



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: 11. April 2018

Inhaltsverzeichnis

1	Einleitung	2
1.1	Verfügbare Lösungen	2
1.2	Zielsetzung	4
2	Use Cases	5
3	System Design	6
3.1	Dongle	6
3.1.1	Programmlogik	6
3.1.2	GPS-Empfänger	9
3.1.3	Zeit	12
3.1.4	Beschleunigungssensor	13
4	Implementierung	14
4.1	App	14
4.2	Backend	14
4.3	Dongle	14
4.3.1	Programmlogik	14
4.3.2	GPS und Zeit	14
4.3.3	Beschleunigungssensor	16
5	Tests	18
6	Ausblick	19

1 Einleitung

Ein zu beobachtender Trend ist die zunehmende Digitalisierung verschiedenster Lebensbereiche mit dem Ziel, die Effizienz 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 Telemetriedaten die unter anderem zukünftig mit anderen Fahrzeugen oder Serviceanbietern geteilt werden. Neben den Vorteilen für andere Verkehrsteilnehmer, durch verbesserte Unfallvermeidung oder Stauererkennung, 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 aufsetzen.

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, so 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 Programmlogik

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

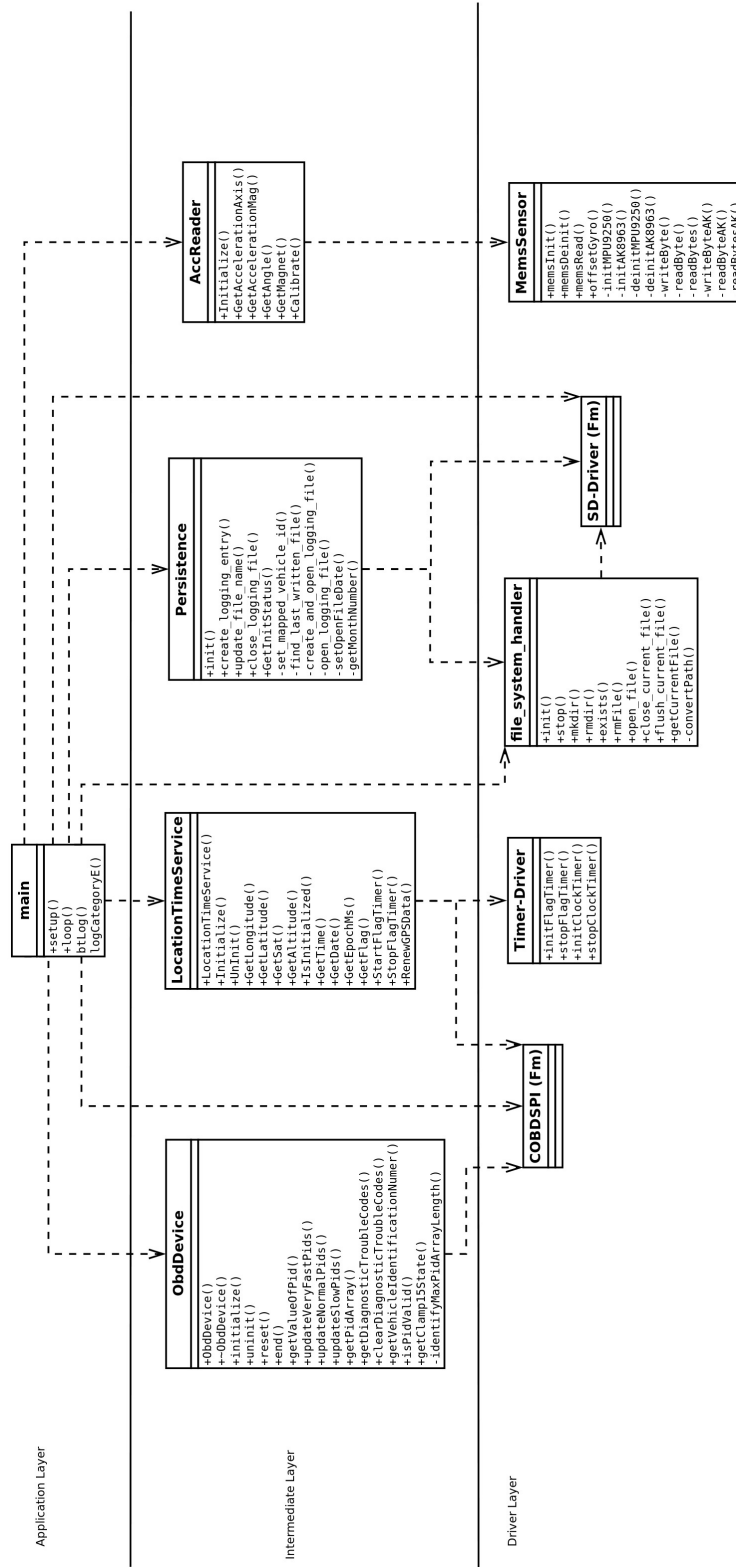


Abbildung 3.1: Architektur der Dongle-Software

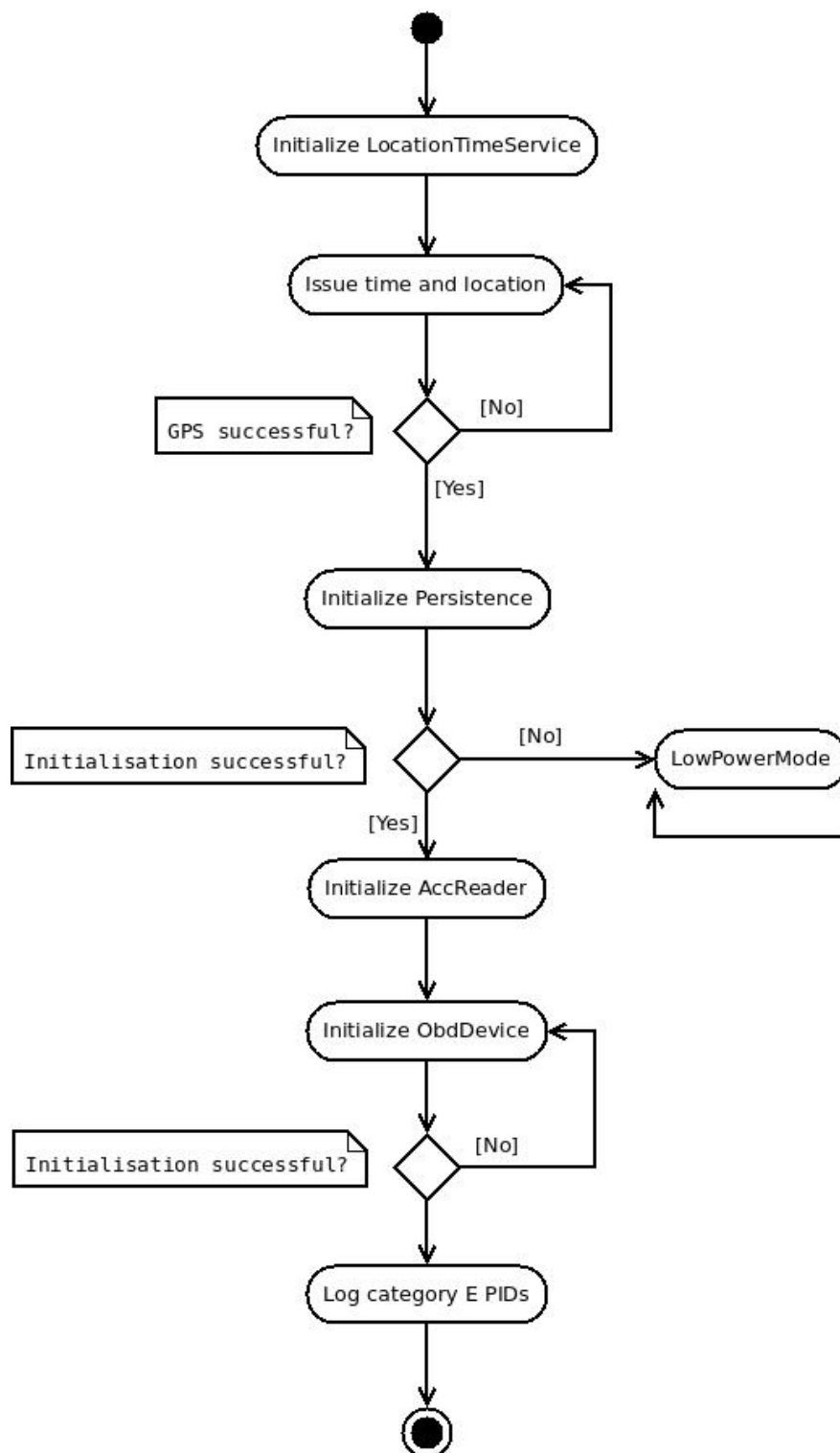


Abbildung 3.2: Programmablauf der Initialisierung

Die Abbildung 3.2 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. Während der Integration zeigte sich, dass die Persistenz-Klasse zur Initialisierung wesentlich mehr RAM benötigt als während des normalen Programmablaufs. Deshalb muss diese so früh wie möglich initialisiert werden. Nur die Klasse für GPS und Zeit muss zuvor bereit sein, da die Persistenzklasse diese während der Initialisierung aufruft. In Abbildung 3.3 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.

3.1.2 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 die Möglichkeit, einen externen GPS-Empfänger per UART 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 Produkt-

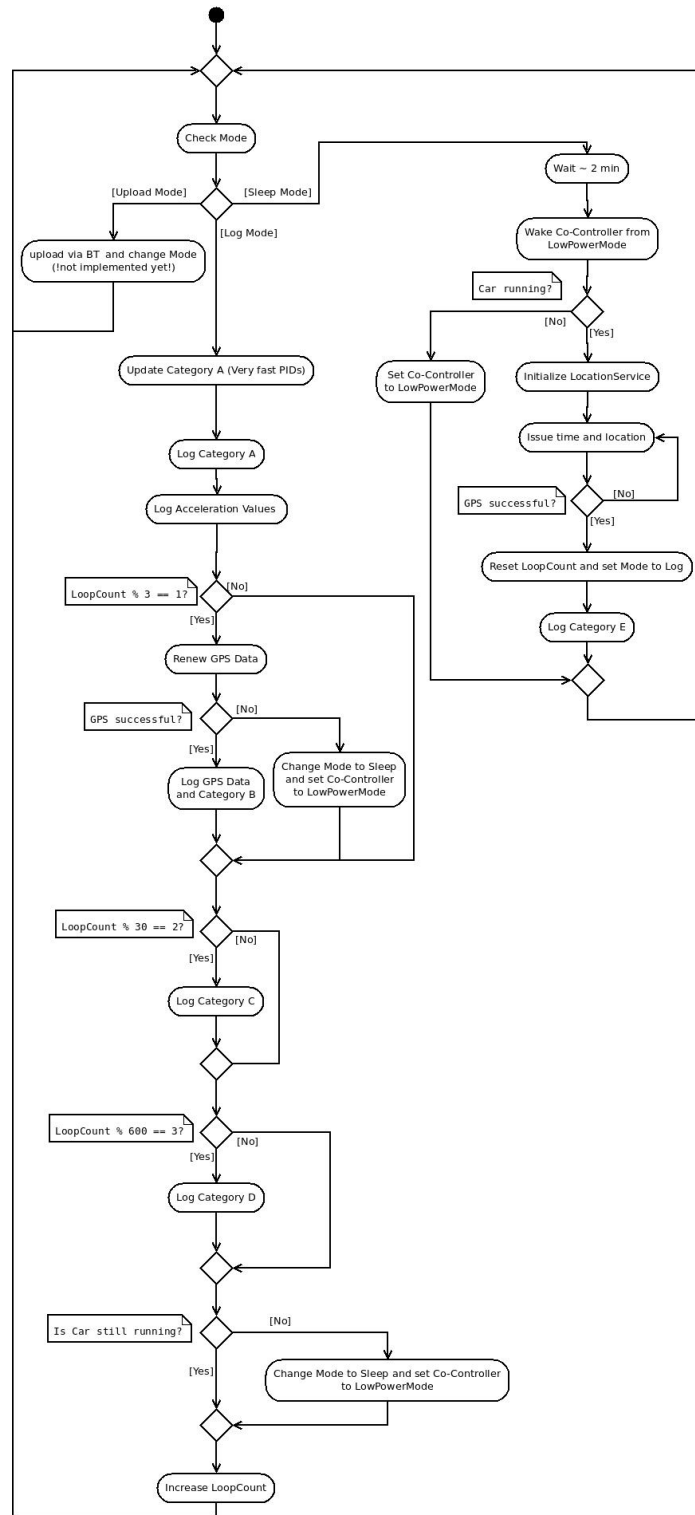


Abbildung 3.3: Programmablauf der Endlosschleife

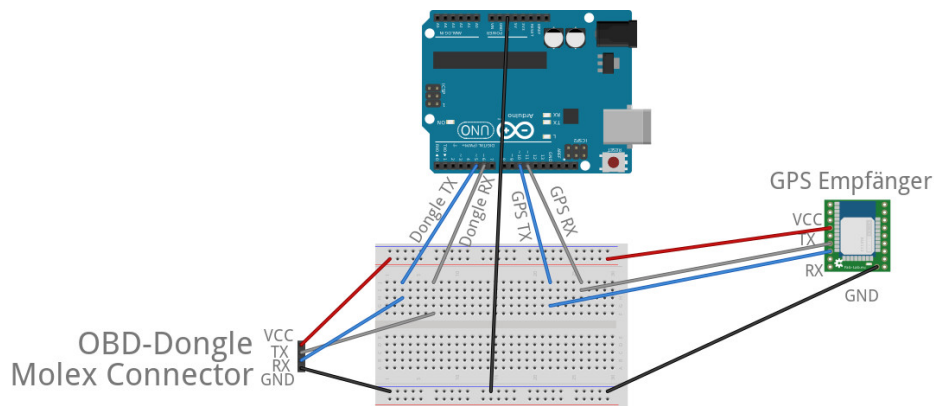


Abbildung 3.4: Versuchsaufbau zur Analyse der UART-Kommunikation zwischen Dongle und GPS-Empfänger

homepage 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 3.4 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 NMEA Standard. Darüber hinaus wurde ersichtlich, dass der OBD-Dongle keine Nachrichten zum GPS-Chip sendet.

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-Quadroptopter 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 Freemantics 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.

3.1.3 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 200 Millisekunden erfasst werden.

Die Anforderung nach einem genauen Zeitintervall von 200 Millisekunden zwischen dem Abrufen der OBD-Werte der Klasse 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-Klassen B, C und D ein Vielfaches der 200 Millisekunden der Klasse 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 3.3).

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 MEMS Bewegungssensor mit jeweils 3 Achsen für Beschleunigungs-, Drehraten- und Magnetfeldmessung verbaut ist. Hierbei ist besonders, dass der Sensor für das Magnetfeld als I²C-Submodul am Sensor ausgeführt ist.

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

Was kann das Backend, und besonderes erklären

4.3 Dongle

Was kann der Dongle, und besonderes erklären

4.3.1 Programmlogik

4.3.2 GPS und Zeit

Um den GPS-Empfänger auch in der in diesem Projekt geschriebenen Dongle-Software zu nutzen, wurde zunächst der Code des von Freemantics 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 geographischen Längen- und Breitengrads, der Höhe über Normalnull und die Anzahl der verfügbaren Satelliten. 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.

4.3.3 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.

5 Tests

Was haben wir getestet, fahren aufgenommen und verifiziert....

6 Ausblick

Was kann man jetzt mit unserem Zeug als Grundlage anstellen, was haben wir uns überlegt aber konnten es nicht mehr umsetzen.

Abbildungsverzeichnis

3.1	Architektur der Dongle-Software	7
3.2	Programmablauf der Initialisierung	8
3.3	Programmablauf der Endlosschleife	10
3.4	Versuchsaufbau zur Analyse der UART-Kommunikation zwischen Dongle und GPS-Empfänger	11

Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

HKIWS Hier könnte ihre Werbung stehen