OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Smart Car

Hauptseminar Projektstudium

An der
Ostbayerischen Technischen Hochschule Regensburg
Fakultät Informatik/Mathematik

Betreut durch Prof. Dr. Klaus Volbert

Vorgelegt von:

Knorr Thomas	xyzk	HSP1
Lackner Andreas	3120204	HSP2
Schleinkofer Michael	3119501	HSP2
Welker Franz	3119754	HSP2
Wiche Fabian	xyzk	HSP1

Datum: 19. April 2018

Inhaltsverzeichnis

1 Einleitung

Ein zu beobachtender Trend ist die zunehmende Digitalisierung verschiedenster Lebensbereiche mit dem Ziel, die Effizient und Bequemlichkeit des Alltags zu steigern. Die Grundvoraussetzung für die Funktionalität bestehender und die Entwicklung neuer Lösungen ist meist eine Datenquelle welche die angebotenen Dienste mit Informationen versorgt.

Eine Industrie die dieser Trend betrifft, ist die Automobilbranche. Moderne Fahrzeuge sammeln durch die verbauten Sensoren eine Vielzahl von Daten die sowohl den Betrieb des Fahrzeugs selbst, sowie den diverser Sicherheits- und Komfortfunktionen ermöglichen. Dabei nimmt ebenfalls die Zahl der Informationen zu, die mit dem Fahrer bzw. der Umwelt geteilt werden. Dies umfasst beispielsweise den Kraftstoffverbrauch, verbleibende Reichweite oder die Motorlast sowie Positions- und Telemtriedaten die unter anderem zukünftig mit anderen Fahrzeugen oder Serviceanbietern geteilt werden. Neben den Vorteilen für andere Verkehrsteilnehmer, durch verbesserte Unfallvermeidung oder Stauerkennung, sind die Hauptprofiteure die Fahrzeugführer bzw. Besitzer die dadurch eine verbesserte Übersicht über ihr Fahrzeug und Fahrverhalten bekommen.

Von dieser Entwicklung ausgeschlossen sind vor allem Besitzer älterer Fahrzeuge, die oft nur eine begrenzte Zahl von Informationen über die Kombiinstrumente teilen und keine Schnittstelle besitzen, um automatisiert Daten zu extrahieren.

1.1 Verfügbare Lösungen

Um auch Fahrzeuge älteren Baujahres oder eingeschränkter Ausstattung mit den Vorteilen moderner Analyse und Statistik Applikationen auszustatten, sind am Markt verschiedene Lösungen verfügbar:

- Smartphone Applikationen
  Hier finden sich in den App Stores verschiedene Anwendungen die Services wie das automatisierte führen von Fahrtenbüchern (`https://www.mycartracks.com/`) oder Navigation gepaart mit Echtzeitmeldungen über Stau oder andere Hindernisse anbieten (`https://www.waze.com/de/`). All diese Applikationen verwenden für ihren Betrieb entweder nur Positionsdaten durch das GPS Modul des Smartphones oder manuelle Eingaben der Fahrer. Die Nachteile bestehen darin, dass sowohl viel Interaktion mit der App erforderlich ist (Start und Stopp bei Fahrtantritt und -ende) als auch keine Fahrzeuginternen Daten zur Verfügung stehen, die vor allem bei Fahrzeugen ohne Boardcomputer einen bedeutenden Mehrwert darstellen.

- Pace (`https://www.pace.car/de`)
  Das Produkt der Firma Pace löst das Problem der fehlenden Fahrzeugschnittstelle über einen Adapter der mittels OBDII Zugriff auf Fahrzeugdaten ermöglicht. Gepaart mit einer App können so Services wie ein Performance Monitor, Fehleranalyse, Spritspartraining oder ein elektronisches Fahrtenbuch angeboten werden. Der ständige Betrieb der App ist dabei zwingend erforderlich, da der eingesetzte Adapter weder über internen Speicher noch über ein eigenes GPS Modul verfügt, um eigenständig alle relevanten Daten aufzuzeichnen. Auch hier ist dadurch die ständige manuelle Interaktion des Fahrers notwendig.

- Mojio (`https://www.moj.io/`)
  Im Unterschied zu den zuvor vorgestellten Ansätzen ist das Ziel von Mojio nicht nur ein Gadget für Fahrzeugbesitzer anzubieten, sondern eine Plattform für Fahrzeugdaten zu entwickeln. Die Telemetriedaten werden wie auch bei Pace über einen OBDII Adapter erfasst und in das Mojio Backend geladen. Die Firma stellt dabei lediglich die Schnittstellen und die Plattform zur Verfügung, weshalb die Anwendung nicht auf einen speziellen Adapter beschränkt ist. Mit den von vielen Fahrzeugen gesammelten Daten können im Nachgang verschiedene Statistiken und Analysen erstellt oder eigenen Applikationen mit Daten versorgt werden.

1.2 Zielsetzung

Ziel des Projekts ist die Entwicklung einer ganzheitlichen Lösung, um Fahrzeuge die nur über eingeschränkte Informationsschnittstellen verfügen nachträglich mit den Vorzügen moderner Visualisierungs- und Statistikapplikationen auszustatten. Dazu werden die Vorteile der in 1.1 beschriebenen Lösungen vereint. Um die Fahrzeugdaten extrahieren zu können, wird ein Programmierbarer OBDII Dongle verwendet, der zusätzlich über einen eigenen GPS Empfänger und Speicher verfügt, um unabhängig von einem Smartphone Daten aufzeichnen zu können. Die zusätzliche App mit deren Hilfe sich verschiedene Fahrzeugparameter in Echtzeit visualisieren lassen, ist damit optional. Für eine statistische Analyse der gesammelten Bewegungs- und Telemetriedaten soll ein Backend zur Verfügung gestellt werden, welches entweder über die auf der Speicherkarte gesicherten Logs oder einen Upload über das Smartphone gespeist wird. Basierend auf dieser Infrastruktur lassen sich in weiterer Folge verschiedene neue Applikationen aufsetzten.

Die Gliederung des Berichts beginnt mit einer Aufstellung der Use Cases in 2 die von der finalen Lösung umgesetzt werden sollen. Darauf folgt in Kapitel 3 eine Übersicht über die Systemarchitektur, gefolgt von der Dokumentation der Implementierung in 4, die in die jeweiligen Teilprojekte gegliedert ist. Abgeschlossen wird der Bericht durch einen Test des Gesamtsystems (5) sowie einen Ausblick auf mögliche weitere Entwicklungen (6).

2 Use Cases

Was haben wir uns vorgestellt, dass wir am Ende erreichen wollen. Z.B. einfaches wie Strecken im Backend anzeigen mit infos wie länge, V,... Im Dongle aufnehmen können In der App live Daten sehen können

3 System Design

3.1 Dongle

Wie auf der Produkthomepage beschrieben, nutzt der Dongle als Haupt-Controller einen ATmega328p, wie er auch auf einem Arduino UNO verwendet wird. Bei der Architektur der Dongle-Software wird deshalb für den Programmablauf ein für Arduino-Projekte klassischer Aufbau mit einer „setup“- und einer „loop“-Funktion innerhalb einer main-Datei verwendet.

Um bei der Entwicklung der Software für den Dongle möglichst wenig Inhaltsüberschneidungen der Teammitglieder zu erreichen und um die Verständlichkeit und Wartbarkeit des Codes zu verbessern wurde entschieden die Schichtenarchitektur wie in Bild 3.1 umzusetzen. Hierbei wird für die meisten Funktionsmerkmale mindestens eine Klasse auf der Intermediate-Layer sowie der Driver-Layer implementiert. Dies hat zur Folge, dass bei einer Funktionsänderung wie beispielsweise der Verwendung eines anderen GPS-Empfängers nur die entsprechende Treiber-Klasse geändert werden muss. Die Hauptklasse mit der eigentlichen Programmlogik bleibt dabei unangetastet.
Die im Bild 3.1 mit „(Fm)“ ergänzten Klassen werden aus den Bibliotheken des Herstellers übernommen.

3.1.1 GPS-Empfänger

Da ein zentrales Ziel der Applikation die Anfertigung eines Fahrtenbuches mit Streckenaufzeichnung ist, muss auch die Position des Fahrzeuges möglichst genau bestimmt werden. Der in diesem Projekt verwendete Freematics ONE bietet hier

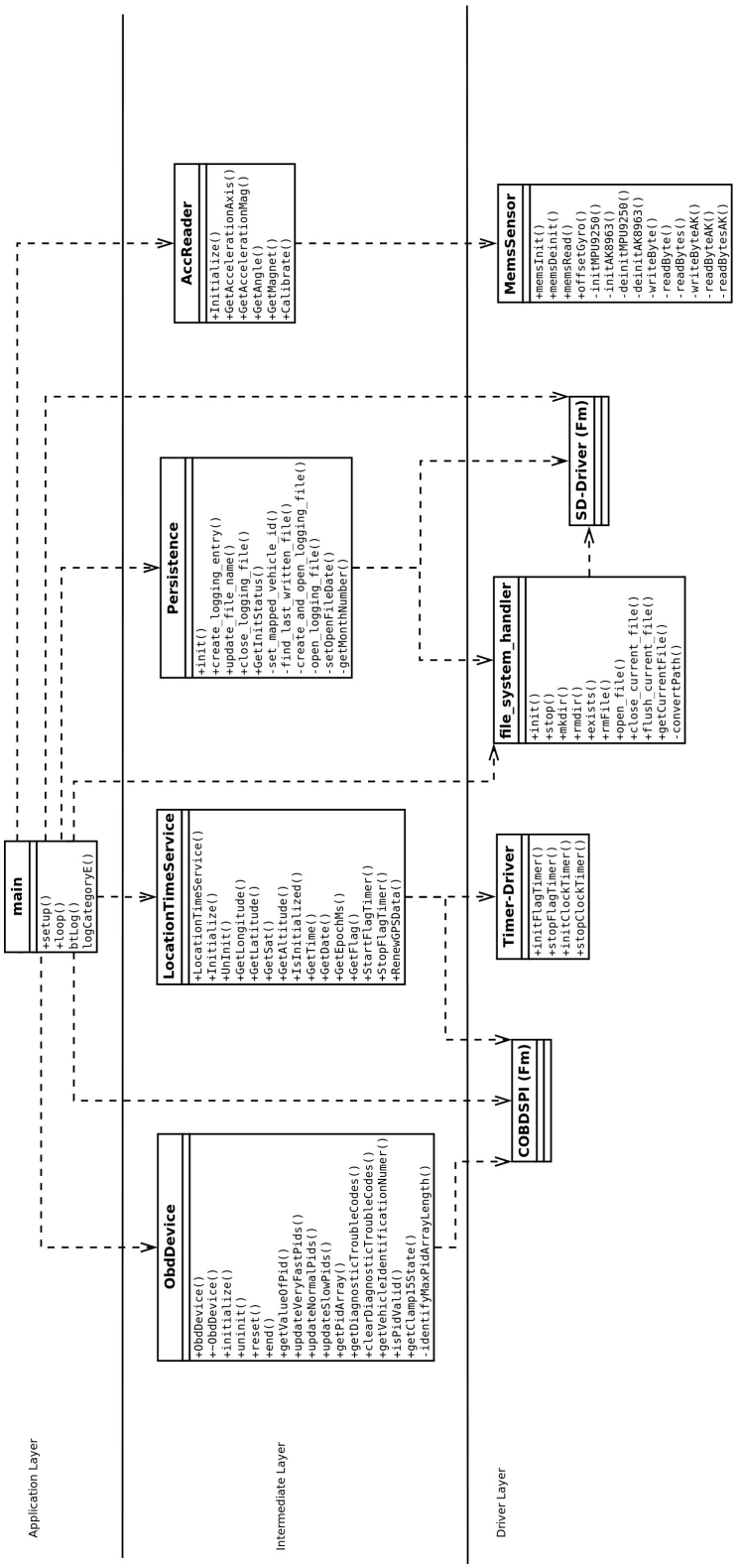

Abbildung 3.1: Architektur der Dongle-Software

die Möglichkeit, einen externen GPS-Empfänger über eine Universal asynchronous receiver-transmitter (UART)-Schnittstelle anzubinden.

Um die Anschaffungskosten zu reduzieren, wurde zunächst untersucht, ob neben dem von Freematics verkauften GPS-Empfänger auch andere GPS-Receiver-Chips mit dem OBD-Dongle kompatibel sind. Ein Problem bei dieser Untersuchung ist die Architektur des Freematics ONE, da die Kommunikation mit dem GPS-Empfänger nicht auf dem ATmega328p Haupt-Controller sondern auf einem STM32 Coprozessor ausgeführt wird. Leider ist der Code auf dem Coprozessor nicht öffentlich einsehbar und auch nicht ohne großen Aufwand auslesbar. Ein weiteres Problem bestand darin, dass auch ein Öffnen des Gehäuses des von Freematics selbst vertriebenen GPS-Empfängers nicht zur Identifikation des Chips beitragen konnte. Es wurde allerdings klar, dass dieser nicht der Angabe auf der Produkthomepage des Freematics ONE entsprach. Der Empfänger-Chip ist nur mit einem QR-Code versehen und eine Recherche zum Hersteller verwies nur auf den chinesischen Produzenten des ganzen Empfänger-Moduls.(www.szgrltd.com)

Daher wurde eine andere Vorgehensweise zur Untersuchung der Kommunikation angewandt. Dazu wurde, wie in Abbildung 3.2 abgebildet, ein zusätzlicher Arduino UNO als Zwischenstation in die UART-Kommunikation zwischen Dongle und Empfänger eingefügt. Zwei durch Software simulierte, serielle Schnittstellen auf dem Arduino UNO werden nun genutzt, um die vom Dongle und vom GPS-Empfänger gesendeten Daten aufzufangen, auf der über USB angeschlossenen seriellen Konsole eines Rechners auszugeben und an den jeweils anderen Kommunikationspartner weiterzuleiten.

Nach Auswertung der Kommunikation, stand fest, dass der von Freematics gelieferte GPS-Empfänger kompatibel zu einem u-blox UBX-G7020 ist. Dieser versendet standardmäßig Nachrichten gemäß dem National Marine Electronics Association (NMEA) Standard. Darüber hinaus wurde ersichtlich, dass der OBD-Dongle keine Nachrichten zum GPS-Chip sendet.

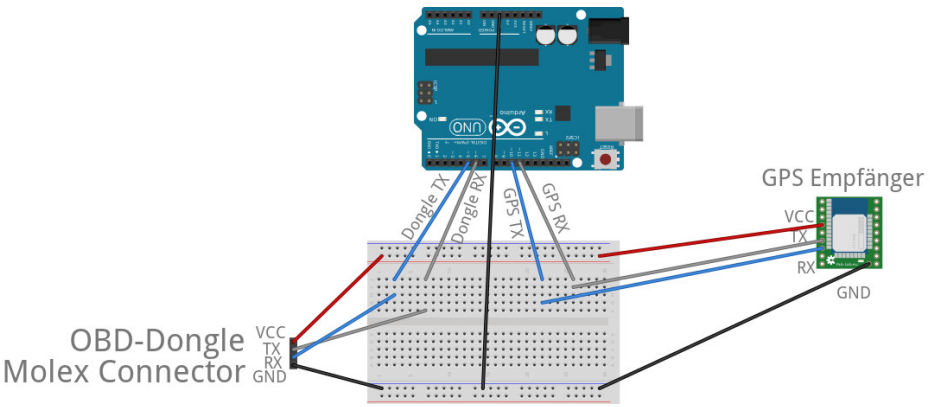

Abbildung 3.2: Versuchsaufbau zur Analyse der UART-Kommunikation zwischen Dongle und GPS-Empfänger

Da nun allerdings der konkrete Empfänger feststand, konnte dazu die entsprechende Protocol Specification heruntergeladen und mit weiteren GPS-Empfängern verglichen werden. Letztendlich wurde ein Pixhawk GPS Empfänger für einen Modellbau-Quadrokopter auf Basis eines u-blox Neo6M mit zusätzlichem Magnetfeld-Sensor ausgewählt. Dieser Mikrochip verfügt zwar nicht über die exakt gleiche Protocol Specification, die ab Werk konfigurierte Kommunikation jedoch ist nahezu identisch und kompatibel mit der des von Freematics gelieferten Produktes.

Um den neuen Empfänger am Dongle zu betreiben, wurde an dessen Signal-Eingängen ein 2x2-Molex Stecker passend angelötet. Die I2C-Pins des Magnetfeld-Sensors wurden dabei nicht belegt.

Ein erster Test mit der mitgelieferten Software zeigte die grundsätzliche Funktion des neuen GPS-Moduls. Allerdings ist die Genauigkeit des Pixhawk-Empfängers etwas schlechter als die des UBX-G7020.

3.1.2 OBD-Schnittstelle

Da der Dongle vorwiegend genutzt werden soll um die OBD-Daten (kurz Parameter Identifikators (PIDs)) aufzuzeichnen besitzt der Freematics ONE eine hierfür

passende Schnittstelle. Bei den PIDs[**OBD-2.net2018**] handelt es sich um Daten die von bestimmten Sensoren (z.B. Drehzahl, Geschwindigkeit, Öl-Temperatur, ...) im Auto zu Verfügung gestellt werden.
Bei OBD handelt es sich um ein Protokoll welches nach dem „challenge-response"-Prinzip funktioniert. Dabei werden vom Client (Dongle) Anfragen an den Server (OBD-Steuergerät im Auto) gesendet. Dieses antwortet anschließend mit den Werten des entsprechenden Sensors.
Da es vom vorliegenden Auto abhängig ist, welche Steuergeräte verbaut sind und somit auch was für PIDs unterstützt werden, ist es nicht bei jedem Auto möglich die selben Daten auszulesen.
Da weiterhin der Platz auf dem Freematics ONE (sowohl RAM als auch Flash) sehr begrenzt ist, wurde von vorne herein entschieden, nur bestimmte PIDs zu verwenden. Insgesamt wurden xy viele PIDs ausgewählt. Um sowohl die OBD-Schnittstelle als auch den Prozessor im Freeatics ONE unter wenig Last auszusetzen, dabei allerdings möglichst viel Nutzen aus den gewonnen Informationen ziehen zu können, wurden die PIDs in unterschiedliche Gruppen eingeteilt.
Eine Gruppe entspricht somit einem Zusammenschluss aus PIDs welche alle in einem gleichen Zeitintervall abgerufen werden. Es wurde sich für folgende Gruppen entschieden (Tabelle 3.1.2):

Kategorie	Intervall
A	500ms
B	1,5s
C	15s
D	5min
E	neue Route
F	neues Auto

Der Tabelle 3.1.2 kann zum einen entnommen werden, welche PIDs ausgewählt wurden und in welcher Kategorie sie zugeordnet wurde.

Name	Wert	Kategorie
Engine coolant temperature	0x05	C
Engine RPM	0x0C	A
Vehicle speed	0x0D	A
Run time since engine start	0x1F	C
Distance traveled with malfunction indicator lamp	0x21	D
Fuel tank level input	0x2F	D
Absolute barometric pressure	0x33	C
Ambient ari temperature	0x46	C
Fuel type	0x51	E
Ethanol fuel %	0x52	E
Relative accelerator pedal position	0x5A	A
Engine oil temperature	0x5C	C
Engine fuel rate	0x5E	A
Driver's demand engine-percent torque	0x61	A
Actual engine-percent torque	0x62	A
Engine reference torque	0x63	A
Engine run time	0x7F	C

3.1.3 Zeit

Wie bereits erwähnt, muss auch auf dem Dongle eine Repräsentation der genormten Zeit vorhanden sein. Zunächst soll jeder erfasste Datenwert mit einem Zeitstempel versehen werden um mit einer totalen Ordnung die Analyse dieser Werte erst zu ermöglichen. Zum anderen sollen die Datenwerte mit einem Intervall von 500 Millisekunden erfasst werden.

Die Anforderung nach einem genauen Zeitintervall von 500 Millisekunden zwischen dem Abrufen der OBD-Werte der Kategorie A kann durch den Einsatz eines Hardware-Timers und Interrupts gelöst werden. Auf dem ATmega328p Hauptcontroller stehen dem Entwickler 3 Hardware-Timer zur Verfügung. Allerdings muss hierbei beachtet werden, dass die Arduino-Bibliothek den Timer 0 für die Funktionen delay() und millis() verwendet und diese daher unangetastet bleiben

sollten.[**arduinoTimer**] Da die Intervalle zum Abrufen der PID-Kategorien B, C und D ein Vielfaches der 500 Millisekunden der Kategorie A sind, müssen für diese keine weiteren Timer verwendet werden. Statt dessen kann ein einfacher Vergleich in Kombination mit dem Modulo-Operator genutzt werden (vgl. Abbildung 3.4).

3.1.4 Beschleunigungssensor

Zunächst wurde ermittelt, welcher Sensor im Dongle verbaut wurde. Anhand der Informationen auf der Produkthomepage sowie des Source-Codes des Treibers wurde ersichtlich, dass ein MPU-9250 Microelectromechanical System (MEMS)-Sensor mit jeweils 3 Achsen für Beschleunigungs-, Drehraten- und Magnetfeldmessung verbaut ist. Hierbei ist besonders, dass der Sensor für das Magnetfeld als I^2C-Submodul am Sensor ausgeführt ist.

3.1.5 Programmlogik

Die eigentliche Programmlogik kann wie bereits erwähnt in die Teile „Setup“ und „Loop“ getrennt werden.

Die Abbildung 3.3 beschreibt den Ablauf des Programms beim Einstecken des Adapters in die Schnittstelle des Autos. Hervorzuheben ist, dass die Reihenfolge der Initialisierungen von großer Bedeutung ist. Näheres dazu wird im Kapitel 4.3.4 erklärt. In Abbildung 3.4 ist der Ablauf des Programms ersichtlich, welches die eigentliche Funktionalität enthält. Dies wurde in drei Modi umgesetzt. Während des Logging-Modus werden die Fahrzeug-Daten gesammelt, auf eine SD-Karte geschrieben und per Bluetooth an ein Smartphone gesandt. Der Upload-Modus dient dazu, die gesammelten Daten auf der SD-Karte an die Smartphone-App weiterzugeben. Dies soll ein Entnehmen der Karte zum Auslesen der Daten optional machen. Der Schlafmodus dient letztendlich dazu, bei abgeschalteter Zündung das Bordnetz des Fahrzeugs möglichst wenig zu belasten.

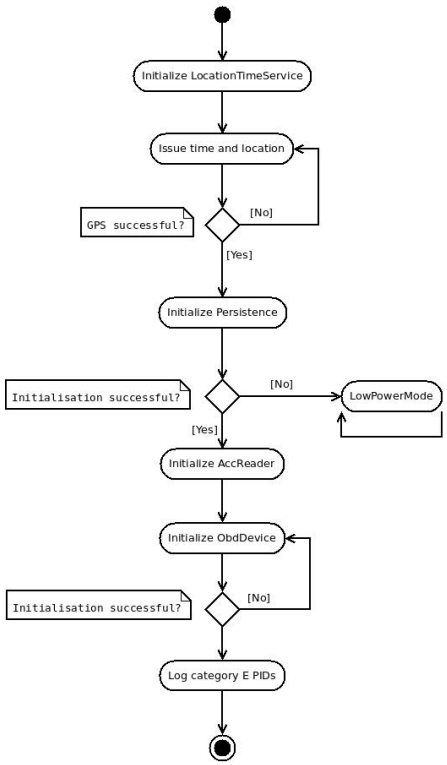

Abbildung 3.3: Programmablauf der Initialisierung

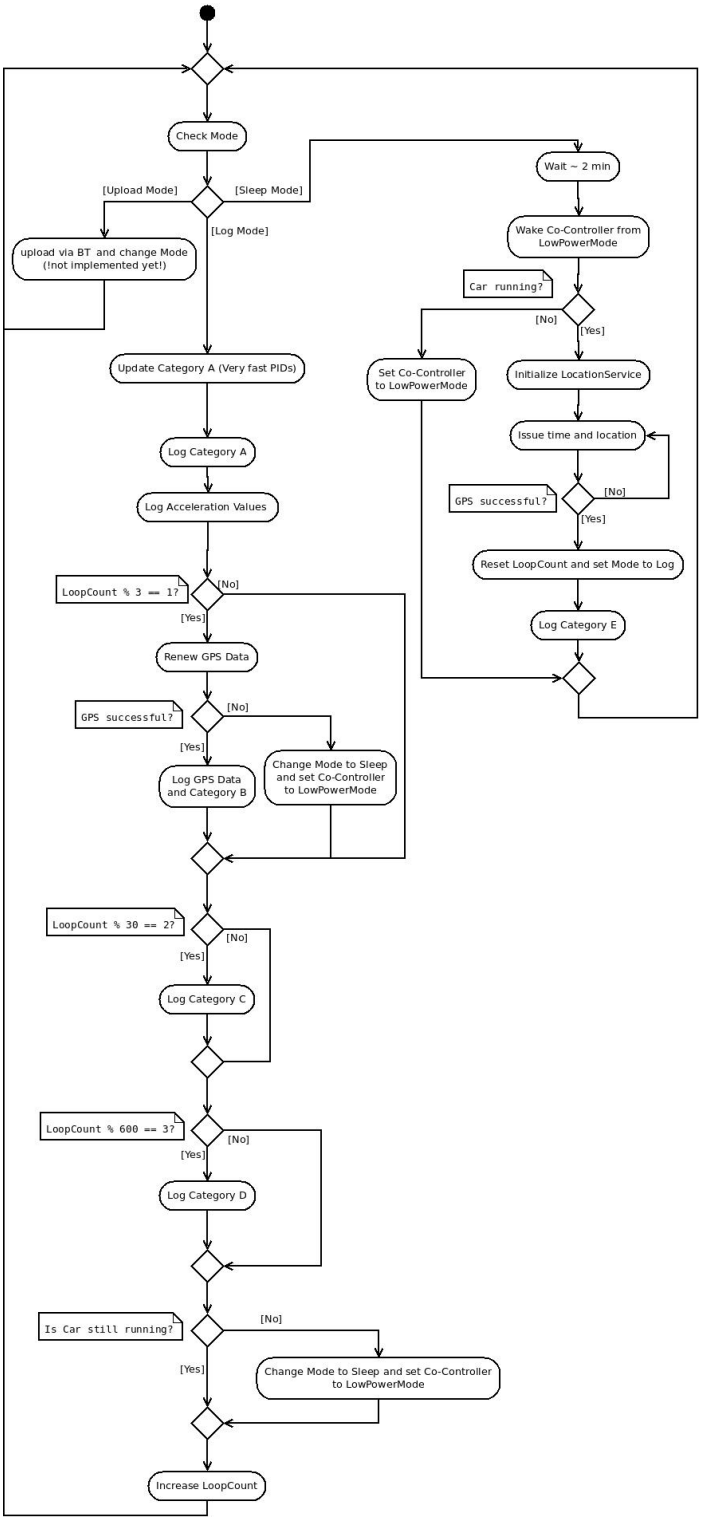

Abbildung 3.4: Programmablauf der Endlosschleife

4 Implementierung

Wie wurde es umgesetzt, wie waren die Ideen (also hier z.b. auf die Gesamte Architektur eingehen, spezifische der einzelnen Bereiche in den unter Kapiteln)

4.1 App

Was kann die App, und besonderes erklären

4.2 Backend

4.3 Dongle

4.3.1 GPS und Zeit

Um den GPS-Empfänger auch in der in diesem Projekt geschriebenen Dongle-Software zu nutzen, wurde zunächst der Code des von Freematics veröffentlichten Treibers für den Coprozessor übernommen, da dieser den Datenaustausch mit dem Empfänger regelt und die serielle Schnittstelle mit diesem verbunden ist. Um die Kommunikation von der eigentlichen Anwendungslogik abzutrennen, wurde eine weitere LocationTimeService-Klasse auf Ebene der Intermediate-Layer implementiert. Diese bietet nun vereinfachten Zugriff auf die gemessenen Werte des geografischen Längen- und Breitengrads, der Höhe über Normalnull und die Anzahl der verfügbaren Satteliten. Darüber hinaus stellt sie auch Funktionen zum erneuten Abrufen und Speichern der GPS-Daten und zur Initialisierung der Kommunikation mit dem GPS-Chip über UART zur Verfügung. Dabei ist zu beachten, dass die Anzahl der Satelliten für eine möglichst korrekte Positionsbestimmung zwischen 4 und 14 liegen muss [**gpsPrecision**].

Während der Initialisierung des LocationService, wird bis zu fünf mal versucht, über den Treiber des Coprozessors eine serielle Datenübertragung aufzubauen. Um die Genauigkeit vor allem des Pixhawk-Empfängers zu verbessern, wird während der Initialisierung der LocationService-Klasse der Intermediate Schicht der GPS-Chip für die Nutzung für Satellite-based augmentation systems (SBAS) konfiguriert. Dazu wird die write-Methode des Treibers genutzt, mit dem ein Byte-Array 1:1 übertragen werden kann. Gemäß der Protocol-Specification beider GPS-Module, kann SBAS mit folgendem Code konfiguriert werden:

```
1   uint8_t cmd[] = {0xB5, 0x62, 0x06, 0x16, 0x00, 0x08,
        0x03, 0x07, 0x00, 0x00, 0x00, 0x00, 0x51, 0x7F,
      0xEE };
2   uint8_t cmdLen = 15;
3   [...]
4   //send config command
5   _coProc->setTarget(TARGET_GPS);
6   _coProc->write(cmd, cmdLen);
```

Um die Kommunikation mit dem Coprozessor nicht unnötig zu belasten und die Verarbeitung der OBD-Daten auf diesem nicht zu kompromittieren, werden die Sensordaten nur nach Bedarf mit der Methode RenewGPSData in Member-Variablen der LocationTimeService-Klasse zwischengespeichert. Ein Aufruf der Getter-Methoden führt nur dazu, dass diese zwischengespeicherten Werte ausgegeben werden.

Da mit dem GPS auch Zeitinformationen übertragen werden, werden diese genutzt, um die aktuelle Zeit auf dem System verfügbar zu machen. Dazu erhält die LocationTimeService-Klasse zusätzliche Methoden um die Hardware-Timer 1 und 2 zu konfigurieren und um die Millisekunden seit dem 1.1.1970 abzurufen. Diese Zeit wird in der LocationTimeService-Klasse als Membervariable zwischengespeichert.

Um die GPS-Information zur „Unix-Epoch“ umzuwandeln wird auf Funktionen der Header-Datei „time.h“ zurückgegriffen, welche in der Arduino Header Sammlung enthalten ist. Allerdings muss während der Konversion der Wert 946684800 hinzuaddiert werden, da Arduino die Epoch seit dem 1.1.2000 rechnet und der

genannte Wert den Sekunden zwischen 1.1.1970 und 1.1.2000 entspricht. Bei der Rückgabe der Millisekunden muss darauf geachtet werden, dass ein Datentyp mit 64 Bit verwendet wird und auch keine impliziten Umwandlungen bei der Berechnung auftreten.

In diesem Zuge wird Timer 1 mit global sichtbaren Funktionen und einem Interrupt so konfiguriert, um das Logging-Intervall von 200 ms einzuhalten. Timer 2 wird ähnlich konfiguriert, sorgt aber dafür, dass der zwischengespeicherte Epoch-Wert alle 8 Millisekunden um diesen Wert erhöht wird. Dadurch muss nicht jedes mal die GPS-Zeit abgerufen werden, wenn der Zeitstempel benötigt wird.

4.3.2 OBD-Schnittstelle

Da die Kommunikation mit der OBD-Schnittstelle ebenfalls über den Coprozessor läuft, von welchem die Software nicht bekannt ist, wurde hier die von Freematics bereitgestellte Klasse „COBDSPI“ verwendet. Diese stellt gewisse Funktionen wie z.B. Auslesen einer PID oder der Vehicle Identification Number (VIN). Da diese Funktionen allerdings komplizierter zu verwenden sind, wurde auch hierfür auf der Ebene der Intermediate-Layer eine neue Klasse erzeugt mit der es einfacher ist die PID-Kategorien auszulesen und für die weitere Verarbeitung zu verpacken.
Da bei OBD zwar der Stecker spezifiziert ist, es allerdings trotzdem unterschiedliche Protokolle gibt welche sich durch die Belegung der Pins unterscheiden, sorgt die Implementierung außerdem dafür, dass das passende Protokoll gefunden und für spätere Neuverbindungen gespeichert wird.
Die neue Klasse stellt weiterhin eine Methode zu Verfügung mit der es möglich ist zu erkennen, ob beim Auto die Zündung getätigt wurde. Dies ist wichtig, da die OBD-Schnittstelle (bei den meisten Autos) dauerhaft mit Strom versorgt ist. Aus diesem Grund kann nicht auf einen Neustart des Fahrzeugs gewartet werden um zu entscheiden ob es sich um eine neue Strecke handelt.
Die Funktion überprüft wie viele Timeouts beim Versuch PIDs auszulesen auftreten. Sobald diese Anzahl einen Grenzwert überschreitet, wird davon Ausgegangen, dass die Zündung nicht mehr aktiviert ist. Anschließend kann in Regelmäßigen Abständen geprüft werden ob inzwischen wieder Kommunikation mit dem OBD-

Steuergerät möglich ist.

4.3.3 Beschleunigungssensor

Der Treiber für den im Dongle verbauten Beschleunigungs-Sensor wurde nicht von Freematics übernommen sondern in Anlehnung an diesen neu Implementiert. Dies geschah vor allem um die Einheit der aufgezeichneten Sensorwerte selbst zu definieren und verständlicher darzustellen, sowie um Platz auf dem Flash-Speicher zu sparen.

Die AccReader-Klasse stellt nun Methoden zur Verfügung, welche für eine anzugebende Achse die Beschleunigung in g, die Rotation in Grad pro Sekunde und das Magnetische Feld in μ-Tesla zurückgeben. Darüber hinaus kann auch die Absolut-Beschleunigung zurückgegeben werden und die aktuellen Beschleunigungs- und Gyroskopwerte als „0“ kalibriert werden. Dabei ist zu bemerken, dass für das Gyroskop Biaswerte direkt in Register auf dem Sensor geschrieben werden können, wohingegen diese Biaswerte für den Beschleunigungssensor im RAM des Haupt-Controllers vorgehalten werden müssen.

4.3.4 Programmlogik

Die Implementierung der Programmlogik erfolgte weitestgehend nach dem in Kapitel 3.1.5 vorgestellten Abläufen. Um den Zustand des Programms während der Endlosschleife zu erfassen, wurde eine Datenstruktur definiert, welche den Modus sowie den Schleifenzähler umfasst. Somit kann das Programmverhalten durch die Veränderung einer lokal sichtbaren Datenstruktur beeinflusst werden.

Um die erfassten Beschleunigungs- und GPS-Daten genau so wie die OBD-Werte loggen zu können, mussten für diese noch eigene IDs vergeben werden. Dazu wurde der Bereich 0xF0 bis 0xFF gewählt, da diese im OBD-Protokoll nicht zur Erfassung von Fahrzeugdaten verwendet werden [**ISOobd**].

Während der Integration aller Klassen trat jedoch ein schwerwiegendes Problem auf. Wie bereits in Kapitel 3.1.5 erwähnt, traten massive Speicherprobleme

auf. Zu diesem Zeitpunkt wurden alle funktionalen Klassen erst vor ihrer Initialisierung instantiiert und die Persistenzklasse war dabei als Letztes vorgesehen. Allerdings zeigte der Dongle bei der Initialisierung der Persistenz-Klasse ein undefinierbares Verhalten mit sporadischen Software-Abstürzen. Dies legte eine Knappheit von RAM nahe. Mit der von Bill Earl vorgestellten Funktion „freeMemory()“ [**ardRAMcons**] konnte nachgewiesen werden, dass zum Öffnen bzw. Erstellen einer Datei auf der SD-Karte mindestens 384 Byte im RAM verfügbar sein müssen. Allerdings wurde bei der Analyse des Speicherbedarfes auch ersichtlich, dass zum Schreiben in die bereits geöffnete Datei weniger Platz auf dem Heap benötigt wird.

Um diesem Problem entgegenzuwirken wurden mehrere Maßnahmen getroffen. Zunächst wurden alle Strings, sofern diese auch wirklich benötigt werden, mithilfe des Makros „F()“ auf den Programmspeicher im Flash des μ-Controllers ausgelagert und mehrere Funktionen als inline-Funktionen deklariert [**ardRAMopt**]. Damit soll der Stack und damit der gesamte RAM-Verbrauch optimiert werden. Darüber hinaus wurden die meisten funktionalen Klassen nun nicht mehr mit dem new-Operator instantiiert, sondern als global verfügbare Objekte geführt. Dies stellt einen Versuch dar, der Speicherfragmentierung bei der dynamischen Instantiierung entgegenzuwirken. Auch wird nun die Initialisierung der Persistenzklasse sofort nach der Initialisierung der LocationTimeService-Klasse durchgeführt, da zu diesem Zeitpunkt einige der verbliebenen, dynamisch allokierten Klassen noch nicht instantiiert sind und somit mehr RAM zur Verfügung steht. Ebenfalls erfolgt die Speicherung der abgefragten OBD-Werte nicht mehr in einem Array, welches groß genug ist für alle Werte. Dieses Array wurde nun so verkleinert, dass es nur noch für alle PIDs der umfangreichsten Logging-Kategorie (je nach Fahrzeug entweder Kategorie A oder Kategorie B) ausreicht. Dabei müssen allerdings die alten PID-Werte bei jedem Wechsel der Logging-Kategorie gelöscht werden.

5 Tests

5.1 Dongle

Nach der erfolgreichen Integration der Dongle-Software erfolgte die Testphase, bei der der Dongle in iterativen Durchgängen an die OBD-Buchse verschiedener PKWs angeschlossen wurde. Hierbei wurden mehrere Fehlfunktionen entdeckt und behoben.

Es wurde zunächst deutlich, dass die Dauer der einzelnen Durchgänge der Haupt-Schleife weit über das erwartete Maß hinaus geht. Hierbei wurden die Zeitstempel zweier geloggter OBD-Werte mit der PID 0x0C verglichen. Dieser Wert ist der erste, der in jedem Durchgang erfasst wird. Folgende Werte zeigen beispielhaft die Dauern von Durchgänge bei denen alle Logging-Kategorien mindestens einmal erfasst wurden: Eine Analyse zeigte auf, dass diese hohen Zeiten vor allem durch die Latenz bei der Abfrage der OBD-Daten vom Steuergerät zustande kommen. Dieses Verhalten kann nicht beeinflusst werden, stellt allerdings nur ein kleines

Tabelle 5.1: Beispiel-Zeiten für einzelne Durchgänge der Haupt-Schleife

Durchgang	Dauer in Millisekunden	erfasste Logging-Kategorien
0	488	A
1	708	A + B
2	960	A + C
3	608	A + D
4	656	A + B

Problem dar. Da zu jedem erfassten Wert auch ein Zeitstempel mit einer Genauigkeit von ±8 Millisekunden gespeichert wird, ist die Einhaltung der angestrebten 500 Millisekunden für spätere Berechnungen nicht notwendig. Die Unterschiedliche Frequenz mit der die Logging-Klassen erfasst werden, welche mittels der Modulo-Operationen auf dem Loop-Counter erreicht wird, bleibt aufgrund der geringen Änderungsrate der Werte jedoch bestehen.

Darüber hinaus zeigten die Tests, dass die Verwendung des LowPowerModes des Coprozessors nicht den Erwartungen entsprach. Der STM32 konnte zwar in einen Schlafzustand versetzt werden, die Reaktivierung des selben konnte jedoch trotz intensiver Recherche im veröffentlichten Code und im offiziellen Freematics Forum nicht erreicht werden. Es zeigte sich allerdings, dass die verwendeten Funktionen aus den aktuelleren Versionen der offiziellen Softwarebibliotheken entfernt wurden [**freematicsRevFeb**]. Da bereits ein Timeout bei der Abfrage von OBD-Werten dazu führt, dass die Dongle-Software ein Abstellen des Fahrzeuges vermutet, folgte daraus ein zufälliges Abbrechen der Aufzeichnung während einer laufenden Fahrt. Zur Beseitigung des Problems wurde auf die Verwendung des LowPowerModes verzichtet. Dieses Problem stellt im aktuellen Projektstand eine Verbesserungsmöglichkeit dar.

6 Ausblick

Aufgrund der beschränken Ressourcen die für dieses Projekt zur Verfügung standen, konnte das Potential des zuvor vorgestellten Systems bei weitem noch nicht ausgeschöpft werden. Aus diesem Grund, soll das folgende Kapitel Ideen vermitteln was zum Beispiel mit der Infrastruktur erreicht werden kann.

6.1 Identifikation des Fahrers

Mithilfe der aufgezeichneten Beschleunigungsdaten wäre es möglich die Infrastruktur dementsprechend zu erweitern, dass aus den aufgezeichneten Daten personalisierte Fahrtenbücher auch ohne die Notwendigkeit für ID Karten oder sonstige Mittel zur Identifikation des Fahrers erstellt werden können. Dies würde vermutlich funktionieren, da jeder Fahrer seinen eigenen persönlichen Fahrstil besitzt der z.B. das Anfahren und Abbremsen beeinflusst.
Diese Möglichkeit der Identifikation hätte den Vorteil, dass auch nachträglich festgestellt werden kann wer der Fahrer war.

6.2 Bewertung des Fahrverhaltens

Ebenso wäre es mit den aufgezeichneten Daten möglich den Fahrstil der Fahrer zu bewerten. Wird beispielsweise an der Ampel mit dem Gas gespielt oder der Drehzahlbereich jedes mal bis zum roten Bereich ausgelastet?
Diese Daten wären zum einen z.B. für Versicherungen oder Firmen interessant, können allerdings auch Privatpersonen helfen z.B. herauszufinden wer der anderen Mitfahrer für den schnelleren Verschleiß von Reifen oder anderer Teile zuständig sein könnte.

6.3 Ermittlung von Engpässen

Alleine mit den aufgezeichneten GPS-Daten ist es möglich Engpässe im Straßennetz zu erkennen und dagegen entsprechende Gegenmaßnahmen in die Wege zu leiten (z.B. Ausbau von Straßen, Ampelanlagen zeitlich anders schalten, ...).
Weiterhin wäre eine solche Analyse z.B. auch für Firmen hilfreich. Dies würde nicht dazu führen, dass das Straßennetz geändert würde, aber die Fahrer könnten alternative Routen vorgeschlagen bekommen welche weniger frequentierte Strecken nutzen.
Jede dieser Möglichkeiten würde auf Dauer dazu führen, dass sich die Verkehrslage entspannen kann.

6.4 Erstellung von Spritsparrouten

Die in Kapitel 6.3 vorgestellten Ideen könnten weiterhin verwendet werden um Routen zu finden bei denen der Fahrer aufgrund von weniger Stop and Go Verkehr Sprit einsparen kann.

6.5 ??Softwareoptimierungen??

Wie bereits in Kapitel 5.1 aufgezeigt wurde, besteht immer noch Potential, um die den Dongle auf Energiesparmöglichkeiten zu untersuchen und diese zu implementieren.

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

GPS	Global Positioning System
MEMS	Microelectromechanical System
NMEA	National Marine Electronics Association
OBD	On Board Diagnistic
PID	Parameter Identifikator
SBAS	Satellite-based augmentation systems
UART	Universal asynchronous receiver-transmitter
VIN	Vehicle Identification Number