Prinzip aus Pseudocode 4.7 auf Seite 72. In einer Endlosschleife wird über den asynchronen Methodenaufruf `sub mit_urb()` der GPS-Empfänger gepollt. Der pollende Thread legt sich bis zur Antwort durch den Empfänger und auch nach der Verarbeitung der erhaltenen Daten schlafen und ermöglicht so anderen Prozessen zu rechnen. Diese Prinzip kann so beibehalten werden und daher wird ausschließlich die Scheduling-Strategie durch Aufruf von `sched_setscheduler()` zu `SCHED_FIFO` geändert. Diese Methode erwartet auch die Priorität, mit welcher der Thread ausgeführt wird. Diese kann beim Laden des Kernel-Moduls mitgegeben werden. Standardmäßig wird mit 99 die höchste Priorität verwendet.

[1] Einzig allein der Timer-Interrupt unter Linux läuft auch mit einer RT-Priorität von 99.

RT-fähiger Linux-Kernel durch RT-Patch

Eine umfassende Beschreibung, um den RT-Patch anzuwenden und dessen Funktionsfähigkeit zu prüfen, bietet die offizielle Dokumentation zum RT-Patch unter `https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO`. Die folgenden Abschnitte umreißen die vier wichtigsten Schritte:

1. Linux-Kernel-Quelltext und RT-Patch beziehen.
2. RT-Patch auf Quelltext anwenden.
3. Kernel konfigurieren.
4. Kernel kompilieren.

Der RT-Patch wird nur für bestimmte Versionsnummern des originalen Linux-Quelltexts gepflegt. Auf dem ARTiS PC wird Debian GNU/Linux mit Kernel-Version 3.2.0 betrieben. Für die Modifikation wird der Linux-Kernel in Version 3.2.60 genutzt. Der Linux-Quelltext[1] und der RT-Patch[2] finden sich auf den Seiten des „Linux Kernel Archives“.

Der RT-Patch wird als Textdatei zur Verfügung gestellt. Diese Textdatei enthält eine Beschreibung aller Änderungen, die am originalen Linux-Quelltext durchgeführt werden müssen. Die Änderungen lassen sich durch das Programm `patch` anwenden.

Die Konfiguration des Linux-Kernels erfolgt mit Hilfe des Programms `make <target>`. Dieses Programm nutzt das beim Linux-Kernel beiliegende `Makefile`, welches zahlreiche Ziele zur Konfiguration und für den eigentlichen Bauvorgang bereitstellt[3]. Durch die Konfiguration können zum Beispiel Treiber fest in das Kernel-Image integriert werden und Merkmale des Kernels aktiviert bzw. deaktiviert werden. Der Linux-Kernel bietet verschiedene Frontends an, um eine Konfiguration vorzunehmen. `make xconfig` startet zum Beispiel ein grafisches Konfigurationswerkzeug. `make menuconfig` hingegen startet eine Terminal-basierte Lösung. Damit die Änderungen des RT-Patches ordnungsgemäß funktionieren, müssen im Konfigurationsabschnitt „Processor type and features“ mindestends die Optionen `CONFIG_HIGH_RES_TIMERS` und `CONFIG_PREEMPT_RT_FULL` aktiviert werden.

Der Kompiliervorgang kann durch `make` angestoßen werden. Nachteil hierbei ist, dass nicht die Linux-Header-Dateien erzeugt werden, die notwendig sind, um ein Kernel-Modul wie `ublox` für diesen neuen Kernel zu bauen. Debian liefert aber mit

[1] `ftp://ftp.kernel.org/pub/linux/kernel/`
[2] `https://www.kernel.org/pub/linux/kernel/projects/rt/`
[3] Alle `make`-Targets finden sich unter `https://kernel.org/doc/makehelp.txt`.

dem Paket `kernel-package` das Programm `make-kpkg`. Dieses Programm erlaubt durch Aufruf von `make-kpkg -initrd kernel-image kernel-headers`, sowohl das eigentliche Kernel-Image, als auch die Header-Dateien auf Basis der vorhergehenden Konfiguration zu bauen. Image und Header-Dateien werden dadurch zu einem Debian-Paket mit der Endung `.deb` gepackt und können über eine Paketverwaltungs-Software wie `apt-get` installiert werden.

4.6.3. Evaluierung

Da der pollende Thread mit maximaler Priorität betrieben wird und gleichzeitig der RT-Patch genutzt wird, ist sichergestellt, dass der pollende Thread nicht durch Hardware-Interrupts unterbrochen wird. Daher wird nun erwartet, dass ein Polling-Intervall von 20 ms eingehalten werden kann. Das Messergebnis wird in Abbildung 4.16 visualisiert.

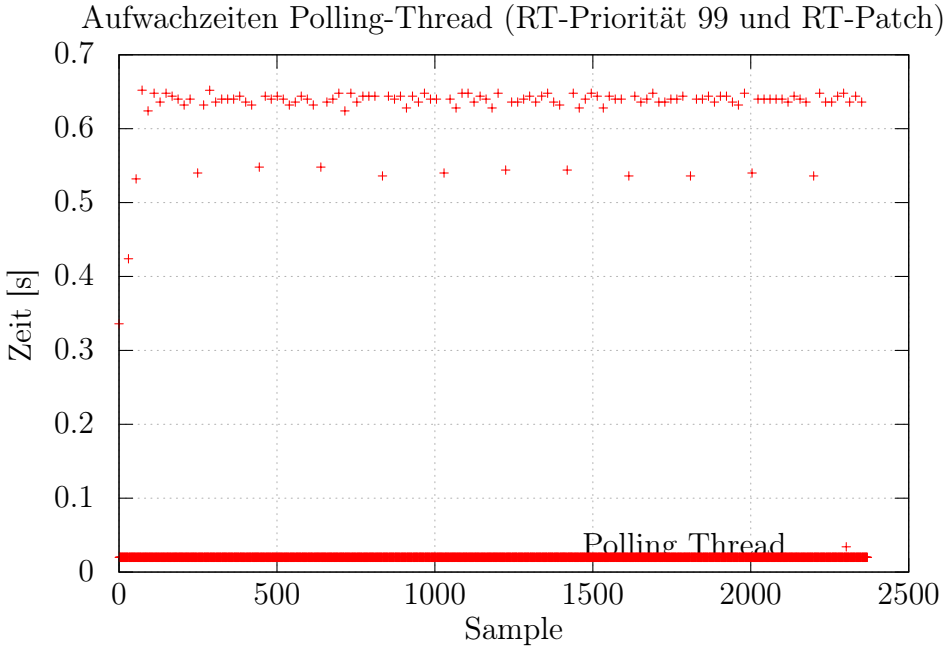


Abbildung 4.16.: ublox: Differenz Aufwachzeiten Polling-Thread bei 1 Hz Update-Rate über 120 s bei RT-Priorität 99 und RT-Patch

Die Aufwachzeiten zeigen, dass ein Polling-Intervall von 20 ms überwiegend eingehalten wird, aber trotz maximaler Priorität und RT-Patch immer wieder Ausreißer auftreten. Dies führt dazu, dass die durchschnittliche Aufwachzeit des pollenden Threads bei 51 ms Sekunden liegt, siehe Tabelle 4.10. Würde ein Intervall von 20 ms eingehalten, ist auch mit 6.000 Samples zu rechnen anstatt mit nur rund 2.500 ($\frac{1 \text{ Sample}}{20 \text{ ms}} \rightarrow \frac{50 \text{ Samples}}{s} \rightarrow \frac{50 \text{ Samples}}{s} \cdot 120 \text{ s} = 6000 \text{ Samples}$).

Tabelle 4.10.: ublox: Statistik Aufwachzeiten Polling-Thread bei RT-Priortät 99

Erwartungswert	**Standardabweichung**	**Min**	**Max**
0,0509327578	0,1337278597	0,019966	0,652059

Aus Softwaresicht sind Scheduling-Parameter nun so optimiert, dass auf Grund des Scheduling-Verhaltens ein Polling-Intervall von 20 ms eingehalten werden kann. Um zu verifizieren, dass die Software tatsächlich dieses Polling-Intervall einhält, wird für die folgende Messung die USB-Kommunikation mit dem GPS-Empfänger simuliert. Der Pseudocode 4.7 auf Seite 72 bezeichnet den Schritt für die USB-Kommunikation als `poll_gps_receiver_via_usb()`. Dieser Schritt wird durch ein `msleep(1)` ersetzt, was den pollenden Thread für mindestens 1 ms schlafen legt. 1 ms wird gewählt, da der GPS-Empfänger als USB-1.1-Gerät angebunden ist und damit eine maximale Übertragungsrate von 12 Mbit/s aufweist. Der Empfänger verfügt über einen Ausgabepuffer mit 64 Byte und kann daher bei einer Übertragung maximal 64 Byte senden. Für 64 Byte ergibt sich daher eine Sendezeit von 0,00004 s (64 Byte = 512 Bit $\rightarrow \frac{512 \text{ Bit}}{\frac{12 \text{ Mbit}}{\text{s}}} = 0,00004 \text{ s}$). Die minimale Schlafzeit für `msleep()` beträgt jedoch 1 ms. Nach dieser Wartezeit wird vom Parser ein vorgefertigtes UBX-Paket verarbeitet. Das Ergebnis dieser Simulation zeigt Abbildung 4.17.

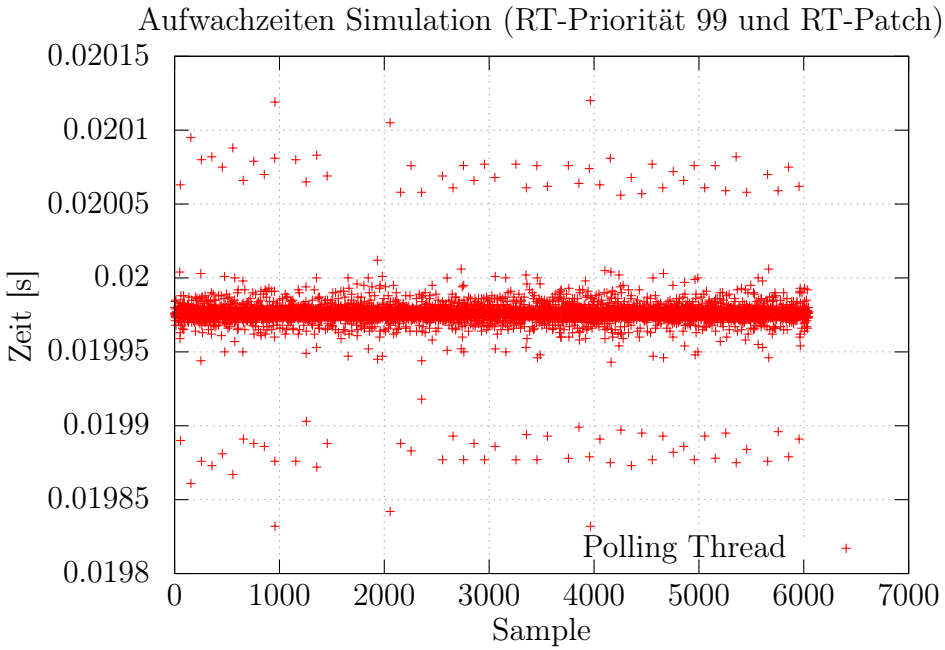


Abbildung 4.17.: ublox: Differenz Aufwachzeiten Simulation bei 1 Hz Update-Rate über 120 s bei RT-Priorität 99 und RT-Patch

Ein Polling-Intervall von 20 ms wird nun eingehalten wie auch die dazugehörige Statistik in Tabelle 4.11 zeigt.

Tabelle 4.11.: ublox: Statistik Simulation bei RT-Priortät 99

Erwartungswert	**Standardabweichung**	**Min**	**Max**
0,0199760382	1,35732274284577E-005	0,019832	0,02012

Um die Interaktion mit dem GPS-Empfänger genauer zu analysieren, werden Ausgaben in das Kernel-Modul `ublox` eingefügt und ausgewertet. Interessante Stellen sind hierbei die asynchrone Anfrage für eine USB-Kommunikation mit dem Empfänger durch `submit_urb()` und der darauf folgenden Antwort. Listing 4.8 beschreibt eine solche Debugging-Sitzung.

Listing 4.8: Debugging-Sitzung USB-Kommunikation 20-ms-Intervall

```
1  ...
2  [12434.384964] USB request
3  [12434.385934] USB response (transferred_bytes = 60)
4  [12434.404079] USB request
5  [12434.405936] USB response (transferred_bytes = 0)
6  [12434.405948] USB request
7  [12434.406932] USB response (transferred_bytes = 26)
8  [12434.424071] USB request
9  [12434.425936] USB response (transferred_bytes = 24)
10 [12434.444077] USB request
11 [12435.382950] USB response (transferred_bytes = 0)
12 [12435.382965] USB request
13 [12435.384090] USB response (transferred_bytes = 60)
14 [12435.404078] USB request
15 [12435.405931] USB response (transferred_bytes = 26)
16 [12435.424087] USB request
17 [12435.425930] USB response (transferred_bytes = 24)
18 [12435.444077] USB request
19 [12436.379935] USB response (transferred_bytes = 0)
20 ...
```

Die Zeitstempel[1] in der linken Spalten zeigen, dass überwiegend in 20-ms-Intervallen gepollt wird und rund 1 ms später die Daten vom GPS-Empfänger empfangen werden. Zeile 10/11 und Zeile 18/19 zeigen jedoch ein auffälliges Verhalten. Erst rund 1 Sekunden nach der asynchronen Anfrage trifft eine Antwort ein, wobei keinerlei GPS-Daten empfangen wurden. Dies spricht dafür, dass der GPS-Empfänger nicht schnell genug GPS-Daten sendet, um ein Polling-Intervall von 20 ms einhalten zu können. Wird beispielsweise das Polling-Intervall auf 100 ms erhöht, so kann dieses

[1] Zeit seit Start des Systems im Format <Sekunden.Mikrosekunden>.

Verhalten unterbunden werden. Dieses Intervall erfüllt die minimalen Anforderungen bezüglich Aktualität der Zeit- und Positionsinformationen. Jedoch muss bedacht werden, dass die Update-Rate des GPS-Empfängers zum Zeitpunkt der Messung 1 Hz betrug. Zudem kann durch ein festes Polling-Intervall von 100 ms die Zeit seit Auftreten des Pulses nicht systematisch reduziert werden und es werden Positionsdaten verarbeitet, die um maximal 100 ms Sekunden veraltet sind.

Listing 4.9: Debugging-Sitzung USB-Kommunikation 100-ms-Intervall

```
...
[20443.676090] USB request
[20443.677898] USB response (transferred_bytes = 55)
[20443.776088] USB request
[20443.777901] USB response (transferred_bytes = 63)
[20443.876074] USB request
[20443.877903] USB response (transferred_bytes = 63)
[20443.976089] USB request
[20443.977902] USB response (transferred_bytes = 33)
[20444.076112] USB request
[20444.077929] USB response (transferred_bytes = 34)
[20444.176069] USB request
[20444.177905] USB response (transferred_bytes = 44)
[20444.276088] USB request
[20444.277901] USB response (transferred_bytes = 33)
[20444.376146] USB request
[20444.377906] USB response (transferred_bytes = 39)
[20444.476080] USB request
[20444.477902] USB response (transferred_bytes = 64)
...
```

Werden die Aufwachzeiten bei einem Polling-Intervall von 100 ms überprüft, so ist zu erkennen, dass der Polling-Thread tatsächlich alle 100 ms aufwacht, um den GPS-Empfänger zu pollen. Es kommt zu keiner Verzögerung durch den GPS-Empfänger, siehe Abbildung 4.18.

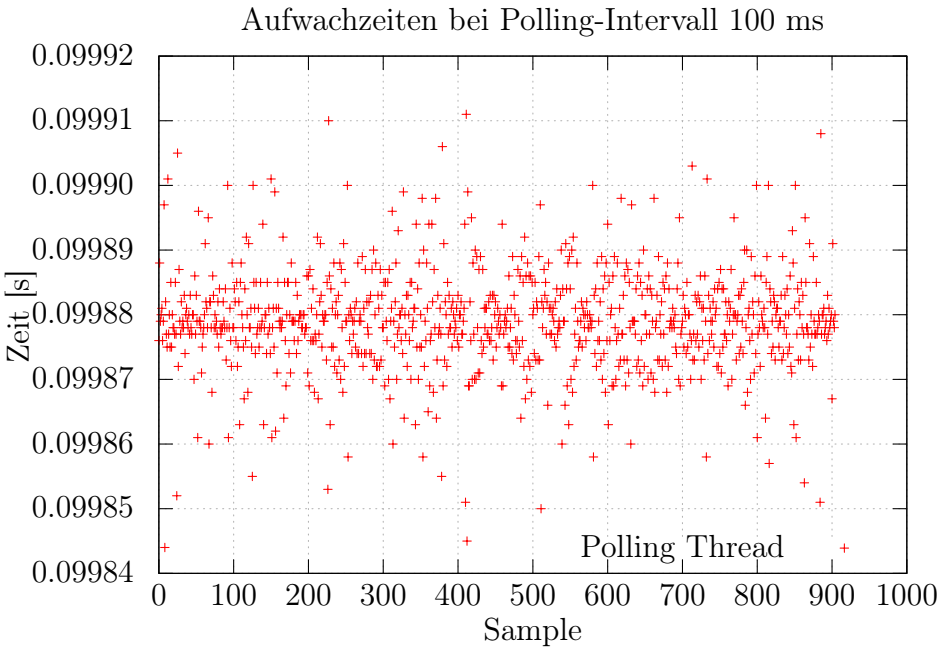


Abbildung 4.18.: ublox: Differenz Aufwachzeiten Polling-Thread bei 1 Hz Update-Rate über 90 s mit 100 ms Polling-Intervall

Die statistische Auswertung für Abbildung 4.18 ist in Tabelle 4.12 zusammengefasst.

Tabelle 4.12.: ublox: Statistik Aufwachzeiten Polling-Thread bei 100 ms Polling-Intervall

Erwartungswert	**Standardabweichung**	**Min**	**Max**
0,0998790365	7,88086687081149E-006	0,099844	0,099911

Die Zeit seit dem Auftreten des Pulses betrug bei dieser Messung durchschnittlich 0,069 ms, siehe Abbildung 4.19.

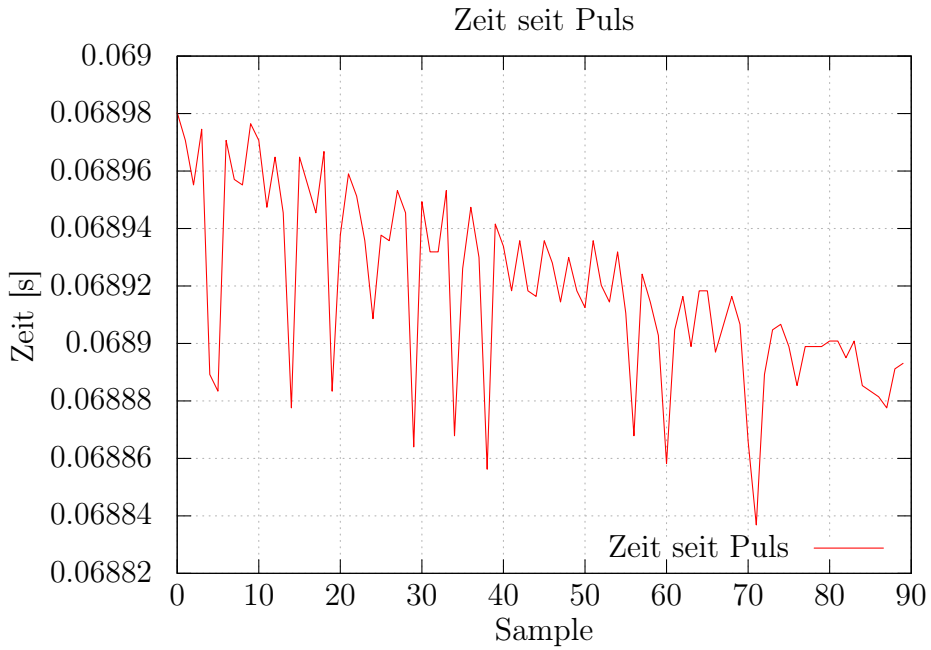


Abbildung 4.19.: ublox: Zeit seit Puls bei 1 Hz Update-Rate über 90 s bei 100 ms Polling-Intervall

Ursprünglich wurde ein Polling-Intervall von 20 ms angestrebt, um einen Kompromiss aus Ressourcenverbrauch und Aktualität der GPS-Daten zu erhalten. Der GPS-Empfänger ist jedoch nicht in der Lage ein solches Intervall einzuhalten, wie Abbildung 4.16 auf Seite 83 zeigt. Im Anschluss wird das Polling-Intervall auf 100 ms erhöht, um die minimalen Anforderungen der Anwendungsfälle von Seite 18 ff. zu erfüllen. Ein Polling-Intervall von 100 ms kann zwar eingehalten werden, jedoch beträgt die Verzögerung zwischen Eintreffen der GPS-Daten an der Antenne des Empfängers und der Verarbeitung der Daten durch die CPU durchschnittlich 69 ms. Bei einem fahrenden Auto hat dies zur Folge, dass sich in dieser Zeit das Auto weiter fortbewegt und Daten verarbeitet, die 69 ms veraltet sind. Diese Verzögerung kann verschiedene Ursachen haben. Es besteht die Möglichkeit, dass der GPS-Empfänger eine so lange Zeit benötigt, um die GPS-Signale auszuwerten. Das Datenblatt [21] nennt keine internen Verarbeitungszeiten für den Empfänger. Eine maximale Update-Rate von 10 Hz legt nahe, dass die maximale Verarbeitungszeit 100 ms betragen könnte. Eine weitere Möglichkeit, welche diese lange Verzögerung erklärt, ist ein ungünstiges Polling-Intervall. Bei einem Intervall von zum Beispiel 100 ms und einer Pulsfrequenz von 1 Hz werden immer die selben Abtastpunkte gewählt: 100 ms, 200 ms, ..., 1100 ms, 1200 ms usw. Tritt der Puls das erste Mal zum Beispiel nach 30 ms auf, so wird der Puls erst beim Abtastpunkt 100 ms erkannt. Die Zeit seit Auftreten des Pulses beträgt dann 70 ms. Nach einer Sekunde tritt der Puls bei 1030 ms auf und wird erst mit dem Abtastpunkt 1100 erkannt. Eine letzte Messung soll zeigen,

ob die Zeit seit Auftreten des Pulses durch den GPS-Empfänger begrenzt ist oder durch ein anderes Polling-Intervall verbessert werden kann. Die Messung mit einem Polling-Intervall von 124 ms ist in Abbildung 4.20 zu sehen und zeigt, dass die Zeit seit Auftreten des Pulses auf bis auf 1 ms reduziert werden kann.

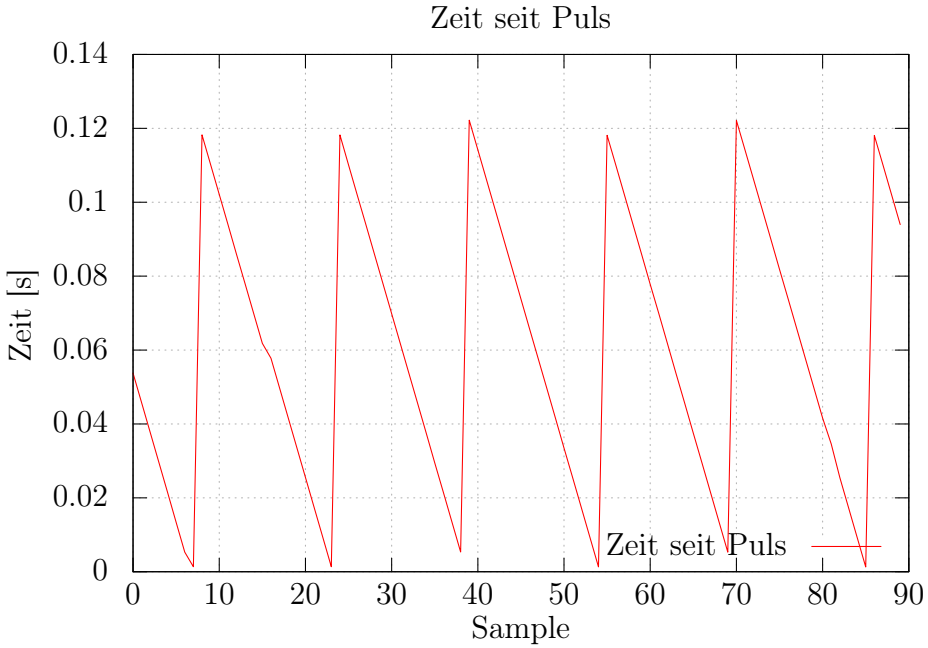


Abbildung 4.20.: ublox: Zeit seit Puls bei 1 Hz Update-Rate über 90 s bei 124 ms Polling-Intervall

5. Schluss

Der folgende Abschnitt 5.1 fasst die wesentlichen Ergebnisse dieser Arbeit zusammen und Abschnitt 5.2 gibt einen Ausblick über weitere notwendige Tätigkeiten.

5.1. Zusammenfassung

Die vorliegende Arbeit hat zum Ziel eine Softwarelösung zu entwickeln, um hochgenaue Zeit- und Positionsinformationen auf GPS-Basis zu akquirieren. Diese Daten sollen von einem Software-Framework des ESK benutzt werden, um in Car2X-Szenarien unterstützend auf den Fahrer einzuwirken. So zum Beispiel kann ein wartendes Fahrzeug an einer Kreuzung vor herannahenden Fahrzeugen und einer damit einhergehenden Kollisionsgefahr warnen. Mit dem Wort „hochgenau“ wurden Erwartungen geweckt Genauigkeiten im Mikrosekunden- bzw. Zentimeterbereich zu erreichen. Diese Arbeit beschäftigte sich zuerst mit den technischen Grundlagen, darunter GPS im Allgemeinen und GPS-Empfängern im Speziellen und zeigt auf, dass der verwendete GPS-Empfänger NEO 7 des Herstellers u-blox unter Verwendung von DGPS nur bis auf 2,0 m genau Daten liefert. Zudem liefert der GPS-Empfänger GPS-Daten mit einer maximalen Update-Rate von 10 Hz. Dies erfüllt noch die minimalen Anforderung, die vom ESK erwartet werden. Durch diese technischen Gegebenheiten lassen sich jedoch Positionsgenauigkeiten im Zentimeterbereich nicht realisieren. Zeitinformationen des GPS-Signals hingegen lassen sich durch die PPS-Unterstützung des GPS-Empfängers so korrigieren, dass Latenzen bei der Verarbeitung des GPS-Signals im GPS-Empfänger und im ARTiS PC berücksichtigt werden. Durch diese Korrektur ist mit einer größtmöglichen Genauigkeit zu rechnen. Absolute Zahlen zur Zeitgenauigkeit können nicht gegeben werden, da eine Möglichkeit zur Verifikation zum Beispiel durch eine IMU fehlt. Um trotzdem eine Anhaltspunkt für die Zeitgenauigkeit zu bekommen, werden die korrigierten Zeitinformationen mit der Zeit eines NTP-Servers verglichen. Je nach Implementierung ergaben sich dabei Abweichungen zwischen 0,0002 und 0,0037 s bzw. 0,0024 und 0,0026 s. Die Arbeit schafft somit im ersten Schritt ein Verständnis für die technischen Gegebenheiten.

Im zweiten Schritt werden zwei Implementierungsansätze entwickelt, um die Verarbeitung der GPS-Daten möglichst latenzfrei zu gewährleisten. Der erste Ansatz baut auf den existierenden Linux-Daemon `gpsd` auf. Dadurch kann schnell ermittelt werden, dass zwischen Ankunft der GPS-Daten an der Antenne des GPS-Empfän-

gers und der Verarbeitung der Daten durch den ARTiS PC durchschnittlich 85 ms liegen. Daraus resultiert bei fahrenden Autos, zusätzlich zur Ungenauigkeit des GPS-Empfängers, eine weitere Fehlerquelle. Ein fahrendes Auto verarbeitet Positionsdaten, die um 85 ms veraltet sind. Um diese Verzögerungszeit zu minimieren, wird ein zweiter Implementierungsansatz entwickelt. Dieser Ansatz ist als Linux-Treiber umgesetzt und sieht vor, den GPS-Empfänger in einem regelbaren Intervall zu Pollen. Eine verlässliche Konfiguration des GPS-Empfängers ist nicht möglich und so kann der Empfänger nur mit Standardeinstellungen, darunter eine Update-Rate von 1 Hz, betrieben werden. Unter dieser Voraussetzung wird eine Polling-Intervall von 20 ms angestrebt, welches ein Kompromiss aus Ressourcenverbrauch und Aktualität der GPS-Daten darstellt. Trotz diesem Intervall kann die Zeit zwischen Eintreffen der GPS-Daten an der Antenne des GPS-Empfängers und der Verarbeitung der Daten im ARTiS PC nicht reduziert werden. Eine genaue Analyse des Scheduling-Verhaltens zeigt jedoch auf, dass dies nicht der Software geschuldet ist, sondern dem GPS-Empfänger. Dieser ist nicht in der Lage in einem so kurzen Intervall GPS-Daten zu liefern und das obwohl die theoretischen Voraussetzungen bezüglich Übertragungsrate gegeben wären. Ob sich dieses Verhalten durch eine Erhöhung der Update-Rate des GPS-Empfängers verbessern lässt, konnte auf Grund fehlender Konfigurationsmöglichkeiten des Empfängers nicht geklärt werden. Durch Simulation der USB-Kommunikation des GPS-Empfängers kann nachgewiesen werden, dass zumindest die Software in der Lage ist auch ein Update-Intervall von 20 ms einzuhalten. Das Kernel-Modul kann somit als Grundlage für zukünftige Arbeiten dienen, sofern die notwendigen Hardwarevoraussetzungen erfüllt werden. Zudem konnte in einer Messung mit einem Polling-Intervall von 124 ms nachgewiesen werden, dass die Verarbeitungszeit zwischen Eintreffen der GPS-Daten an der Antenne des GPS-Empfängers und der Verarbeitung der Daten durch die CPU im ARTiS PC auf bis zu 1 ms reduziert werden kann.

Grundsätzlich ist aber das Konzept zu überdenken, dass ausschließlich über GPS Zeit- und Positionsinformationen bezogen werden, um die beschriebenen Anwendungsfälle aus Abschnitt 1.1 ab Seite 18 umzusetzen. Zum Einen wird für GPS eine ständige Sichtverbindung zum Himmel benötigt, um ausreichend Signalqualität vorweisen zu können. Dies ist im Straßenverkehr zum Beispiel durch Tunnel und tiefe Häuserschluchten nicht immer gegeben. Es mangelt an dieser Stelle an Verlässlichkeit, welche aber in einer sicherheitskritischen Umgebung wie dem Personenverkehr gegeben sein muss. Eine weitere Problematik stellt die Genauigkeit bei GPS dar, welche je nach GPS-Empfänger variiert. Beim vorliegenden GPS-Empfänger beträgt diese 2 m bei Nutzung von DGPS. Eine weitere Einschränkung ergibt sich durch die begrenzte Update-Rate von GPS-Empfängern. Nur in diskreten Intervalle sind Zeit- und Positionsinformationen bekannt. Zwischen diesen „Abtastungen“ sind keine Informationen bekannt. Diese „Informationslücke“ kann zum Beispiel durch die Fusion mit anderen Sensordaten geschlossen werden.

5.2. Ausblick

Die bisherige Softwarelösung durch das Linux-Kernel-Modul `ublox` ist im jetzigen Zustand mehr Framework als vollständig abgeschlossene Software. Um es in finalen Produkten des ESK einsetzen zu können, sind weitere Verbesserungen notwendig. Zum Einen Bedarf es einer verlässlichen Konfiguration des GPS-Empfängers und zum Anderen kann das Kernel-Modul optimiert werden, sodass ein feineres Polling-Intervall eingestellt werden kann. Beide Ansätze werden im Folgenden näher erläutert.

Im Entwicklungszeitraum konnte der GPS-Empfänger nur eingeschränkt konfiguriert und oftmals nur mit Standardeinstellungen genutzt werden. Im weiteren Verlauf muss deshalb Kernel-Modul und GPS-Empfänger optimal aufeinander abgestimmt werden, sodass Anforderungen des ESK erfüllt werden, so zum Beispiel eine Update-Rate des Empfängers von 10 Hz. Für Konfiguration und Evaluation seiner GPS-Produkte bietet der Hersteller u-blox die kostenlose Software „u-blox u-center“ an. Diese ist nur unter Microsoft Windows lauffähig. Um die Software auf dem ARTiS PC nutzen zu können, wäre dort eine Installation von Microsoft Windows notwendig. Auf Grund der eingeschränkten Hardware-Ressourcen des ARTiS PC bietet sich Windows XP an, wobei dies standardmäßig nicht alle notwendigen Treiber mitliefert, um das Betriebssystem auf der ARTiS-Hardware installieren zu können. Eine alternative Herangehensweise wäre „u-blox u-center“ unter dem bereits installierten Betriebssystem Debian GNU/Linux 7.5 mittels „Wine“ auszuführen. Diese Kompatibilitätsschicht erlaubt es Windows-Programm in Linux auszuführen. Als letzte Möglichkeit bietet sich an das Programm `gpsctl` genauer zu untersuchen und zu klären warum eine erfolgreiche Konfiguration durch UBX-Nachrichten nur sporadisch möglich ist.

Das Kernel-Modul `ublox` pollt bisher in einem Intervall den GPS-Empfänger, welches an die Größe „Jiffy“ gebunden ist. Dadurch lässt sich das Intervall nur in 4-ms-Schritten regeln. Um ein feineres Polling-Intervall gewährleisten zu können, sollte der Polling-Mechanismus, der bisher durch eine `workqueue` realisiert, durch einen `kthread` ersetzt werden.

Literaturverzeichnis

[1] BMW Group: *Car2x-Communication.* Website, 2010. – Online unter https://www.press.bmwgroup.com/usa/download.html?textId=95374&textAttachmentId=117414, besucht am: 26.10.2014

[2] Bovet, Daniel P. ; Cesati, Marco: *Understanding the Linux Kernel.* 3rd Edition. Upper Saddle River, New Jersey : Pearson Prentice Hall, 2005. – ISBN 978–0–5960–0565–8

[3] Brown, Amy ; Wilson, Greg: *The Architecture of Open Source Applications.* Volume II. 2012. – ISBN 978–1–1055–7181–7

[4] Corbet, Jonathan ; Kroah-Hartman, Greg ; Rubini, Alessandro: *Linux Device Drivers.* 3rd Edition. Sebastopol, California : O'Reilly Media, 2005. – ISBN 978–0–5960–0590–0

[5] Dr. Zinas, Nicholas: *GPS Network Real-Time-Kinematic Process.* Website, 2014. – Online unter http://www.tekmon.gr/gps-network-rtk-tutorial/#, besucht am: 15.10.2014

[6] Edge, Jake: *The Future of the Realt-Time Patch Set.* Website, 2014. – Online unter http://lwn.net/Articles/617140/, besucht am: 03.11.2014

[7] embedded brains: *ARTiS PC User's Manual.* Version 1.2.1 for Hardware Revision 1.4.1. Dornierstraße 4, 82178 Puchheim, 2014

[8] ETSI: *Technical Report 102 638 — Intelligent Transport Systems; Vehicular Communications; Basic Set of Applications; Definitions.* Version 1.1.1. Valbonne, France, 2009

[9] Jones, M. T.: *Inside the Linux 2.6 Completely Fair Scheduler.* Website, 2009. – Online unter http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/l-completely-fair-scheduler-pdf.pdf, besucht am: 22.10.2014

[10] Koolwal, Kushal: *Investigating Latency Effects of the Linux Real-Time Preemption Patches (PREEMPT_RT) on AMD's GEODE LX Platform.* Website, 2009. – Online unter http://www.versalogic.com/downloads/whitepapers/real-time_linux_benchmark.pdf, besucht am: 04.11.2014

[11] MediaTek: *MT3332 GNSS Host-Based Solution — Technical Brief*. Website, 2014. – Online unter http://labs.mediatek.com/fileMedia/download/4c04f268-9f17-4525-954c-d1ae6fdc2f57, besucht am: 15.10.2014

[12] Mills, David L.: *Network Time Protocol (NTP) General Overview*. Website, 2004. – Online unter http://www.eecis.udel.edu/~mills/database/brief/overview/overview.pdf, besucht am: 12.10.2014

[13] Mills, David L.: *NTP Architecture, Protocol and Algorithms*. Website, 2007. – Online unter http://www.eecis.udel.edu/~mills/database/brief/arch/arch.pdf, besucht am: 12.10.2014

[14] Navilock: *Navilock EM-506 GNSS GPS SiRF Star IV TTL PPS Engine Board*. Website, 2014. – Online unter http://www.navilock.de/produkt/60432/pdf.html?sprache=en, besucht am: 15.10.2014

[15] Paoloni, Gabriele: *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Website, 2010. – Online unter http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf, besucht am: 18.10.2014

[16] Stübing, Hagen u. a.: *Sichere Intelligente Mobilität Testfeld Deutschland: Konsolidierter Systemarchitekturentwurf*. Website, 2009. – Online unter http://www.simtd.de/index.dhtml/object.media/deDE/6459/CS/-/backup_publications/Projektergebnisse/simTD-Deliverable-D21.2_Konsolidierter_Systemarchitekturentwurf.pdf, besucht am: 04.11.2014

[17] Tanenbaum, Andrew S.: *Moderne Betriebssysteme*. 3. Auflage. Upper Saddle River, New Jersey : Pearson Prentice Hall, 2009. – ISBN 978–3–8632–6561–8

[18] Trimble: *Trimble BD982 GNSS Receiver Module*. Website, 2014. – Online unter http://www.trimble.com/gnss-inertial/pdf/bd982_ds_0411.pdf, besucht am: 15.10.2014

[19] Ts'o, Theodore u. a.: *Real-Time Linux Wiki*. Website, 2014. – Online unter https://rt.wiki.kernel.org/index.php/Main_Page, besucht am: 22.10.2014

[20] u-blox: *u-blox 7 Receiver Description*. GPS.G7-SW-12001-B. Zürcherstraße 68, 8800 Thalwil, 2013

[21] u-blox: *NEO-7*. Website, 2014. – Online unter http://www.u-blox.com/images/downloads/Product_Docs/NEO-7_DataSheet_%28GPS.G7-HW-11004%29.pdf, besucht am: 15.10.2014

[22] Wenzel, Andreas u. a.: *Sichere Intelligente Mobilität Testfeld Deutschland: TP5-Abschlussbericht — Teil B-3 Technische Bewertung.* Website, 2013. – Online unter `http://www.simtd.de/index.dhtml/object.media/deDE/8118/CS/-/backup_publications/Projektergebnisse/simTD-TP5-Abschlussbericht_Teil_B-3_Technische_Bewertung_V10.pdf`, besucht am: 05.11.2014

[23] Zogg, Jean-Marie: *GPS und GNSS: Grundlagen der Ortung und Navigation mit Satelliten.* 4. Auflage. Thalwil, Schweiz : u-blox AG, 2011

A. Anhang auf Datenträger

Der Anhang wird auf einem Datenträger bereitgestellt und umfasst das gesamte Mercurial-Repository, welches im Rahmen der Masterarbeit zur Versionierung von Quelltexten und Messungen entstanden ist. Das Repository besteht auf oberster Ebene aus vier Verzeichnissen:

- `doc`: Enthält unter anderem Paper und Datenblätter.
- `logs`: Enthält Messungen die im Laufe der Masterarbeit entstanden sind. Alle Messungen wurden zu einem ZIP-Paket zusammengefügt.
- `src`: Enthält alle Quelltexte und die Skripte zur Generierung von Messdaten. Im Unterverzeichnis `application` findet sich unter anderem die Modifikation zum Linux-Daemon `gpsd`. Das Unterverzeichnis `kernel` enthält das Linux-Kernel-Modul `ublox`.
- `test`: Enthält Unit-Tests für Code der auf Anwendungsebene entstanden ist, so zum Beispiel der Parser für das NMEA-Protokoll.