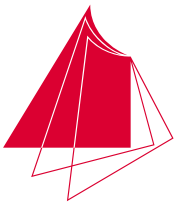


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

EIT Fakultät für Elektro- und Informationstechnik

Hochschule Karlsruhe - Technik und Wirtschaft
Fakultät für Elektro- und Informationstechnik

Masterthesis

Diagnose und Messwerterfassung für Prüfstände

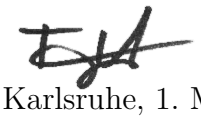
von
Thomas Wagner

Matrikel-Nr.: 65923
Thesis-Nr.: 370

Referent:	Prof. Dr.-Ing. Philipp Nenninger
Korreferent:	Prof. Dr. rer. nat. Klaus Wolfrum
Arbeitsplatz:	PA-Systems GmbH & Co. KG
Betreuer am Arbeitsplatz:	Dipl.-Ing. Oliver Bindschädel Dipl.-Ing. Jürgen Adam
Zeitraum:	01.09.2020 - 01.03.2021
Version vom:	18. Februar 2021

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.



Karlsruhe, 1. März 2021

Zusammenfassung

Kraftfahrzeuge werden immer komplexer. Die resultierende steigende Packungsdichte bedeutet schwer zu erreichende Messpunkte. Anforderungen an Emissionsmessungen werden vielfältiger und neue Messgrößen für elektrifizierte Fahrzeuge kommen hinzu. Deshalb steigt die Anzahl der nötigen externen Messmittel. Es ergibt sich eine Herausforderung für Prüfstände, wobei zusätzlichen Messgeräten kein Platz im Fahrzeug zur Verfügung steht, um ihre Messwerte abzugreifen.

Viele der Messgrößen sind jedoch fahrzeugintern bereits verfügbar und müssen nicht zwangsläufig durch zusätzliche Messmittel bestimmt werden. Diese Arbeit bespricht die Möglichkeit, solche Messwerte über Diagnoseprotokolle wie OBD und UDS als kontinuierliche Messschriebe zu erfassen.

Es wird eine Toolkette erarbeitet, die ein eingebettetes Linux-System mit CAN-Controller zum präzisen Polling von Messwerten nutzt. Ein Add-in für eine etablierte Automatisierungssoftware wird entwickelt, welches das Linux-System steuert und dessen Bedienung automatisiert.

Abstract

Motor vehicles are becoming more and more complex. The resulting packing density increases, which causes measuring points to be hard to reach. Demands for emission related measurements are becoming more diverse and new metrics for electrified vehicles are added. Hence, the required number of external measuring devices is increasing. This poses a challenge for test benches, where there is no space in the vehicle to acquire the additional readings.

However, many of the measured values are already available in the vehicle's board electronics and do not necessarily have to be determined by additional measuring equipment. This thesis discusses the possibility of continuously recording such readings using diagnostic protocols such as OBD and UDS.

A tool chain is being designed that uses an embedded Linux system with a CAN controller for precise polling of measurement values. To control the Linux system and automate its operation, an add-in for an established automation software is being developed.

Inhaltsverzeichnis

1 Einleitung

Dieses Kapitel dient zur Einleitung dieser Arbeit. In Kapitel 1.1 wird das Umfeld beschrieben, indem diese Arbeit angefertigt wurde. Die Kapitel 1.2 und 1.3 beschreiben die Motivation hinter dieser Arbeit und in welcher Aufgabenstellung sich diese äußert.

1.1 Arbeitsumfeld

Die Firma PA-Systems GmbH & Co. KG entwickelt Softwarelösungen für Prüfstände in der Automobilindustrie. Sie hat ihren Sitz in Karlsruhe. Die angebotenen Softwarelösungen sind für Prüfstände in der Entwicklung, in Zulassungstests und in stichprobenartigen Prüfungen während der Serienfertigung, End Of Line (EOL)-Tests, im Einsatz. Das modulare Automatisierungstool Multi Process Automation System (MPAS) (siehe im folgenden Kapitel 2.5) ist ein Beispiel hierfür.

1.2 Motivation

Kraftfahrzeuge werden immer komplexer. Die resultierende steigende Packungsdichte bedeutet schwer zu erreichende Messpunkte. Aktuelle Gesetzesgebungen in der Automobilbranche sind umfangreich und variieren von Land zu Land. Prüfstände und deren Automatisierung müssen diesem Wandel gerecht werden. Es ist nicht mehr ausreichend Bewertungen allein anhand von Rollenprüfstands- und Abgasmessungen durchzuführen. Die zunehmende Elektrifizierung erfordert zusätzliche Messgrößen. In den immer kompakterer gebauten Innenleben der Fahrzeuge sind diese aber mitunter schwierig mit externen Messmitteln zu erfassen, da der Platz fehlt und größere Modifizierungen durch die Gesetzgebung verboten sind. Viele der relevanten Größen sind bereits fahrzeugintern auf den Steuergeräten erfasst. Eine Alternative wäre es diese dort abzugreifen, anstatt externe Sensoren zu benutzen, die aufwendig angebracht werden müssen.

1.3 Aufgabenstellung

Es gilt eine Toolkette zu entwickeln, die es ermöglicht Mess- und Diagnosewerte zu erfassen. Die Lösung ist über ein Add-in in die Hauptsoftware, MPAS, zu integrieren wie in Abbildung 1.1 angedeutet.

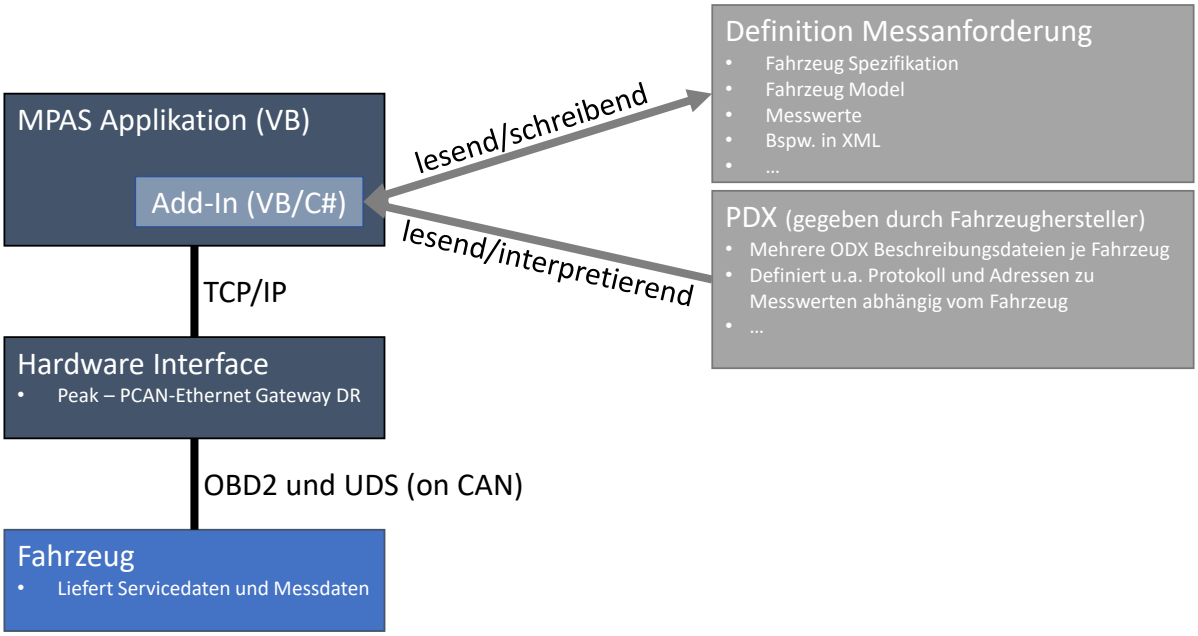


Abbildung 1.1: Skizze der Toolkette

Das Add-in kann diverse Interfaces der MPAS-Software nutzen und ist dank der Flexibilität des .Net-Frameworks in Visual Basic (VB) oder C# zu entwickeln. Neben der Interaktion mit der Hauptsoftware muss das Add-in Messanforderungen in Dateien speichern bzw. daraus laden. Die Messanforderungen in Kombination mit den, aus sogenannten Packaged ODX (PDX)-Archiven gewonnenen, Beschreibungsdateien ergibt die nötigen Informationen zum Steuergerätezugriff.

Auf Basis bereits etablierter Hardware (Peak System PCAN-Ethernet Gateway DR[1]) ist ein Interface zu konzipieren und umzusetzen. Das Interface muss auf der Fahrzeugseite OBD und UDS über CAN unterstützen. Die Kommunikation des Hardware Interface zur MPAS-Software soll über TCP/IP laufen.

Besonderer Augenmerk liegt auf der Robustheit und der Stabilität der zu entwickelnden Toolkette. Da Prüfstände auf Basis der MPAS-Software meist im 24h-Betrieb sind, muss ein reibungsloser und unterbrechungsfreier Ablauf garantiert sein. Anwender ohne Hintergrundkenntnisse dürfen nicht auf unerwartete Systemzustände oder Fehler stoßen, die sie nicht beheben können bzw. die nicht klar nachvollziehbar sind. Ein robustes Fehlerhandling muss dafür sorgen, dass der Test auch bei unerwarteten Fehlern unterbrechungsfrei weiter laufen kann.

[1] https://www.peak-system.com/PCAN-Ethernet-Gateway-DR.330.0.html

1.4 Aufbau der Arbeit

Praktische Softwareentwicklung folgt keiner klaren Linie wie z.B. im bekannten V-Modell [Boe84] beschrieben. Heutzutage wird oft von agiler Softwareentwicklung gesprochen. Die Definition von agiler Entwicklung ist allerdings, vor allem in der praktischen Anwendung, genauso schwammig, wie die Art und Weise in der sich oft an entsprechende Methoden wie SCRUM [Sch04] gehalten wird. [Kha20, Cla20]

Der Aufbau dieser Arbeit wird sich deshalb grob an das klassische V-Modell halten, hier und da aber auch Ausreißer aufzeigen. Laut dem V-Modell folgt auf die Konzeption (Kapitel 4) stets die Umsetzung (Kapitel 5) und dann diverse Tests und Validation (Kapitel 6). Was passiert aber, wenn während der Validation ein Fehler auftritt oder während der Umsetzung bereits Fehler auffallen? Realistisch betrachtet wird je nach Größe des Fehlers erneut in die Umsetzung gesprungen, um den Fehler zu beheben, wobei es ggf. vorab eine neue, teilweise, Konzeption gibt. Spezielle Kapitel zu Besonderheiten und Auffälligkeiten sollen diese zusätzlichen kleinen Iterationen in dieser Arbeit widerspiegeln. Das Ziel dieses Vorgehens ist die transparente Wiedergabe der Arbeit anstatt eines Rückblicks auf eine scheinbar völlig reibungslose Umsetzung, die dennoch zum selben Ergebnis kommt.

2 Grundlagen

In diesem Kapitel werden theoretische Grundlagen erläutert, die für das Verständnis der weiteren Arbeit relevant sind. Dabei handelt es sich vor allem um verschiedene Protokolle (Kapitel 2.2 und 2.3) sowie Softwaretools (Kapitel 2.5 und 2.6). Gleich zu Beginn wird auf das OSI-Schichtenmodell (Kapitel 2.1) eingegangen, das hilft, die verschiedenen Protokolle zueinander einzuordnen. Einige spezifische Grundlagen werden als Einschübe in späteren Kapiteln erläutert, sobald sie relevant werden.

2.1 ISO/OSI-Schichtenmodell

Die International Standards Organization (ISO) Norm 7498 entstand 1983 unter dem Namen „Basic Reference Model for Open Systems Interconnection (OSI)“ [Gen17]. Sie beschreibt ein Referenzmodell, um die verschiedenen Aufgaben bei der Kommunikation zwischen Systemen in sieben Schichten einzuteilen. Das Modell ermöglicht es verschiedene Kommunikationsprotokolle einer oder mehreren Schichten zuzuordnen. Anhand der Einordnung lässt sich nicht nur schnell klar darstellen, welche Aufgaben das jeweilige Protokoll erfüllt, sondern auch mit welchen Protokollen es vergleichbar oder sogar austauschbar ist.

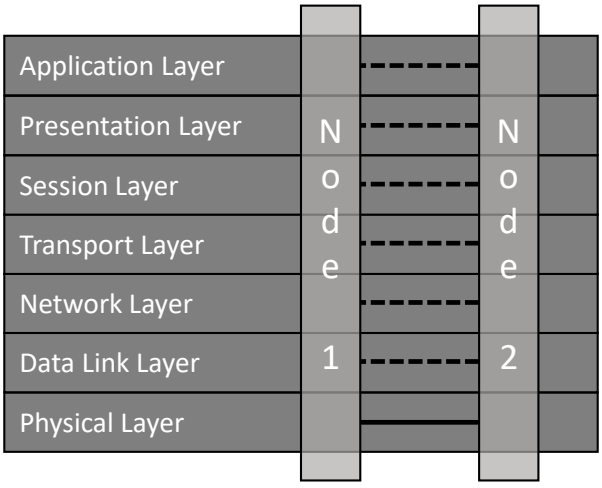


Abbildung 2.1: ISO/OSI-Schichtenmodell

Kommunikationspartner müssen auf beiden Seiten identische Protokolle für die genutzten Schichten bereitstellen. So steht jeder Schicht der entsprechende Gegenpart im

Kommunikationspartner gegenüber. Dieses Verhältnis zwischen den Kommunikationspartnern/Nodes ist in Abbildung 2.1 dargestellt.

Die sieben Schichten beginnen auf der untersten, und somit ersten, Ebene. Prinzipiell steigt die Komplexität und der Abstraktionsgrad der Kommunikation und der Protokolle, je höher die betrachteten Schichten im Modell sind. [LO11]

1. Schicht: Bitübertragung (Physical Layer) Auf der untersten Schicht beginnt das Modell mit der Definition der Bitübertragung (*Physical Layer*). Protokolle die diese Schicht beinhalten, definieren die elektrische und mechanische Eigenschaften der Übertragung. Hier wird u.a. festgelegt, welche physikalischen Medien, welche Bitkodierung und welche Anzahl an physikalischen Verbindungen es gibt. T1000BASE-T, als Basis von Ethernet sowie RS232 sind Beispiele von Protokollen der ersten Schicht.

2. Schicht: Sicherung (Data Link Layer) Der *Data Link Layer* des OSI/ISO-Schichtenmodells beschreibt wie Nachrichten in Rahmen/Frames geformt werden. Ein Frame beinhaltet neben den eigentlichen Nutzdaten Zusatzinformationen. Diese beschreiben u.a. Prüfsummen zur Sicherung der Daten, Anfangs- und Endkennungen sowie die Geschwindigkeitssteuerung und Flusskontrolle auf dem physikalischen Layer. Ein Beispiel für diese Schicht ist das Ethernet Protokoll, das sich über die erste und zweite Schicht erstreckt.

3. Schicht: Netzwerk (Network Layer) Im *Network Layer* geht es vor allem um die Vermittlung von Nachrichten zwischen Absender und Empfänger. Die Zuständigkeit dieser Schicht beinhaltet das Routing von Nachrichtenpaketen über mehrere Netzwerkknoten, falls keine direkte Verbindung zwischen Sender und Empfänger vorhanden ist. Dementsprechend ist diese Schicht auch für eindeutige Adressierbarkeit verantwortlich. Da es vorkommen kann, dass auf niedrigeren Schichten unterschiedliche Protokolle eingesetzt werden, kümmert sich der *Network Layer* auch um anfallende Umsetzungen. Das Internet Protokoll (IP) gehört beispielsweise zu Schicht 3.

4. Schicht: Transport (Transport Layer) Verbreitete Beispiele für die Transportschicht sind die Protokolle User Datagram Protocol (UDP) und Transmission Control Protocol (TCP). Sie segmentieren den Datenstrom und realisieren eine Stauvermeidung. In dieser Schicht können zudem Mechanismen zur fehlerfreien Übertragung umgesetzt sein. Im Falle von UDP und TCP wird auch eine zusätzliche Schicht-4-Adresse, ein sogenannter Port, vergeben. Der *Transport Layer* ist der letzte Layer vor den anwendungsorientierten Layern. Deshalb muss der Transport Layer sicherstellen, dass darüber liegenden Layern ein Kommunikationskanal zur Verfügung steht, dessen genaue Eigenschaften sie nicht berücksichtigen müssen.

5. Schicht: Sitzung (Session Layer) Die Sitzungsschicht ist die erste der anwendungsorientierten Schichten. Oft werden die Schichten 5, 6 und 7 in einem Protokoll realisiert. Dabei ist es meist der Fall, dass nur wenige der Funktionalitäten, der drei Schichten, explizit realisiert sind. Der *Session Layer* hat z.B. die Aufgabe die darunterliegende Kommunikation mehreren Prozessen gleichzeitig/parallel zur Verfügung zu stellen. Explizite Schicht 5 Protokolle wie Remote Procedure Call (RPC) setzen zudem Funktionalitäten um, die abgebrochene Verbindungen wieder synchronisieren können.

6. Schicht: Darstellung (Presentation Layer) In der Darstellungsschicht (*Presentation Layer*) wird sichergestellt, dass die Nutzdaten einer Nachricht auf allen Kommunikationsteilnehmern identisch interpretiert werden. Strings können ASCII oder UTF codiert sein und Zahlenwerte könnten als Ganz- oder Gleitkommazahlen interpretiert werden. Unterschiedliche Teile der Nachricht, die z.B. verschiedene Messwerte beschreiben, müssen für alle Teilnehmer verständlich getrennt sein. Falls Verschlüsselungen genutzt werden, sind diese in Schicht 6 festgelegt. Das Hypertext Transfer Protocol (HTTP) ist ein verbreitetes Beispiel für die Schichten 5 bis 7. Ein HTTP-Header beschreibt z.B. in einem `Content-Type`-Feld, als was die Nutzdaten zu interpretieren sind.

7. Schicht: Anwendung (Application Layer) Die Anwendungsschicht (*Application Layer*) ist der letzte und oberste Layer des ISO/OSI-Schichtenmodells. Diese Schicht ist die direkte Anbindung an die Anwendung, der diese Schicht die Kommunikationsfunktionen zur Verfügung stellt.

Das ISO/OSI-Schichtenmodell ist lediglich eine Referenz um Protokolle einzuordnen. Wie bereits erwähnt sind viele Protokolle mehreren Schichten zuordenbar und das nicht immer absolut eindeutig. Im Falle von TCP/IP wird deshalb das Schichtmodell zu vier Schichten vereinfacht/zusammengefasst. Bitübertragung und Sicherung werden als *Netzzugang* bezeichnet. Die *Netzwerk-* und *Transportschichten* dürfen ihren Namen behalten. Die obersten drei Schichten werden in die *Anwendungsschicht* zusammengefasst. Dieses TCP/IP-Referenzmodell ist in dem RFC 1122 [Net89] beschrieben.

Oft kommt es zudem vor, dass Kommunikationsstacks nur einige der niederen Schichten nutzen und eine Anwendungsschicht jeglichen nötigen Abstraktionsgrad auf höherer Ebene übernimmt. Gerade im Bereich von Feldbussen in Fahrzeugen oder auch in der Industrie ist dies der Fall. [BBB+94]

2.2 CAN

Der CAN-Bus (Controller Area Network) entstand Anfang der 80er Jahre. Die Elektronik in Fahrzeugen wurde immer komplexer und so auch der Installationsaufwand in der Produktion. Aufwendige und vor allem auch schwere Kabelbäume waren ein Problem. Viele Hersteller haben sich deshalb darum bemüht Kommunikationsstandards zu

entwickeln, die Busse nutzen, um die Kabelbäume wieder beherrschbarer zu machen. CAN wurde von Bosch entwickelt und war neben LIN und MOST eines der Bussysteme, die sich aus den neuen Anforderungen herausgebildet haben. Seit Anfang der 90er Jahre ist CAN in Serienfahrzeugen anzufinden. CAN ist für Klasse-C-Anwendungen entwickelt. Klasse-C-Anwendungen brauchen weniger Bandbreite als Klasse-D-Anwendungen, haben aber oft echtzeitkritische Anforderungen wie z.B. bei der Motorsteuerung. Ein Klasse-D-Bus ist z.B. der MOST-Bus (Media Oriented Systems Transport), der für Anwendungen mit höherer Bandbreite und höheren Latenzen im Multimediabereich geeignet ist. Netzwerke die weder allzu hohe Bandbreiten noch geringe Latenzen erfordern, wie in der Spiegel- oder Sitzsteuerung, setzen auf Klasse-A-Busse wie Local Interconnect Network (LIN). [LO11]

Obwohl CAN in Klasse-C-Anwendungen, die teilweise echtzeitkritisch sind, eingesetzt wird, nutzt es kein deterministisches Medienzugriffsverfahren. CAN gehört zu den Bussystemen mit zufälligem Medienzugriffsverfahren, Carrier Sense Multiple Access (CSMA), und setzt auf die Vermeidung von Kollisionen auf dem Bus, Collision Avoidance (CA). CSMA/CA wird bei CAN dadurch realisiert, dass vor dem Senden geklärt wird, welcher Teilnehmer senden darf. Hierzu ist ein dominanter Buspegel definiert, der Teilnehmer, die versuchen den rezessiven Buspegel zu schreiben, aussticht. Eine genauere Erläuterung folgt im Kapitel 2.2.1.2. Dieses Verfahren führt dazu, dass CAN nicht hart echtzeitfähig ist. Ein richtig geplantes CAN-Netz kann aber den Anforderungen weicher Echtzeitfähigkeit entsprechen.

CAN ist vor allem in den Versionen CAN 2.0A und CAN 2.0B verbreitet, die sich durch die Länge des Identifier-Feldes unterscheiden, vgl. Kapitel 2.2.1.1. Die verschiedenen CAN-Varianten, auch im Bezug auf die verschiedenen Bitraten bis zu 1000 kbit/s, sind in der ISO 11898 festgehalten. Typisch wird eine Bitrate von 500 kbit/s genutzt. CAN kann Daten von bis zu 8 B (Byte) pro Frame übertragen. Im Bezug auf das ISO/OSI-Schichtenmodell ist CAN in die 1. und 2. Schicht einzuordnen.

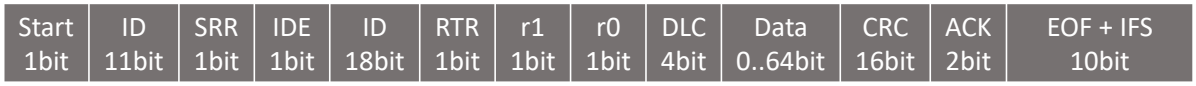
2.2.1 Aufbau

Als Medium nutzt CAN ein Paar aus Drähten zur differenziellen Übertragung. Bei höheren Übertragungsraten als Twisted Pair ausgeführt. Die Leitungen sind normal als CAN-High und CAN-Low bezeichnet. Es gibt auch Lösungen, die mit Lichtwellenleiter (LWL) umgesetzt sind. In CAN-Netzwerken wird inhaltsbezogen adressiert. Das heißt Busteilnehmer haben keine direkte Adresse, auf die sich bezogen wird, sondern Nachrichten haben Identifier. Sind Busteilnehmer also an der Nachricht `ist_gaspedalstellung`, die z.B. den Identifier `0x005` hat, interessiert, müssen sie nichts über den verantwortlichen Sender wissen. Es reicht die ID zu kennen, die zum jeweiligen Nachrichtentyp gehört.

2.2.1.1 Nachrichtenaufbau

In der weiteren Beschreibung des CAN-Busses wird der rezessive Pegel mit logisch „1“ oder `High` gleichgesetzt. Der dominante Pegel wird auch als „0“ oder `Low` beschrieben.

(a) CAN 2.0A mit 11bit-Identifier



(b) CAN 2.0B mit 29bit-Identifier

Abbildung 2.2: Aufbau der CAN-Datentelegramme

Abbildung 2.2 zeigt den Aufbau eines CAN-Datagramms entsprechend den Varianten CAN 2.0A und CAN 2.0B.

Start-Bit Ist immer dominant und leitet ein neues Frame ein.

ID 11 bit, die den Identifier halten, der auch zur Arbitrierung benutzt wird, vgl. Kapitel 2.2.1.2.

RTR/SRR Das RTR-Bit (Remote Transmission Request) wird auf `High` gesetzt, falls die Nachricht selbst keine Daten beinhaltet, sondern als Anfrage nach neuen Daten von der gegebenen ID dient. Im Falle von CAN 2.0B ist an dieser Stelle das SRR-Bit (Substitute Remote Request). Das SRR ist immer rezessiv und ersetzt lediglich das RTR an dieser Stelle ohne bestimmten Informationsgehalt. Für die verlängerte Version, folgt das RTR nach dem zweiten ID-Abschnitt.

IDE Das IDE-Bit (Identifier Extension) wird `High` gesetzt, um anzudeuten, dass die verlängerte ID, CAN 2.0B, genutzt wird.

r0, r1 Dies sind reservierte Bits.

DLC Die vier DLC-Bits (Data length code) enthalten die Länge der nachfolgenden Daten in Bytes.

Data Das Data-Feld beinhaltet die eigentlichen Daten des Telegramms und ist zwischen 0 B bis 8 B lang.

CRC Das CRC-Feld (Cyclic redundancy check) enthält den Fehlercode als CRC-Prüfsumme über alle vorangestellten Bits. Die Hamming-Distanz beträgt mit dem verwendeten Polynom sechs. Das Feld wird mit einem Delimiter abgeschlossen, der immer rezessiv ist.

ACK Das ACK-bit (Acknowledge) wird durch den Sender auf `High` gesetzt. Mindestens ein Teilnehmer, der die Nachricht korrekt empfangen hat, muss dieses auf den dominanten Pegel ziehen. Damit ist ein korrektes Senden anerkannt. Falls kein Teilnehmer den Pegel auf `Low` zieht, liegt ein Fehler vor. Das Feld wird mit einem Delimiter abgeschlossen, der immer rezessiv ist.

EOF und IFS Sowohl End Of Frame (EOF) als auch Interframe Space (IFS) sind durchgängig rezessiv. Die 7 bit des EOF stellen einen Verstoß gegen die Formatierung mit Padding dar, womit das Ende der Übertragung eindeutig identifiziert werden kann. Der darauf folgende IFS ist als Pause auf dem Bus und für alle Teilnehmer gedacht.

2.2.1.2 Zugriffsverfahren

Wie bereits erwähnt nutzt CAN ein CSMA/CA Verfahren. Hierzu realisiert CAN eine Arbitrierungsphase. Diese Phase beginnt für alle Teilnehmer, die daran interessiert sind eine Nachricht zu senden, mit dem Start-Bit. Jeder Sender schreibt hierbei den niedrigen Pegel, also den dominanten, auf den Bus. Nun werden die IDs auf den Bus geschrieben. Die Teilnehmer, die eine Nachricht schreiben wollen, deren ID im ersten Bit „0“ ist, überschreiben die rezessive „1“ der anderen Teilnehmer. Bemerkt ein Teilnehmer, dass seine geschriebene „1“ überschrieben wurde, scheidet er aus der Arbitrierung aus. Es folgt die nächste Runde mit dem zweiten Bit der ID. Der Teilnehmer mit der Nachricht, die die kleinste ID hat, gewinnt, da die kleinste ID in binärer Codierung die meisten führenden Nullen hat. Null bzw. logisch „0“ oder `Low` ist der dominante Pegel und überschreibt deswegen alle Anderen. Das Vorgehen bei der Arbitrierung verdeutlicht, wieso es wichtig für die CAN-Hardware ist, einen dominanten Pegel auf dem physikalischen Bus zu realisieren.

Das Busarbitrierungsverfahren von CAN hat den Effekt, dass Nachrichten mit kleiner ID eine höhere Priorität haben als jene mit größerer ID. Die Nachricht mit der ID 0x000 hat so immer höchste Priorität und für sie könnte rein theoretisch sogar harte Echtzeit gefordert sein. Gleichzeitig heißt das aber auch, dass ein stark ausgelasteter Bus, mit vielen Nachrichten mit kleiner ID, verhindern kann, dass überhaupt noch Nachrichten mit größerer ID gesendet werden können.

Der CAN-Bus ist in der elektrischen Ausführung auf einen Wellenwiderstand von 95 Ω bis 140 Ω spezifiziert. Dies ist wichtig, um vor allem während der Arbitrierungsphase Fehler durch Reflexion zu vermeiden. Zudem ist der Zusammenhang von Datenrate und Leitungslänge wichtig. Gerade während der Arbitrierung müssen auch weit voneinander entfernte Teilnehmer feststellen können, ob ihr rezessiver Pegel überschrieben wurde. Dementsprechend darf bei geringen Datenraten, bei denen ein Bit lange auf dem Bus ansteht, die Leitung und somit die Signalausbreitungszeit auch länger sein. Umgekehrt gilt bei hohen Datenraten eine kürzere maximale Buslänge.

2.2.1.3 Bit-Codierung und Fehlerhandling

Falls klassisch zwei Leitungen im Einsatz sind, werden die Bits Non Return to Zero (NRZ) codiert. Der CAN-Bus nutzt deshalb Bit-Stuffing um die Oszillatoren der Teilnehmer zu synchronisieren. Wenn es dazu kommt, dass mehr als fünf aufeinanderfolgende Bits denselben Pegel haben, wird ein zusätzliches Bit in das Frame „gestopft“, das den inversen Pegel hat. Damit ist garantiert, dass regelmäßig Flanken zur Synchronisierung vorhanden sind. Zudem ist somit das EOF, mit seinen sieben aufeinander folgenden rezessiven Bits, immer eindeutig identifizierbar.

Das Fehlerhandling soll hier nur kurz angerissen werden. Falls ein Fehler auftritt, z.B. CRC, Bit Stuffing- oder ACK-Fehler, wird ein Error Frame versendet. Dieses ist, ähnlich wie das EOF-Feld, durch seine sechs dominanten Bits am Anfang eindeutig identifizierbar, weil es die Bit-Stuffing-Regel verletzt. Durch die rein dominanten Bits ist auch sichergestellt, dass jegliche andere Kommunikation auf dem Bus unterbrochen bzw. überschrieben wird. Darauf folgen acht rezessive Bits als EOF. Nun können die Busteilnehmer die vorangegangene, möglicherweise fehlerhafte, Nachricht verwerfen und der ursprüngliche Sender kann sie neu übertragen.

2.2.2 CAN FD

CAN Flexible Data Rate (FD) ist eine abwärtskompatible Weiterentwicklung von CAN. Sie wurde mit der Absicht entwickelt höhere Datenraten zu erreichen. CAN FD ist auch in der ISO 11898 enthalten. Die zwei größten Unterschiede von CAN FD sind die erweiterte mögliche Datenmenge von 8 B auf 64 B pro Frame und die flexible Datenrate. Während der Arbitrierung nutzt CAN FD bis zu 1 Mbit/s und während der Datenphase bis zu 10 Mbit/s. Typisch sind, wie bei CAN, 500 kbit/s während der Arbitrierung und 2 Mbit/s in der Datenphase. [RS19]

CAN FD unterstützt genauso wie CAN, IDs mit 11 bit und 29 bit.

Start 1bit	ID 11bit	SRR 1bit	IDE 1bit	FDF 1bit	r0 1bit	BRS 1bit	ESI 1bit	DLC 4bit	Data 0..512bit	STC 4bit	CRC 16bit	ACK 2bit	EOF + IFS 10bit

Abbildung 2.3: Aufbau eines 11bit CAN FD-Datentelegramm

Abbildung 2.3 zeigt ein CAN FD-Datentelegramm mit 11 bit ID. Ein RTR gibt es bei CAN FD-Frames nicht, stattdessen kommt wie bei CAN 2.0b ein rezessives SRR vor. Das FDF-Bit (FD Format) zeigt an, dass es sich um eine CAN FD-Nachricht handelt, was aufgrund der Abwärtskompatibilität nötig ist. Durch das BRS-Bit (Bit Rate Switch) wird die flexible Datenrate von CAN FD angekündigt und die darauf folgende Übertragung der Nutzdaten inkl. des CRC wird mit einer höheren Datenrate ausgeführt. Zur besseren Fehlererkennung wurde das ESI-Bit (Error State Indicator) eingeführt. Mit diesem kann der Sender seinen aktuellen Fehlerzustand bekannt machen. Die letzte Neuerung im CAN FD-Frame ist das STC-Feld (Stuff Count). Drei Bit, mit einem vierten

Bit als Parity Check, geben die Anzahl der in diesem Frame vorkommenden Stuffing-Bits an. Aufgrund der zusätzlichen Länge von CAN FD-Frames ist dies nötig, um die Restfehlerwahrscheinlichkeit zu minimieren.

2.2.3 ISO TP

Die moderne Fahrzeugdiagnose entwickelt sich vor allem im Bezug auf Services zur Messwerterfassung stetig weiter. Immer mehr emissionsbezogene Messwerte und neue Messwerte für elektrifizierte Fahrzeuge kommen hinzu. Diverse Standards und Normen der ISO und Society of Automotive Engineers (SAE) haben sich etabliert. Abbildung 2.4 gibt einen Überblick über die, in dieser Arbeit relevanten, Normen. Gleichzeitig sind die Schichten des ISO/OSI-Schichtenmodell dargestellt, in die die jeweiligen Protokolle einzuordnen sind. Zu den meisten ISO-Standards gibt es SAE Äquivalente, die als technisch gleichwertig betrachtet werden können und deshalb hier nicht weiter von Relevanz sind.

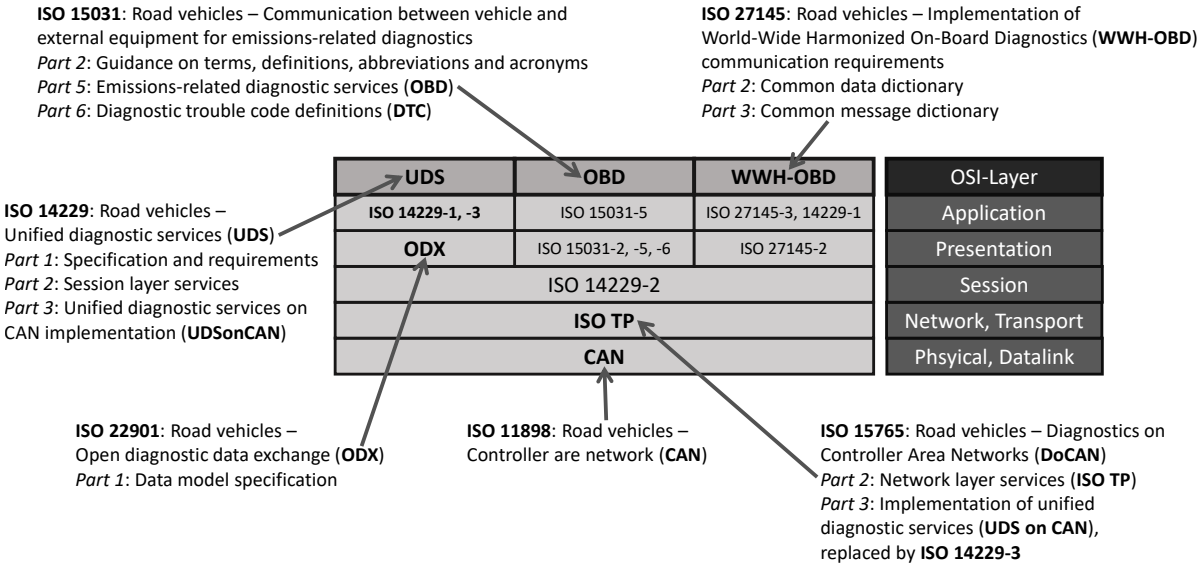


Abbildung 2.4: Fahrzeugdiagnose über CAN [LO11, ISO05, ISO17, SAE06, SZ11b]

Sowohl UDS als auch OBD und dessen weltweit homologierte Variante World-Wide Harmonized OBD (WWH-OBD), setzen im Falle von CAN auf ISO 15765-2, kurz ISO TP. ISO Transport Protocol (TP) ist in Verbindung mit CAN dafür verantwortlich, dass auch Nachrichten versendet werden können, die länger sind als die von CAN erlaubten 8 B. Um lange Nachrichten zu versenden, werden diese in mehrere CAN-Frames gesplittet. Diese Frames gibt es in den folgenden Varianten, vgl. auch Abbildung 2.5.

Single Frame (SF) Falls eine ISO TP-Nachricht versendet wird, die maximal 7 B groß ist, und somit zusammen mit der Protocol Control Information (PCI) in ein CAN-Frame passt. Die PCI beinhaltet hierbei die ersten vier Bits mit der Flag, die das Frame als SF auszeichnet, und das DL-Feld (Data Length).

First Frame (FF) Falls eine ISO TP-Nachricht versendet wird, die bis zu 4095 B groß sein kann. Diese Zahl ergibt sich entsprechend $2^{12} - 1 = 4095$, also der größten Zahl, die mit dem 12 bit DL-Feld abgebildet werden kann.

Consecutive Frame (CF) Nachrichten die über 7 B lang sind und mit einem FF eingeleitet wurden, werden in weitere CF geteilt. Entsprechend dem DL-Feld im FF kann berechnet werden, wie viele CF folgen. Jedes CF enthält eine Sequence Number (SN) um bei der Rekonstruktion der Nachricht zu helfen.

Frame Control (FC) Falls ein ISO TP FF empfangen wurde, wartet der Sender, bis ihm durch den Empfänger ein FC-Frame übermittelt wurde. Der Empfänger kann mit der FS-Flag (Flow Status) Bescheid geben, dass er bereit ist die ganze Nachricht zu empfangen. Ggf. kann er auch mitteilen, dass er (noch) nicht bereit ist oder ein Overflow vorliegt. Falls der Empfänger nur einen bestimmten Datendurchsatz verarbeiten kann, kann er dem Sender eine maximal zulässige Block Size (BS) sowie eine minimale Seperation Time (ST) mitteilen. Die BS beschreibt, wie viele CF gesendet werden dürfen, bevor der Sender wieder auf ein FC-Frame warten soll. Min. ST bezieht sich auf den minimalen Abstand zwischen zwei aufeinander folgenden CF in Millisekunden.

SF	**0x0** 4bit	**DL** 4bit	**Data** 56bit	
FF	**0x1** 4bit	**DL** 12bit	**Data** 48bit	
CF	**0x2** 4bit	**SN** 4bit	**Data** 56bit	
FC	**0x3** 4bit	**FS** 4bit	**BS** 8bit	**Min. ST** 8bit

Abbildung 2.5: ISO TP-Datagrams und -Frames

OBD-Testgeräte sind laut Norm dazu verpflichtet BS=0 und ST=0, also beliebig viele CF ohne FC-Frame (bis auf das Erste) und ohne minimalen Zeitabstand, verarbeiten zu können. Falls das Steuergerät also dem Tester eine ISO TP-Nachricht senden will, besteht der Kommunikationsablauf aus einem SF, einem FC-Frame und einer Anzahl CF entsprechend der Nachrichtenlänge. Das FC-Frame enthält hierbei lediglich 24 bit mit dem Inhalt `0x300000`. Nachrichten mit weniger als 8 B werden fast identisch mit reinen CAN-Nachrichten versendet. Lediglich das erste Byte in den CAN-Daten enthält ISO TP spezifische Informationen.

Manche Steuergeräte erwarten, dass auch bei kurzen ISO TP Übertragungen immer CAN-Frames mit maximalem DLC genutzt werden. Hierzu können die ungenutzten Bytes mit beliebigen Daten als Padding-Bytes gefüllt werden. Das DL-Feld lässt dann die Länge der relevanten Daten ermitteln.

2.2.4 OBD

On-Board-Diagnose (OBD) ist die wohl bekannteste Schnittstelle zur Erfassung von Diagnosedaten (Quelle: Google Trends 09.2020, On-Board-Diagnose vs. Unified Diagnostic Services vs. Controller Area Network). Die aktuelle Variante von OBD, wie sie in der ISO 15031 beschrieben wird, ist oft auch unter der Bezeichnung OBD2 zu finden. Im Nachhinein kann die erste Stufe von OBD, wie sie vor allem Anfang der 1990er Jahre in den USA aufkam, also als OBD1 bezeichnet werden. OBD war dabei lediglich die kontinuierliche Diagnose des Fahrzeugs von sich selbst, die über das kommuniziert wurde. Erst die spätere Version, OBD2, führte eine Schnittstelle ein, über die Fehlercodes (Diagnostic Trouble Codes (DTCs)) und Messwerte mit den entsprechenden Testgeräten abgefragt werden konnten. Einer der Haupteinsatzzwecke von OBD sind vor allem Abgas-/Emissionsrelevante Daten und Diagnose. [ZS14, LO11, SZ11b, ISO05, ISO17]

OBD als Diagnoseschnittstelle setzt auf ein einfaches *Request-Response Anfrage-Antwort*, System. In einem Netzwerk darf nur ein Tester anwesend sein, der *Requests* an ein oder mehrere (Broadcast) Electronic Control Units (ECUs) stellt. Die Adressierung wird über CAN von den CAN IDs übernommen. Hierbei wird zwischen funktionaler und physikalischer Adressierung unterschieden. Die funktionale Adressierung entspricht einem Broadcast des Testers an alle ECUs, die den angefragten Dienst unterstützen. Physikalische Adressierung ist direkt an einzelne ECUs gerichtet, bzw. wird genutzt, wenn eine ECU antwortet. Mit der funktionalen Adresse in Form der CAN ID `0x7DF` stellt der Tester eine Anfrage an alle Steuergeräte. Diese antworten dann mit den physikalischen Adressen `0x7E8` bis `0x7EF`. Entsprechend dieser IDs kann der Tester die Antworten den Steuergeräten zuordnen und nun direkt Anfragen an bestimmte Steuergeräte stellen. Hierzu dienen die physikalischen Adressen `0x7E0` bis `0x7E7`. Wie aus den Adressen ersichtlich ist, erlaubt OBD maximal acht ECUs, die auf *Requests* antworten.

Requests über OBD folgen einem Aufbau mit Service ID (SID) und Parameter ID (PID). Beide IDs nehmen ein Byte ein. Ein OBD-Request hat also meist drei Bytes Nutzdaten in einem CAN-Frame. `{0x02, 0x09, 0x0A}` ist dementsprechend der Inhalt eines CAN-Frames für den OBD-Request, um den ECU-Namen/ID zu erfragen. Diese Anfrage ist funktional adressiert sinnvoll, um alle teilnehmenden ECUs zu erfassen. Das erste Byte ist das ISO TP-PCI-Byte. Es folgt die SID für *Request vehicle information* und die PID für den ECU-Namen. Alle verfügbaren ECUs werden mit ihren jeweiligen physikalischen Adressen antworten. Da die ASCII-codierten ECU-Namen normalerweise länger als 7 B sind, gehören mehrere ISO TP CF zu den jeweiligen Antworten. OBD-Responses beinhaltet im ersten und zweiten Byte die SID und PID der Anfrage, um zugeordnet werden zu können. Die SID der Antwort wird um `0x40` zur Anfrage SID erhöht bzw. das 7. Bit gesetzt, um eine erfolgreiche Anfrage zu signalisieren. Fehlgeschlagene Anfragen werden mit einer SID mit dem Inhalt `0x7F` signalisiert.

Der Standard WWH-OBD in der ISO 27145 soll in Zukunft die ISO 15031 ablösen, ist zum Zeitpunkt dieser Arbeit (2021) allerdings nur für Nutzfahrzeuge vorgeschrieben. Viele PKWs unterstützen ihn aber schon heute.

WWH-OBD teilt die bereits aus OBD bekannten DTCs in Gruppen ein. Die Gruppen unterscheiden sich in ihrer Erheblichkeit für die Emissionswerte im Fehlerzustand. Weiter versucht WWH-OBD OBD in Einklang mit UDS zu bringen. Dazu setzt es auf die Services, wie sie von Unified Diagnostic Services (UDS) definiert werden. Auf lange Sicht kann so die OBD spezifische ISO 15031 ganz abgelöst werden. Die ISO 27145 ist zudem dahingehend in der Weiterentwicklung, dass sie WWH-OBD über z.B. Ethernet definiert. Abschließend sei auch die aktuelle ISO 15765-4 genannt, die OBD spezifisch auf dem CAN-Protokoll beschreibt.

2.2.5 UDS

UDS ist als Superset von OBD zu verstehen, da es OBD mit vielen Funktionalitäten erweitert und Herstellern mehr Freiheiten für eigene Anwendungen lässt. Es ist in der ISO 14229 beschrieben, während die ISO 15765-3, ähnlich wie bei OBD die ISO 15765-4, die CAN spezifische Implementierung von UDS beschreibt. Prinzipiell setzt UDS auf dasselbe Anfrage-Antwort-Prinzip wie OBD und setzt als Transportprotokoll auch ISO TP (ISO 15765-2) ein. Wo OBD von SID und PID spricht, nutzt UDS nur die Service ID und ein Subfunction Level Byte (LEV). Das LEV entspricht der PID, ist aber optional und muss nicht bei allen Services genutzt werden. Bit 7 des LEV kann zudem explizit gesetzt werden, um dem/den Steuergerät(en) mitzuteilen, dass keine positive Antwort gefordert ist. Dies kann die Buslast bei Anfragen, wie der „Tester Preset“-Anfrage, reduzieren. [ZS14, LO11, SZ11b, ISO17]

Die spezielle „Tester Preset“-Anfrage ist u.a. aufgrund der unterschiedlichen Sessions nötig, die UDS unterstützt. Ein Tester kann mit der entsprechenden Anfrage in Sessions wechseln, die umfangreichere Messwerte zur Abfrage zulassen oder sogar direkten Speicher- und Flashzugriff auf die ECU erlauben. Nach einer bestimmten Zeit ohne UDS-Kommunikation fällt die Session automatisch auf die Standardsession zurück. Um dies zu verhindern kann die „Tester Preset“-Anfrage zyklisch gesendet werden.

Das Wechseln zwischen Sessions ist teilweise durch Sicherheitsmechanismen geschützt. Bevor das Steuergerät das Wechseln der Session zulässt, muss der richtige Schlüssel (Key) übermittelt werden. Der Tester berechnet diesen Key über einen geheimen Algorithmus mit einem Seed, den er in der Standardsession vom Steuergerät abrufen kann.

Während die Services, die per UDS erreichbar sein müssen, genormt sind, sind es deren spezifische Implementierungen nicht. Hersteller können frei wählen, wie sie diese umsetzen. Das heißt Services zur Messwertabfrage können z.B. die verschiedensten Messwerte anbieten. Lediglich allgemeine Werte zur Fahrzeugidentifikation wie die Vehicle Identification Number (VIN) oder der ODX-Identifier, um die richtigen ODX-Beschreibungsdateien für eine bestimmte ECU zu bestimmen, sind laut Norm gefordert. Mit der

richtigen ODX-Datei kann dann genau ermittelt werden, welche Messwerte das jeweilig angefragte Steuergerät anbietet, mehr hierzu in Kapitel 2.4. Meist ist aber davon auszugehen, dass die Messwerte mindestens denselben Umfang haben, wie die, die über OBD abfragbar sind.

Gegenüber OBD bietet UDS neben den Sessions weitere nützliche komplexere Möglichkeiten, wie den Service `ReadDataByPeriodicIdentifier`. Damit ist kein Polling für Messwerte mehr nötig, sondern das Steuergerät schickt von sich aus Messwerte an den Tester in gegebenem Takt. Der Service `ResponseOnEventService` bietet eine ähnliche Entlastung für den Tester, indem das Steuergerät bei bestimmten Ereignissen automatisch Meldungen über die Ereignisse und Ergebnisse übermittelt.

2.3 TCP/IP

Das Transmission Control Protocol (TCP) setzt auf das Internet Protokoll (IP) auf. Wie in Kapitel 2.1 angesprochen, ist TCP in Schicht 4, IP in Schicht 3 und das meist auf den untereren Ebenen verwendete Ethernet in den Schichten 1 und 2 anzuordnen. Wie ebenso bereits angesprochen, wurde aufgrund der Verbreitung dieser Protokolle ein eigenes Schichtenmodell, das TCP/IP-Referenzmodell (siehe Abbildung 2.6), erdacht. IP ist hier in dem *Network Layer* einzuordnen während TCP oder das auch oft verwendete User Datagram Protocol (UDP) im *Transport Layer* arbeiten.

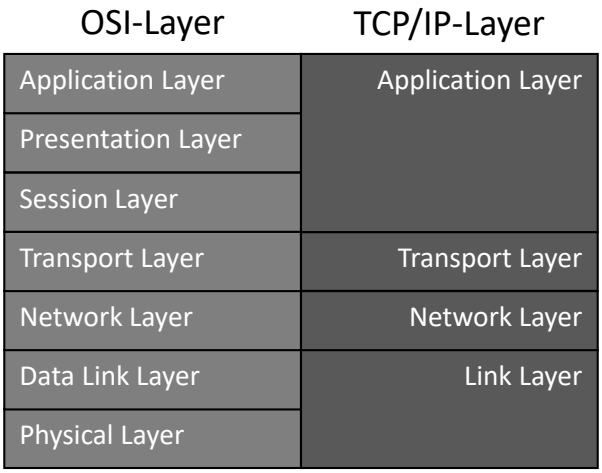


Abbildung 2.6: TCP/IP-Referenzmodell

TCP/IP wird oft als eine Bezeichnung genutzt, da es ursprünglich ein einziges Protokoll war. Der Begriff wurde erst später in IP und TCP geteilt, um dem Hinzukommen von UDP gerecht zu werden. Das oft als Basis genutzte Ethernet ist ähnlich wie CAN ein Protokoll mit CSMA-Medienzugriffsverfahren, also nicht deterministisch. Im Gegensatz zu CAN werden bei Ethernet Kollisionen nicht vermieden, sondern erkannt und

dann weiter behandelt, CSMA/Collision Detection (CD). Dementsprechend kümmert sich TCP/IP um Aufgaben der höheren Schichten, wie der Vermittlung zwischen Sender und Empfänger (Routing), Datensegmentierung und Übertragungssicherung. Die letzten beiden Punkte fallen in die Zuständigkeit von TCP. Wie diese Mechaniken umgesetzt sind, ist der größte Unterschied zwischen TCP und UDP. Während TCP Datenströme segmentieren kann und die Übertragung durch sogenannte *Acknowledgements* absichert, fallen diese beiden Funktionalitäten bei UDP weg.

Falls Nachrichten anfallen, die zu lang für ein Frame sind, werden diese durch TCP in mehrere einzelne Frames aufgeteilt. TCP stellt durch Sequenznummern sicher, dass mehrere Frames auf der Empfängerseite wieder in der korrekten Reihenfolge interpretiert werden, auch wenn sie in einer anderen Abfolge empfangen wurden. Dementsprechend ist auch ein Zusammensetzen geteilter Nachrichten problemlos möglich. Der Acknowledge-Mechanismus von TCP erwartet eine Bestätigung des Empfängers auf jedes empfangene Frame. Falls diese ausbleibt, werden Frames automatisch erneut übermittelt.

Um einen höheren Durchsatz zu erreichen, setzt TCP diverse Mechaniken um, die beispielsweise mehrere Nachrichten oder Acknowledges sammeln und dann in einem großen zusammengefassten Frame versenden (Nagle's-Algorithmus oder TCP-Delay). So wird vermieden, dass viele kleine Pakete versendet werden, die alle den Protokoll-Overhead haben. Stattdessen wird nur ein Paket mit dementsprechend nur einmal dem Overhead versendet. Je nach Anwendung kann die damit zustande kommende Verzögerung aber auch ein Nachteil sein.

UDP verzichtet auf einen Kontrollfluss und Pakete werden versendet, ohne ein Acknowledge zu erwarten. Es fällt das „Anmelden“ beim Server, wie es bei TCP nötig ist, weg und Pakete werden blindlings einfach an den Empfänger gesendet. Es liegt also in der Verantwortung der Applikation, Paketverluste sowie verletzte Reihenfolgen zu behandeln. UDP hat dementsprechend in manchen Anwendungen einen Vorteil gegenüber TCP, da weniger Overhead auf dem Bus nötig ist. Der offensichtliche Nachteil ist, dass es an den Mechanismen für den abgesicherten Datenversand mangelt.

Beide Protokolle haben eine erweiterte Adressierung in Form von Ports gemeinsam. Neben den IP-Adressen, die normalerweise einzelne Rechner/Geräte identifizieren, können so einzelnen Anwendungen über ihre Port-Nummer identifiziert und direkt angesprochen werden.

2.4 ODX

Bereits in Kapitel 2.2.5 wurde von ODX-Dateien (Open Diagnostic Data Exchange) gesprochen. Vor allem in Kombination mit dem offen definierten UDS-Protokoll sind diese Dateien nötig, um je nach Fahrzeug ermitteln zu können, welche Services erreichbar sind und wie diese anzusprechen sind. Mehrere ODX-Dateien kommen meist in einem komprimierten Archiv namens PDX. PDX-Dateien sind klassische Zip-Archive mit an-

gepasster Dateiendung (`*.pdx`). ODX und PDX sind im Standard ASAM MCD-2 D beschrieben [ASA08, SZ11a].

Jedes PDX-Archiv enthält eine `index.xml`, die alle enthaltenen Dateien mit Beschreibung listet. Enthalten sind neben ODX-Dateien teilweise auch Java Bibliotheken (in `*.jar` gepackt), die fertige Java-Methoden zur Interaktion mit bestimmten Steuergeräteservices zur Verfügung stellen. Solche Bibliotheken können beispielsweise den Algorithmus für das Seed-Key-Verfahren beim geschützten Sessionwechsel beinhalten.

Besonders interessant für diese Arbeit sind aber die ODX-Dateien. Aus ihnen kann genau ermittelt werden, welches Steuergerät des vorliegenden Fahrzeugs welche Messwerte zur Verfügung stellt, wie diese abzufragen und zu interpretieren sind. Damit Fahrzeughersteller ODX-Dateien wiederverwenden können, sind diese hierarchisch aufgebaut und erlauben ein Konzept von Bibliotheken und Vererbung. Oft gibt es z.B. eine ODX-Datei, die als Bibliothek für verbreitete Datentypen und deren Interpretation dient. Dort können Ganz- und Gleitkommazahlentypen mit ihrer Größe, Bytereihenfolge und Limits definiert werden. Eine solche separate Bibliothek ist offensichtlich sinnvoll von fahrzeugspezifischeren Definitionen zu trennen, da sie ein Hersteller kaum von Fahrzeug zu Fahrzeug ändern wird.

In ODX-Dateien/-Dokumenten können andere Dokumente eingebunden werden, womit alle Objekte, die im eingebundenen Dokument definiert sind, geerbt werden. Spezielle ODX-Dokumente, die die bereits angesprochenen expliziten Bibliotheken (`ECU-SHARED-DATA`) beinhalten, können über denselben Weg eingebunden werden. Diese zwei Varianten unterscheiden sich. Sie sind als Vererbung aus Elterndokumenten und Import von Bibliotheken betitelt. Die Vererbung erlaubt das Überschreiben oder gar gänzliche Ignorieren bestimmter Objekte aus dem Elterndokument. Vererbung kann über mehrere Dokumente hinweg geschehen, wobei ein Dokument (D1) von einem Elterndokument (E1) erbt, das selbst von einem weiteren Elterndokument (E2) erbt. E2 ist dann in D1 sichtbar. Falls E1 Objekte aus E2 ignoriert, sind diese auch für D1 unsichtbar. Vererbte Elemente von E2, die E1 überschreibt, sind in der E1-Variante in D1 sichtbar. Importierte Bibliotheken unterscheiden sich dahingehend, dass sie nicht mit vererbt werden. Außerdem können Definitionen/Objekte aus Bibliotheken nicht überschrieben, aber durchaus beim Import ignoriert werden.

Damit sich ein Objekt in einem Dokument auf ein anderes, möglicherweise importiertes oder vererbtes, Objekt beziehen kann, bietet der ODX-Standard zwei Referenztypen:

ODX-Link/ID-REF Dieser Verweis erfolgt auf eine Objekt `ID`, die im gesamten Projekt einmalig sein sollte. Die `ID` wird sowohl im lokalen Dokument/der aktuellen Datei als auch in allen eingebundenen/geerbten Dokumenten gesucht. Wenn das gesuchte Objekt in einem Elterndokument oder in einer Bibliothek anhand seiner `ID` gefunden wird, aber im lokalen Dokument entsprechend seines Namens (`SHORT-NAME`) überschrieben wird, ist die per `ID` eindeutige Referenz auf das Objekt im Elterndokument/in der Bibliothek zu verwenden.

SHORT-NAME-REF/SN-REF Im Gegensatz zur `ID-REF` ist hier immer das Objekt im aktuellen Namensraum gültig. Das heißt überschreibt das untersuchte Dokument das Objekt, das in einem Elterndokument schon definiert war, so ist die Definition aus dem lokalen/aktuell untersuchten gültig. Falls es sich um eine Referenz auf ein Objekt aus einer Bibliothek handelt, ist es natürlich weiterhin ungültig dieses lokal zu überschreiben, weshalb die Unterscheidung der Referenztypen nicht getroffen werden muss.

Neben der komplexen Struktur der Vererbung in ODX-Dateien, die stark an die Vererbung in objektorientierten Programmiersprachen erinnert, arbeitet ODX auch mit unterschiedlichen Containern. Solche Container können als typisierte Namensräume betrachtet werden. Es gibt verschiedene Typen von Containern, von denen es mehrere Instanzen geben kann, die sich entsprechend ihres Namens über mehrere Dateien hinweg erstrecken können. Diese Container unterscheiden sich nicht nur durch ihren Inhalt, sondern auch in der Art wie sie die besprochenen Vererbungen und Importe zulassen und inwiefern es mehrere Instanzen geben darf oder nicht. Der volle Umfang, der sich so ergebenden Möglichkeiten, ist für diese Arbeit voraussichtlich nicht erforderlich. Deshalb werden alle weiteren spezifischen Eigenheiten von ODX nach Bedarf im Abschnitt der Umsetzung 5.2.3 erläutert.

2.5 MPAS

Multi Process Automation System (MPAS) ist eine Softwaresuite der PA-Systems GmbH & Co. KG. Die Hauptanwendung der Software ist die Prüfstandautomatisierung in der Automobilbranche. Ein modularer Aufbau erlaubt es eine Vielzahl von Geräten anzusprechen; klassisch sind dies vor allem die Prüfstandsrolle(n), das Fahrerleitgerät und diverse Messeinrichtungen für u.a. Abgasemissionen. MPAS ist hauptsächlich in Visual Basic auf Basis des .Net-Framework 4.6 (zum Zeitpunkt dieser Arbeit) realisiert.

Entsprechend des .Net-Frameworks von Microsoft ist es möglich, dynamisch linkbare Bibliotheken, DLLs, in die Applikation zu laden. Die in MPAS als Add-in bezeichneten Erweiterungen machen davon umfangreich Gebrauch, um je nach Bedarf Gerätetreiber in die Hauptsoftware zu laden. Ein Add-in wird als eine DLL kompiliert und kann einen oder mehrere Gerätetreiber beinhalten. Ein Gerätetreiber ist die Schnittstelle von MPAS zu einem Gerät, beispielsweise besagten Rollen, Abgasmesseinrichtungen oder dem Fahrerleitgerät. Die Gerätetreiber kommunizieren über die verschiedensten Protokollen mit Geräten, die mit dem Leitrechner verbunden sind. Der Leitrechner ist der Rechner, auf dem MPAS ausgeführt wird. Beispiele für solche Protokolle sind Ethernet basierte Protokolle wie TCP und UDP oder auch RS-232.

Im Grundaufbau bietet MPAS ein Hauptfenster, das den aktuellen Test und dessen Fortschritt sowie Meldungen anzeigt. Die Meldungen sind, ähnlich wie bei PLC basierten Automatisierungen weit verbreitet, in Fehler, Warnungen und Infos aufgeteilt, die protokolliert und teilweise vom Bediener explizit quittiert werden müssen. Ein extra Fenster,

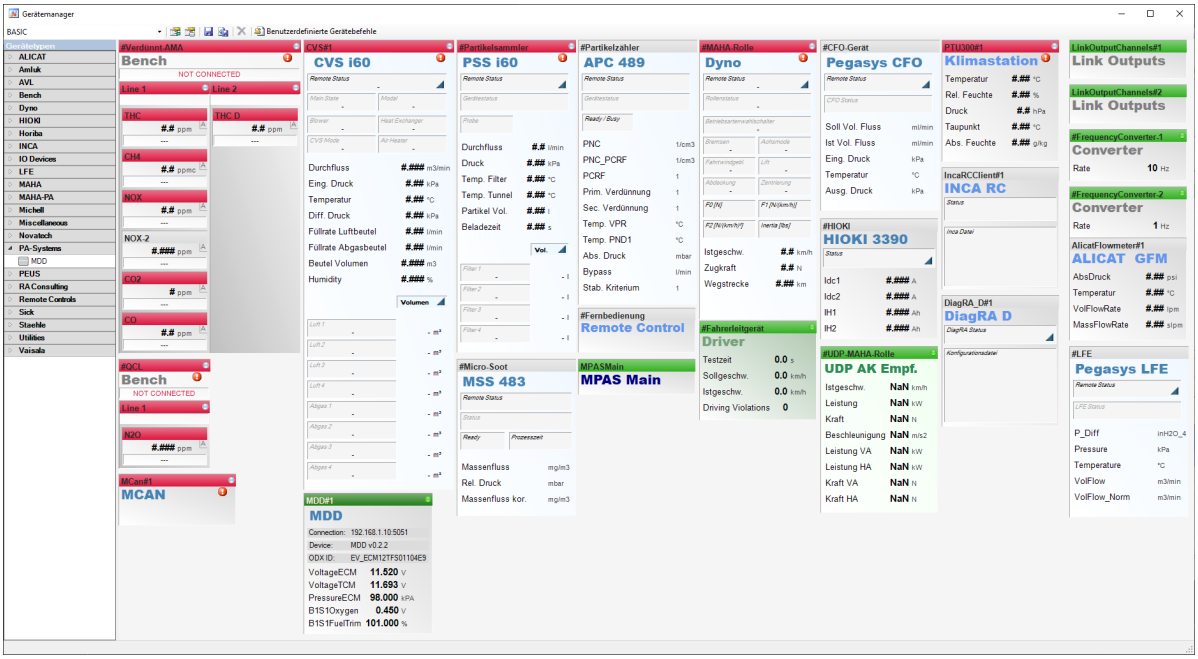


Abbildung 2.7: MPAS Gerätemanager

der Gerätemanager (siehe Abbildung 2.7), zeigt die aktuelle Gerätekonfiguration. Hier kann per Drag&Drop aus den Gerätetreibern, die über DLLs geladen wurden, eine Auswahl getroffen werden. Kleine Kontrollfenster/Applets zeigen den Zustand der aktivierten Geräte, beispielsweise ob ihre Verbindung steht und die Konfiguration gültig ist. Im Gerätemanager werden die meisten Konfigurationen für Geräte hinterlegt, wie TCP/IP-Adressen, Polling-Raten und aufzuzeichnende Messkanäle. Neben dem Gerätemanager ist der Testprozess Designer (siehe Abbildung 2.8) ein Grundbaustein von MPAS. Dort kann visuell ein Ablauf programmiert werden, der eine Testdurchführung beschreibt. Die Bausteine aus denen sich ein solcher Ablauf/Prozess zusammensetzt, werden als Nodes bezeichnet. Nodes gibt es in unterschiedlichster Komplexität. Es gibt Nodes die ein Neuladen der Konfiguration anstoßen, Timer starten oder stoppen, Messungen starten oder stoppen, Fahrkurven auslösen oder dem Nutzer Meldungs- und Dialogfenster öffnen. Indem bestimmte Nodes Subnodes haben können, ergibt sich die Möglichkeit mit gewissen Nodes sogar nebenläufige Abläufe zu realisieren. Viele einsetzbare Nodes kommen mit dem Standardumfang von MPAS selbst. Gerätetreiber können für sie spezifische Nodes zusätzlich definieren.

Die größte Einheit/Komponente in MPAS ist der Test. Hier finden die verschiedenen Komponenten, wie die Gerätekonfiguration aus dem Gerätemanager, Fahrkurven für den Test und ein Testprozess zusammen. Falls ein Testprozess Messwerte aufzeichnet, werden diese im Test hinterlegt. Eine Datenbankverbindung archiviert ausgeführte Tests.

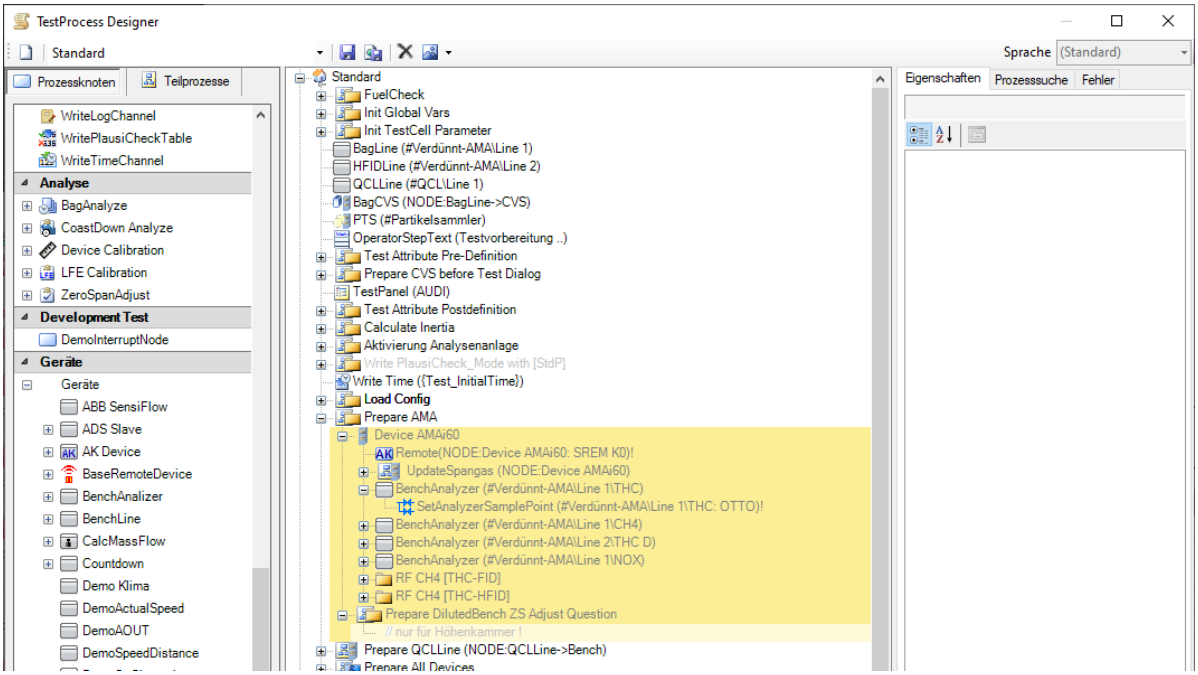


Abbildung 2.8: MPAS Testprozess

2.6 MCAN

MCAN ist eine Hardware- und Softwarelösung, die bei PA-Systems im Zuge einer früheren Masterthesis entstanden ist [Kra15]. Das Peak System PCAN-Ethernet Gateway DR wurde dabei mit einer eigenen Anwendung geflasht, die über eine TCP/IP-Verbindung Daten mit MPAS austauschen kann. Ein dazugehöriges MPAS Add-in bezieht aus einer sogenannten DBC-Datei Informationen über den CAN-Bus. Das Format der DBC-Datei wurde von der Vector Informatik GmbH etabliert und beschreibt die Kommunikation über CAN in Messwerten und Gerätestatus. Mit dieser Information kann das MPAS Add-in entscheiden, welche CAN-IDs das Gateway filtern und durchreichen soll und wie die empfangenen Daten zu interpretieren sind. Die in dieser Arbeit behandelte Lösung stellt im weitesten Sinne eine Neuentwicklung von MCAN mit neuem Fokus dar. Der Fokus liegt auf OBD und UDS, anstatt auf dem „rohen“ CAN-Bus. Je nach Bedarf kann MCAN parallel zur neuen Lösung eingesetzt werden, ggf. auf derselben Hardware.

2.7 Buildroot

Buildroot [Bui20] ist ein Open-Source-Projekt, eine Art Framework, um eigenständig lauffähige Linux-Systeme zu entwickeln. Es unterstützt verschiedenste Plattformen und Architekturen, ist aber vor allem für eingebettete Systeme gedacht. Die Buildroot-Umgebung stellt alle nötigen Tools zur Verfügung, um eine Cross-Compiler-Toolchain aufzuset-

zen, den Linux-Kernel zu konfigurieren und zu kompilieren, das Linux-Root-Filesystem anzulegen und einen Bootloader zu kompilieren.

Die umfangreichen Repositorien/Archive von Buildroot stellen bereits viele Tools und Programme zur Verfügung, die als Pakete direkt in den Build-Prozess integriert werden können. Buildroot setzt auf das *make* Build-Managementtool. Dementsprechend kann Buildroot die, für den Linux-Kernel entwickelte, *Kconfig* Konfigurationssprache [ker20] nutzen, die *make* Projekte konfigurieren kann. Die damit einhergehende Möglichkeit verbreitete Tools wie *menuconfig* zur Konfiguration zu nutzen, garantiert ein einheitliches Erlebnis bei der Konfiguration von Buildroot, Linux, Bootloader usw. Um eigene Pakete/Programme einbinden zu können, unterstützt Buildroot neben *make* aber auch komplexere/erweiternde Build-Tools wie *CMake* und *autotools*. Die angelegte Cross-Compiler-Toolchain kann zudem direkt genutzt werden, um einzelne Programme zu übersetzen ohne das Programm direkt als Paket in Buildroot integrieren zu müssen.

Buildroot etabliert eine Projektstruktur, die es ermöglicht diverse Konfigurationen, zusätzliche Pakete, Patches usw. unabhängig von der eigentlichen Buildroot-Struktur, einzubinden. Anbieter von Mikrocontrollern und Evaluationsboards können dank dieser Funktionalität sogenannte Board Support Packagess (BSPs) anbieten. Anwendungsentwickler können diese BSPs nutzen, um schnell ein minimales Setup in Buildroot aufzusetzen, das es ermöglicht den jeweiligen Mikrocontroller zu programmieren.

Ein explizit auf das jeweilige Projekt angepasste Betriebssystem hat den Vorteil, dass weitaus kleinere und effizientere Systeme gestaltet werden können, als es mit klassischen generischen Linux-Distributionen möglich wäre. Buildroot vereinfacht eben dieses Vorgehen.

3 Stand der Technik

Dieses Kapitel beschreibt den Stand der Technik auf dem diese Arbeit aufbaut. Es handelt sich um einen Überblick zu diversen relevanten Themen. Ähnlich wie bezüglich der Grundlagen wird in späteren Kapiteln ggf. noch einmal genauer auf Abhängigkeiten und Entwicklungen zum Stand der Technik eingegangen.

3.1 CAN unter Linux

Linux ist ein monolithischer Kernel. Dennoch wird viel mit Kernelmodulen gearbeitet, die nach Bedarf hinzugeladen werden können. Solche Module beinhalten z.B. Gerätetreiber. Im offiziellen Mainline-Kernel 5.11-rc7 (Stand Februar 2021), finden sich CAN-Gerätetreiber für diverse CAN-Controller von elf verschiedenen Herstellern[1], inklusive der virtuellen CAN-Treiber. Diese Vielzahl von Treibern benötigt ein einheitliches Interface, um von Userspace-Anwendungen genutzt werden zu können. Hierfür wurde durch Volkswagen Research in 2008 das *SocketCAN*-Interface eingeführt [Har12, HTK+20], vergleiche auch Kapitel 4.3.1.1. SocketCAN vereinheitlicht die Nutzung der CAN-Controller über deren Treiber, indem es eine Schnittstelle bietet, die Berkeley-Sockets realisiert. Programmierer, die bereits Erfahrung mit Berkley-Sockets, z.B. durch die Programmierung von TCP/IP-Kommunikation, haben, können diese dank SocketCAN auch für CAN anwenden.

SocketCAN kam bereits als Grundbaustein für die bestehende MCAN-Lösung von PA-Systems zum Einsatz. Während MCAN [Kra15] noch selbst eine Lösung realisierte, um CAN-Frames zu filtern und per TCP/IP weiterzuleiten (vergleiche Kapitel 2.6), gibt es seitdem Lösungen für derartige Probleme, die direkt Teil von SocketCAN sind. Einerseits ist es möglich SocketCAN Sockets mit Filtern zu parametrisieren, andererseits gibt es CAN-Gerätetreiber, die TCP/IP-CAN-Tunnel erlauben.

Es gibt alternative Interfaces zu SocketCAN mit anderen Zielen. *LinCAN* und *RTCAN* sind Beispiele für alternative CAN-Interfaces. Beide zeichnen sich durch andere Designphilosophie als SocketCAN aus. SocketCAN ist im Netzwerkstack mit anderen Protokollen wie TCP/IP angeordnet. Das damit einheitliche Interface erleichtert die Programmierung, bringt aber Overhead mit, u.a. weil sich Ressourcen geteilt werden müssen. Wenn CAN in besonders zeitkritischen Anwendungen zum Einsatz kommen soll, können deshalb die alternativen Interfaces LinCAN und RTCAN wünschenswert

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/can?h=v5.11-rc7

sein. Beide konzentrieren sich darauf zeitlich strengere Vorgaben erfüllen zu können, auf Kosten der Bedienbarkeit. Da SocketCAN als einziges der Interfaces Teil des Mainline-Kernels ist, ist es dennoch die verbreitetste Alternative. [SNV15, SPP+10]

Weder Linux noch CAN sind für harte Echtzeitfähigkeit gedacht. SocketCAN ist, wie gezeigt wurde, nicht die beste Wahl für zeitkritische Anwendungen. Es stellt sich die Frage, ob die Kombination dieser Elemente überhaupt für Anwendungen über nicht zeitkritisches Messen hinaus sinnvoll ist. Beispiele wie ein Regelungssystem für Elektromotoren und Superkondensatoren zeigt, dass die Kombination dieser Komponenten durchaus auch für komplexere Anwendungen genutzt werden kann [AMMC17]. Das Fazit betont dennoch, dass das Ergebnis noch nicht fehlerfrei ist. Vor allem Jitter und Delay durch den Linux-Kernel stellen noch Probleme dar.

Für Linux gibt es deshalb diverse Patches und Erweiterungen, um den Kernel für Echtzeitanwendungen zu optimieren [DKS11]. Das vorab erwähnte Regelungssystem zeigt jedoch, dass je nach Anwendung auch schon ein angepasster minimaler Kernel mit optimiertem Scheduler ausreichend sein kann. Kapitel 4.4.1.2 beschäftigt sich damit weitergehend.

Abschließend sei noch auf das ISO TP-Modul hingewiesen, dass für SocketCAN zur Verfügung steht. Hierbei handelt es sich um eine Erweiterung für SocketCAN, die das ISO TP-Protokoll auf Basis von CAN umsetzt. Das Modul wurde während des Zeitraums dieser Arbeit in den Mainline-Kernel aufgenommen [Har20a]. Während der Konzeption wird in Kapitel 4.3.1.1 genauer auf dessen Möglichkeiten und Vorteile eingegangen.

3.2 OBD und UDS zur Messwerterfassung

Die im Grundlagenkapitel 2.2 beschriebenen Protokolle OBD und UDS setzen auf CAN auf. Im Zuge dieser Arbeit sollen über sie kontinuierlich Messwerte durch Polling erfasst werden. OBD, und damit auch das als Weiterentwicklung zu betrachtende UDS, sind vor allem für die Diagnose entwickelt worden. Damit wird sich vor allem auf die Diagnose von Problemen mit Serienfahrzeugen in Werkstätten bezogen. Hierzu kommen meist proprietäre Prüfgeräte[2] in Markenwerkstätten zum Einsatz, die auf dem offenen Markt nur beschränkt verfügbar sind. Alternativ gibt es diverse OBD-USB- und OBD-Bluetooth-Adapter[3], teilweise mit Software, von Drittanbietern. Hierbei handelt es sich meist um Geräte für den Privatgebrauch mit weniger umfangreichen Funktionalitäten und ohne Garantie, dass sie prinzipiell mit allen OBD-fähigen Fahrzeugen kompatibel sind. Weiter beschränken sich diese Geräte oft auf das Auslesen von Fehlercodes, nicht auf das kontinuierliche, fehlerfreie Messen von Messwerten mit Frequenzen von über 1 Hz.

[2] https://www.auto-diagnostik.de/Profi-Diagnosegeraete/

[3] https://www.conrad.de/de/p/adapter-universe-obd-ii-interface-7170-1-st-1889389.html

Es gibt diverse Softwarelösungen, Open- und Close-Source, die mit verschiedenen OBD-Interfaces arbeiten können. Auch hier geht es meist um die Diagnose mit DTCs. Allerdings bieten diese Tools teilweise auch APIs, um sie in anderen Anwendungen zu nutzen. Sowohl Open-Source-Lösungen, bspw. OBD2-Scantool[4], als auch Close-Source-Lösungen, z.B. Peak Systems PCAN-View[5] und OBD/UDS-APIs, haben gemeinsam, dass sie zum Experimentieren und Testen in der Entwicklung gedacht sind, nicht aber für den Einsatz rund um die Uhr auf Prüfständen. Während einiger Tests im Kapitel 4 wird darauf weiter eingegangen. Lösungen über Umwege, bei denen z.B. durch das PCAN-View-Tool Werte aufgezeichnet und die Aufzeichnungen danach durch andere Tools ausgewertet werden, wirken für den automatisieren Dauerbetrieb schlichtweg nicht flexibel und zuverlässig genug [KRV16]. Dabei ist zudem der nötige Polling-Vorgang für OBD/UDS noch gar nicht beachtet.

Abschließend lässt sich sagen, dass SocketCAN und das Linux ISO TP-Modul vielversprechende Grundlagen für diese Arbeit sind. Entsprechend der Anforderungen zur Messwerterfassung über OBD und UDS gilt es allerdings eigene Grundlagen zu schaffen.

[4] https://github.com/AustinMurphy/OBD2-Scantool
[5] https://www.peak-system.com/PCAN-View.242.0.html

4 Konzeption

Dieses Kapitel behandelt die Konzeption zur angestrebten Lösung. Es wird auf genutzte Hardware, Software und verschiedene Architekturentscheidungen eingegangen. Diverse Evaluationen und Tests werden für Entscheidungen herangezogen.

4.1 Geräte und Software

Während der Konzeption wurden diverse Hardware- und Software-Komponenten genutzt. Die folgende Auflistung soll einen Überblick hierzu verschaffen:

Testlaptop, Windows-Host, Leitrechner für MPAS Ein Dell Latitude 5510 mit einer Windows 10 Pro Installation.

Test-VM, Ubuntu/Linux-Host Eine Virtuelle Maschine (VM) auf Basis von Ubuntu 20.04, die als Gastsystem auf dem Testlaptop läuft.

Gateway, Linux-Target Das Peak System PCAN-Ethernet Gateway DR[1], auf Basis eines ARM9 Freescale iMX257 Prozessors. Es stellt zwei CAN-Kanälen, ein Ethernet Interface (RJ45), eine RS232 Schnittstelle für ein Remote-Terminal und einen microSD-Karten-Slot zur Programmierung zur Verfügung. Letzterer ist allerdings nur in einer speziellen Developer-Version verfügbar. Der Slot ist hilfreich, um bei der Entwicklung von eigener Firmware nicht jedes Mal den Flash des Gateway-Boards überschreiben zu müssen.

USB-Adapter PCAN-USB[2] ist ein CAN-Interface für USB. Es kann mit Tools wie PCAN-View[3] oder der kostenfreien PCAN-Basic-Application Programming Interface (API)[4] genutzt werden.

Peak Firmware Die Firmware von Peak Systems für das PCAN-Ethernet Gateway. Ursprünglich diente diese nur dazu, CAN-Nachrichten über TCP oder UDP bidirektional zwischen zwei Gateways auszutauschen. Das heißt es kann mit Hilfe von zwei Gateways eine physikalische Verlängerung von CAN-Netzen über Ethernet erreicht werden. Hierbei können Filter eingerichtet werden, die nur bestimmte

[1] https://www.peak-system.com/PCAN-Ethernet-Gateway-DR.330.0.html
[2] https://www.peak-system.com/PCAN-USB.199.0.html
[3] https://www.peak-system.com/PCAN-View.242.0.html
[4] https://www.peak-system.com/PCAN-Basic.239.0.html

CAN-IDs weiterleiten. Um CAN-Frames auch auf Windows-Systemen in eigenen Applikationen zu verwenden, bietet Peak Systems das Virtuelle Gateway[5] an. Hierbei handelt es sich um eine Windows-Applikation, die als Service im Hintergrund läuft. Ein virtuelles Gateway, das mit einem Physikalischen verbunden wurde, kann dann über dieselben Tools, PCAN-View und PCAN-Basic, wie der USB-Adapter genutzt werden. In neuen Firmwareversionen ist es zudem möglich, Peak's proprietäre Handshake-Mechaniken zu deaktivieren. Zusammen mit der veröffentlichten Protokolldefinition ist damit die Kommunikation nicht mehr auf Gateway zu Gateway (egal ob virtuell oder physikalisch) beschränkt. Nutzer können in eigenen Applikationen Sockets für UDP oder TCP und die Protokolldefinition nutzen, um direkt mit dem Gateway zu kommunizieren, ohne dass ein virtuelles Gateway nötig ist. Für die bidirektionale Kommunikation sind stets zwei Sockets nötig. Falls das Gateway CAN-Nachrichten sendet, muss der Empfänger ein Server-Socket öffnen, dass auf einem bestimmten Port lauscht. Wenn CAN-Nachrichten an das Gateway gesendet werden sollen, muss die Anwendersoftware sich durch einen Client-Socket mit einem bestimmten Port auf dem Gateway verbinden.

Custom/PA-System Firmware Jegliche genutzte Firmware, die anstatt der originalen Peak Firmware auf dem Gateway genutzt wurde. Diese kann entsprechend Abschnitt 4.3 erstellt werden.

SocketCAN Die native Schnittstelle für die Nutzung von CAN-Hardware in Linux-Systemen. Applikationen auf dem Gateway können die zwei CAN-Interfaces über solche Sockets direkt ansprechen. Wird der USB-Adapter mit einem Linux-Host verbunden, kann auch dieser so eingerichtet werden, dass er direkt via SocketCAN nutzbar ist (vgl. Kapitel 4.3.1.1).

ECUs, Steuergeräte Für einige Testanwendungen stand eine Kombination aus Motor- (ECM) und Getriebesteuergerät (TCM) zur Verfügung. Hierbei war der CAN-Bus, über den beide kommunizieren, offengelegt. Aufgrund eines fehlenden Fahrzeuges oder Fahrzeugsimulators sind genaue Messwerte auf dem Bus allerdings nicht zu erwarten. Ausnahmen sind Messwerte für z.B. die aktuelle Versorgungsspannung oder den Umgebungsluftdruck. Ersterer hat logischer Weise immer einen sinnvollen Wert, wenn das Steuergerät mit Strom versorgt wird. Letzterer ist dank internem Barometer im Steuergerätegehäuse verfügbar.

4.2 Performance-Bewertung CAN over Ethernet

Wie im Kapitel 1.3 angesprochen, besteht die grundlegende Aufgabenstellung darin, OBD und UDS über CAN in MPAS zu unterstützen. Hierzu soll externe Hardware zum

[5] https://www.peak-system.com/Virtual-PCAN-Gateway.390.0.html

Einsatz kommen, die CAN- und Ethernet-Interfaces besitzt. Um zeitnah und genau Messwerte über diesen Weg aufzunehmen, bestand die erste Evaluation dieser Arbeit darin, zu erwartende Busauslastung, Delays, Packageloss, Jitter und Zeitstempel zu testen.

4.2.1 Busauslastung

Die ersten Messungen wurden getätigt, um ein Gefühl für die zu erwartende Busauslastung auf Ethernet zu entwickeln. Hierzu wurden die CAN-Frames, die von den Steuergeräten generiert wurden, entweder an das Virtual Gateway (VG), das als Service läuft, oder an einen einfachen Socket-Tester[6] gesendet. Letzterer verbindet sich lediglich, bzw. stellt einen Server-Socket bereit und gibt alle empfangenen Daten nach ASCII decodiert aus. Die Ausgabe der Nachrichten im Peak Protokoll als ASCII-Zeichenketten ergibt erwartungsgemäß keine sinnvollen Informationen. Für diesen Test ist aber lediglich ein empfangendes Socket nötig, womit die Interpretation irrelevant ist. Alle Messungen wurden über den Netzwerkprotokollanalyzer Wireshark[7] getätigt. Zuerst war lediglich die Peak Firmware im Einsatz. Es wurde TCP und UDP gegenübergestellt, wobei als empfangende Stelle einerseits ein virtuelles Gateway und andererseits Sockets genutzt wurden. Die Peak Firmware ermöglicht es Nagle's-Algorithmus für TCP zu aktivieren oder zu deaktivieren. Daher wurde auch das getestet. Im Falle von UDP gibt es die Option, dass pro UDP-Paket 1 bis 15 CAN-Frames zusammengefasst werden. Hier wurde 1fpp und 15fpp (frames per package) gegenübergestellt.

Tabelle 4.1: Durchsatz- und Buslastmessungen in Wireshark: Messung über 180 s mit Buslast, die von den beiden Steuergeräten erzeugt wird (etwa 710,75 CAN-Frames pro Sekunde), oder künstlicher Buslast über USB-Adapter und PCAN-View (1000 CAN-Frames pro Sekunde)

Protocol	**Delay/fpp**	**Rate** (Hz)	**Reciever**	**IO** (pkgs/s)		**T** (kB/s)	**RTT** (ms)	**Len** (B)	**Data** (B)	**fpw**
				total	**incoming**					
TCP	w/	710,75	VG	54,81	26,43	29,97	41	1134	1080	30
TCP	w/	710,75	Socket	48,12	24,06	27,29	41	1134	1080	30
TCP	w/o	710,75	VG	986,28	704,64	63,42	2	90	36	1
TCP	w/o	710,75	Socket	n/a	n/a	n/a	n/a	n/a	n/a	n/a
UDP	15	710,75	VG	n/a	n/a	n/a	n/a	n/a	n/a	n/a
UDP	15	710,75	Socket		710,65	55,43	-	78	36	1
UDP	1	710,75	VG		710,81	55,44	-	78	36	1
UDP	1	710,75	Socket		710,81	55,44	-	78	36	1
TCP	w/o	1000	VG	1501,88	997,28	89,76	1	90	36	1

Tabelle 4.1 zeigt die Ergebnisse dieser Messungen. Bereits in den ersten beiden Messungen für TCP mit (*w/*) TCP-Delay zeigt sich, dass sich zwischen VG und reinem

[6] https://github.com/akshath/SocketTest
[7] https://www.wireshark.org/

Socket kaum ein Unterschied ergibt. Deshalb wurde in einigen weiteren Messungen dieser Vergleich ausgelassen. Der Unterschied zwischen aktivierten und deaktiviertem (*w/o*) TCP-Delay/Nagle's-Algorithmus zeigt sich jedoch deutlich. Die Anzahl der insgesamt benötigten Pakete verringert sich bei aktiviertem Algorithmus deutlich und dank des gesparten Overhead sinkt zudem der Durchsatz/Troughput (T) merklich. Mit aktivierten TCP-Delay werden Pakete mit einer durchschnittlichen Länge von 1134 B versendet. Abzüglich des Protokoll-Overhead sind das 1080 B Nutzdaten. Ein Frame, entsprechend des Protokolls der Peak Firmware, hat 36 B. Es ergeben sich 1080 B/36 B = 30 frames per window (fpw). Zuletzt fällt der Unterschied in der Round Trip Time (RTT) auf. Diese wird als Verzögerung zwischen Paketversand und empfangenem Acknowledge gemessen. Aufgrund der Verzögerung durch die Sammlung von Frames durch Nagle's-Algorithmus ist die höhere RTT nachvollziehbar.

Die folgenden Messungen behandeln UDP. Auch hier unterscheiden sich die VG vs. Socket-Varianten kaum. Weiter zeigt sich, dass die Einstellung in der Firmware zum sammeln von mehreren Frames in ein UDP-Paket nicht zu greifen scheint, vgl. eingestellte frames per package (fpp) zu wirklich empfangenen fpw. Dementsprechend wurden hierzu weitere Messungen ausgelassen. Der Overhead von UDP ist erwartungsgemäß kleiner als der von TCP. Während bei TCP bei einzelnen Frames 90 B − 36 B = 54 B Overhead entsteht, sind es bei UDP nur 78 B − 36 B = 42 B. Zudem fallen die Acknowledges weg. Der Troughput auf der Leitung ist entsprechend kleiner.

Abschließend wurde eine Messung mit künstlicher Buslast, generiert durch den USB-Adapter und PCAN-View, durchgeführt. Ein Test-Frame wird mit einer Rate von 1000 Hz verschickt, was der großzügig aufgerundeten Rate an Frames entspricht, die durch die zwei Steuergeräte generiert wird. Für die meisten folgenden Messungen, die künstlich generierte Frames benötigen, wird deshalb diese Rate gewählt.

4.2.2 Delay, Packageloss und Jitter

Mit der Hilfe einiger Python-Skripte wurde nun Delay, Packageloss und Jitter gemessen. Zuerst wurden CAN-Frames durch das PCAN-View-Tool generiert, während ein Python-Skript die PCAN-Basic-API nutzt, um die CAN-Frames im Peak Protokoll zu empfangen. Das ermöglicht es einerseits den USB-Adapter als Sender und das Gateway+VG als Empfänger und vice versa zu nutzen. Für jede empfangene Nachricht wird ein Zeitstempel auf dem Windows-Host hinterlegt. Bei einer Rate von 100 Hz wurden 1000 Frames, bzw. die Zeitstempel ihrer Ankunft, aufgezeichnet. Im Falle von Tests mit 1000 Hz wurden 10000 Zeitstempel aufgezeichnet.

In einem anderen Skript konnten diese Zeitstempel dann in Deltas zwischen je zwei aufeinander folgenden Zeitstempeln ausgewertet werden. Diese Deltas sollten unabhängig von der Latenz/dem Delay auf dem CAN-Bus sowie der TCP- bzw. UDP-Verbindung sein, da sie relativ dazu sind. Zu erwarten sind Deltas, die der Rate der generierten Frames, entspricht. Falls es allerdings zu Jitter, vermutlich vor allem auf der Ethernet-Verbindung, kommt, sollten sporadisch größere Deltas auftreten. Tabelle 4.2 zeigt diese

Tabelle 4.2: Delay- und RTT-Messungen mit Python-Skripten: Messungen über 1000 Frames bei 100Hz und 10000 Frames bei 1000Hz

Protocol	**Delay/fpp**	**Rate** (Hz)	**Gen.**	**Delta** (µs)		**RTT** (µs)	
				avg.	**3σ**	**avg.**	**3σ**
TCP	w/	100	USB	9997,27	43164,01	23861,71	44853,13
TCP	w/	100	Eth	9880,66	12928,26	3846,76	5192,01
TCP	w/	1000	USB	997,73	41419,08	n/a	n/a
TCP	w/	1000	Eth	n/a	n/a	n/a	n/a
TCP	w/o	100	USB	10000,50	17945,34	2708,36	4020,42
TCP	w/o	100	Eth	9879,13	13951,01	3616,93	5819,49
TCP	w/o	1000	USB	1000,03	2018,01	2268,34	3990,02
TCP	w/o	1000	Eth	997,58	3797,01	4170,29	5985,01
UDP	1	100	USB	n/a	n/a	2644,98	4056,38
UDP	1	100	Eth	n/a	n/a	3766,09	5797,62
UDP	1	1000	USB	999,94	2035,10	2174,19	4543,59
UDP	1	1000	Eth	941,59	3798,00	4391,07	5993,00

Werte. Wie vermutet entspricht der Durchschnitt der Deltas immer der Rate der gesendeten Frames. Bei 100 Hz sind Frames alle 10 000 µs zu erwarten. Die Perzentile über 99,7 % aller Deltas, beschreibt die maximalen Deltas, die in 99,7 % der Fälle auftreten. Falls eine Normalverteilung zu Grunde liegt, entspricht dies etwa drei Standardabweichungen (3σ). Hier zeigen sich bei der Übertragung mit TCP-Delay deutlich höhere Ausreißer als bei der Übertragung ohne. Das entspricht der Logik, dass Nagle's-Algorithmus sporadisch Verzögerungen für manche Frames einführt. UDP im Vergleich zu TCP zeigt kaum Unterschiede. Im besten Fall kann davon ausgegangen werden, dass selbst bei einer Busauslastung mit 10000 CAN-Frames pro Sekunde, die Messeinrichtung spätestens nach etwa 4 ms neue Werte liefert. Laut der Gesetzgebungen müssen die meisten Messwerte mit maximal 5 Hz gesampelt werden [Cou18, p. 687]. 4 ms sind also mehr als ausreichend.

Interessant ist, dass der USB-Adapter scheinbar genauer als die Gateway-Kombination ist, wenn es darum geht CAN-Frames in einer bestimmten Rate zu generieren und zu versenden. Der Grund hierfür könnte in der Art und Weise liegen, wie das PCAN-View--Tool mit den zwei Geräten kommuniziert. Es ist wahrscheinlich, dass der USB-Adapter durch das Tool so parametrisiert wird, dass er intern mit eigenem Oszillator Frames generiert, während die Gateway-Kombination auf Frames setzt, die das Tool generiert. Da das Tool als Userspace-Anwendung unter Windows läuft, ist es vom Windows-Scheduling beeinflusst. Für weitere Messungen bedeutet das, dass stets der USB-Adapter dazu genutzt werden sollte, Frames zu generieren. Damit ist der Fehler in den Messungen durch die Toolkette vermindert.

Der zweite Teil dieser Messungen war die RTT. Hiermit ist nicht die RTT von TCP-Paketen und den jeweiligen Acknowledgements gemeint, sondern die Zeit die vergeht, bis

ein versendetes CAN-Frame wieder empfangen wird. Dementsprechend muss die Toolkette so konfiguriert werden, dass Frames an derselben Stelle empfangen werden können, an der sie versendet werden. Die Hardware bleibt dieselbe. Gateway und VG auf einer Seite, USB-Adapter auf der anderen. Anstatt das PCAN-View-Tool einzusetzen, werden nun aber beide Seiten über Python und die PCAN-Basic-API angesprochen. Das entsprechende Python-Skript nutzt mehrere Threads um Daten zu generieren, zu versenden und zu empfangen. Bei jedem Versenden und Empfangen wird ein Zeitstempel auf dem Windows-Host hinterlegt. Indem der Inhalt der CAN-Nachricht als eine fortlaufende Nummer gewählt wird, kann dann der Zeitstempel des Sendens, dem des Empfangens zugeordnet werden. Das Delta zwischen beiden ergibt die dargestellte RTT. Zudem kann überprüft werden, ob Sequenznummern ausgeblieben sind, was auf Packageloss hindeutet. Da dieses Phänomen in keiner der Messungen aufgetreten ist, kann es vernachlässigt werden.

Das dargestellte Ergebnis deckt sich mit den meisten bereits etablierten Vermutungen. Der USB-Adapter schafft es Frames zeitnäher und in genauerem Takt zu versenden. UDP erreicht keine besseren Werte als TCP. Die maximalen Verzögerungen in der optimalen Variante, TCP, ohne Nagle's Algorithmus, bei 1000 Hz, von USB nach Ethernet, erreicht nach 3σ etwa 4 ms, was mehr als ausreichend ist.

Tabelle 4.3: RTT-Messungen mit Python-Skripten: Je 5 gemittelte Messungen über 180s bei 1000Hz

Protocol	**Delay/fpp**	**Rate** (Hz)	**Firmware**	**Recv.**	**RTT avg.** (µs)	**RTT 3σ** (µs)
TCP	w/o	1000	Peak	Socket	1123,59	2079,21
TCP	w/o	1000	Peak	VG	2289,89	4196,42
UDP	1	1000	Peak	Socket	1127,86	2103,43
TCP	w/o	1000	Custom	Socket	1087,62	2209,01

In Tabelle 4.3 sind noch einmal die RTTs bei unterschiedlichen Setups dargestellt. Da sich 1000 Hz als die sinnvollste Datenrate für Tests herausgestellt hat, und RTT als Messwert die höchsten Aussagekraft hat, wurde sich hier auf diese Werte konzentriert. Erneut wurden Python-Skripte genutzt, um die Messungen aufzunehmen und auszuwerten. Das Protokoll wurde zwischen UDP und TCP variiert. CAN-Frames werden über das Python-Skript, die PCAN-Basic-API und den USB-Adapter generiert. Empfangen werden die CAN-Frames in den jeweiligen TCP- oder UDP-Paketen über das Virtual Gateway oder direkt an einem Socket.

Es zeigt sich erneut, dass UDP nicht schneller als TCP ist. Weiter ist auffällig, dass der Windows-Service, der das VG bereitstellt, scheinbar eine zusätzliche Latenz von etwa 2 ms einführt. Die letzte Messung in der Tabelle zeigt die Ergebnisse mit einer alternativen Firmware. Diese wurde entsprechend der in Kapitel 4.3 beschriebenen Methodik entwickelt. Sie imitiert die Funktionalität der Peak Firmware, bezüglich des Weiterleitens von CAN-Frames als TCP/IP-Pakete, in der einfachst möglichen Art. Die erreichte

RTT, die praktisch identisch mit der der Peak Firmware ist, stellt den Proof of Concept für die Implementierung einer eigenen Firmware dar, die spezifischeren Anforderungen entspricht, siehe Kapitel 4.4.

4.2.3 Ergebnis

Zusammenfassend lässt sich sagen, dass UDP unerwartet kaum schneller als TCP ist. Obwohl die fehlenden Absicherungsmechanismen von UDP eigentlich die Geschwindigkeit als Vorteil haben, kann dieser Vorteil in dieser Anwendung nicht ausgespielt werden. Dies liegt zum einen daran, dass der nötige Handshake beim Verbindungsaufbau über TCP sowieso nur einmal, zum Beginn der Messung, vorkommt. Zum anderen werden in den meisten Tests CAN-Frames nur in eine Richtung über TCP versandt. Damit ergibt sich, dass die zusätzlichen Pakete für TCP-Acknowledges nur auf der Verbindungsrichtung anfallen, auf der sonst kein Verkehr ist. Die weitere Arbeit wird sich deshalb auf TCP als Transportprotokoll festlegen.

TCP erlaubt es zwischen aktiviertem und deaktiviertem Nagle's Algorithmus zu wechseln. Während dieser deutlich schlechtere Werte für die RTT erzeugt, sorgt er doch für signifikant weniger Buslast und somit geringere benötigte Bandbreite. Welchen dieser Vorteile diese Arbeit bevorzugt, wird in Kapitel 4.4 erläutert.

Das Virtual Gateway (VG) sorgt für zusätzliche, unerwünschte Latenz. Ob die Vorteile des VG, wie bereits bestehende APIs für Protokolle wie UDS und OBD diesen Nachteil aufwiegen, bespricht ebenfalls Kapitel 4.4.

4.3 Custom Firmware

Für weitere Tests wurde es nötig, eigene Firmware und Programme für das Peak System PCAN-Ethernet Gateway DR entwickeln zu können. Peak stellt hierfür ein Buildroot-BSP zur Verfügung. Das BSP ermöglicht es eine Cross-Compilation-Toolchain, ein Linux-Kernel-Image, einen Bootloader und ein Root-Filesystem für den ARM9 Freescale iMX257 Prozessor zu erstellen. Zudem bietet es diverse Konfigurationen, um die zusätzlichen Funktionen auf dem Board und Chip des Gateways zu parametrisieren. Dazu gehören zwei PCA82C251 CAN-Transceiver von NXP, eine RS232-Schnittstelle für z.B. eine serielle Konsole, Ethernet und ein WLAN Modul bei der wireless Variante des Gateways. Das BSP konfiguriert Buildroot so, dass BusyBox[8], uClibc[9] und U-Boot[10] genutzt werden. BusyBox ist ein Programm, das viele klassische Linux/UNIX-Tools in eine Programmdatei packt. Dazu gehören u.a. Tools zur Nutzer-, Rechte- und Schnittstellenverwaltung. uClibc ist eine C-Standardbibliothek, die im Vergleich zur verbreiteten GNU-C-Standardbibliothek (glibc) weitaus kleiner ist. U-Boot ist ein Bootloader.

[8] https://busybox.net/
[9] https://www.uclibc.org/
[10] http://www.denx.de/wiki/U-Boot

4.3.1 CAN und Linux

Auch wenn Linux eigentlich ein monolithischer Kernel ist, so wird doch viel mit Kernelmodulen gearbeitet. Die meisten Geräte- und Protokolltreiber sind z.B. Module, die zur Laufzeit nach Bedarf geladen oder entladen werden können. Je nach Kernelkonfiguration können Module wahlweise auch statisch integriert werden. [SBP07]

Ursprünglich haben Hersteller, die ihre CAN-Hardware unter Linux unterstützen wollten, ihre eigenen Schnittstellen für Userspace-Anwendungen definiert. Das heißt es standen Kernelmodule zur Verfügung, die entweder Teil von Mainline-Linux waren oder quergeladen werden konnten. Die Entwickler dieser Module mussten selbst das Interface definieren, über das die Anwendungen im Userspace auf die Module im Kernelspace zugreifen. Die Funktionalität war oft eingeschränkt und es konnten meist nur rohe CAN-Frames versendet und empfangen werden. Die Einführung von *SocketCAN* änderte das.

4.3.1.1 SocketCAN

SocketCAN ist eine Schnittstelle zwischen Gerätetreibern für CAN und dem Userspace. Die Schnittstelle nutzt viele bereits vorhandene Funktionalitäten für Kommunikationsprotokolle im Linux-Kernel. Für den Anwender lässt sie sich fast identisch wie die Schnittstelle für z.B. TCP/IP nutzen, da sie auch auf Berkeley Sockets setzt. [HTK+20, Ste90, Har12]

Entsprechend der klassischen Nutzung von Sockets stehen bekannte Funktionen wie `socket`, `bind`, `connect`, `recv`, `send` usw. zur Verfügung. Zwei Besonderheiten der SocketCAN-Implementierung sind der Broadcast Manager (BCM) und das ISO TP-Modul. Statt einem „RAW“ CAN-Socket kann „BCM“ als Socket-Typ gewählt werden. Der BCM kann zyklisch ein oder mehrere sequentielle CAN-Frames in einem definierten Zeitraster versenden. Das hat den Vorteil, dass die nötigen Timer für eine zyklische Übertragung im Kernelspace verwaltet werden. Eine Anwendung, die im Userspace Timer verwaltet, hat mit mehr/häufigeren Verzögerungen im Scheduler zu rechnen. Der dritte Socket-Typ ist ISO TP. Dieser Typ kann ISO TP-Nachrichten empfangen, wobei er automatisch FC-Frames versendet/empfängt und Nachrichten aufteilt bzw. wieder zusammensetzt. Auch hier ist der Vorteil, dass die protokollspezifischen Funktionalitäten direkt im Kernel abgebildet sind, anstatt den Protokollstack in einer Userspace-Anwendung umzusetzen. Zum Zeitpunkt dieser Tests, Ende 2020, ist das ISO TP-Modul [Har20b] noch nicht Mainline und muss deshalb quergeladen werden. Die Bemühungen das Modul in die Mainline-Repositorien aufzunehmen laufen allerdings schon [Har20a].

Da das BSP auf dem Linux-Kernel mit der Version 2.6.31 von 2009 [Tor09] basiert, gab es beim Querladen Kompatibilitätsprobleme mit dem deutlich neueren ISO TP-Modul. Das größte Problem entstand durch den Support von CAN FD in neueren Linux-Versionen. Das Modul setzt diesen Support voraus. Indem der entsprechende `can.h`-Header in den alten Linux-Quellen für Buildroot gepatcht wurde, entsteht zwar kein CAN FD-Support, allerdings sind die Abhängigkeiten für das ISO TP-Modul damit gegeben.

So konnte das Modul für Linux 2.6.31 kompiliert und dann geladen werden. Das Modul wird mit Header-Patch direkt im Buildroot-Projekt integriert.

4.3.2 Projektstruktur

Wie vorab schon angedeutet, waren einige Anpassungen am Buildroot-Projekt, das auf dem BSP basiert, nötig. Diese gehen über die eigentlichen Anwendungsprogramme hinaus. Einzelne Programme können manuell, mit der von Buildroot angelegten Toolchain, kompiliert und dann manuell auf das Zielsystem kopiert werden. Die übersichtlichere und nachvollziehbarere Methode ist jedoch, sie direkt in das angelegte Buildroot-Projekt zu integrieren. Das erstellte Linux-Image und das Root-Filesystem beinhalten diese dann direkt. Je nach Änderung ist es allerdings ratsam, diese erst manuell zu kompilieren und zu testen. Einzelne Komponenten zu kompilieren ist deutlich schneller als die komplette Buildroot-Umgebung inkl. Compiler, Kernel etc. zu übersetzen. Aus Gründen der Übersichtlichkeit und Pflegbarkeit wurde deswegen eine Projektstruktur erarbeitet, um diese Anpassungen möglichst getrennt vom Buildroot-Hauptverzeichnis und dem Peak BSP abzulegen. Die folgende Liste demonstriert die Ordnerstruktur:

- `buildroot-mdd-env`: Root-Verzeichnis der Buildroot-Umgebung (*git repository*)
  - `buildroot-2018.02.6`: Buildroot-Release (externe *git repository*)
  - `peak-mx25-bsp-2.8.1`: Teil des BSP
  - `pasystems-external`: Projektspezifische Anpassungen (*git repository*)
    - `board`: Patches, Programmquellcode (*git repositories*), Root-Filesystem-Overlay, Kernelkonfiguration und User-Table
    - `configs`: Angepasste Standardkonfigurationen für Buildroot
    - `package`: Kconfig-Dateien für u.a. die Anwendungsprogramme

Die komplette Projektstruktur wird mit der Versionsverwaltung *git* versioniert. Das Root-Verzeichnis ist hierbei die Basis des Projektes. Es ist Teil des Peak BSP. Im Root-Verzeichnis findet sich ein Shell-Skript, das Buildroot initialisiert. Es klont das Buildroot-Release mit der gegebenen Version in den `buildroot-2018.02.6` Ordner. Damit muss die eigentliche Buildroot-Installation nicht Teil der Versionsverwaltung für das Projekt sein. Der `peak-mx25-bsp-2.8.1` Ordner ist ebenso Teil des BSP. Er enthält von Peak bereitgestellte Standardkonfigurationen für Buildroot, uClibc, U-Boot und Linux. Weiter enthält er Patches, zusätzliche Tools, die Systeminitialisierungsdateien u.v.m. Da der Ordner Teil des BSP ist, teilt er sich die Repository mit dem Root-Verzeichnis. Diese Repository sollte sich in Zukunft höchstens minimal ändern und kann ggf. gegen BSPs für andere Boards ausgetauscht werden.

Ein Ordner für den Großteil dieser Arbeit wurde mit `pasystems-external` angelegt. Er wird als *git*-Submodul von `buildroot-mdd-env` als eigenständige Repository

integriert. Der Ordner teilt sich eine ähnliche Struktur mit dem Ordner `peak-mx25-bsp-2.8.1`. Das Shell-Skript des BSP im Root-Verzeichnis konnte deshalb so angepasst werden, dass einige Einstellungen/Konfigurationen, Patches etc. überschrieben werden können, ohne das eigentliche BSP zu editieren. Einzig das Skript musste angepasst werden. Der `pasystems-external` Ordner beinhaltet auch jegliche C-Programme, die im Zuge dieser Arbeit als Pakete in die Buildroot-Umgebung integriert werden müssen. Während der `package`-Ordner die entsprechenden Konfigurationen beinhaltet, liegt der Quellcode der Programme in `board`. Einzelne Programme sind in extra Unterordnern und über eigene Repositories versioniert. Diese Repositories können als *git*-Submodul von `pasystems-external` integriert werden.

Da die einzelnen Programme separate Repositories sind, können sie ohne die gesamte Buildroot-Umgebung getrennt geklont und bearbeitet werden. Buildroot erlaubt es, die generierte Toolchain, die es ermöglicht auf der Test-VM (Ubuntu-Host) mit x86-Architektur Binärdateien für das ARM Target zu kompilieren, als Software Development Kit (SDK) zu exportieren. Mit einer solchen SDK ist es dann möglich, die Repositories der Programme völlig separat von einer ganzen Buildroot-Umgebung zu klonen und zu kompilieren. Die erstellten Binärdateien können auf das bereits lauffähige Target, über beispielsweise die SD-Karte oder `scp` (ein Tool um über Secure Shell (SSH) Dateien zu kopieren), kopiert werden. Dieses Vorgehen ist weitaus schneller als mit der vollständigen Buildroot-Umgebung zu arbeiten, vor allem während der Entwicklung.

Zusätzliche Kernelmodule, wie das für ISO TP, können ähnlich wie zusätzliche Programme eingebunden werden. Lediglich die Struktur der entsprechenden Kconfig-Dateien in `package` müssen angepasst werden.

4.4 Architekturentwurf

Während der Architekturentwicklung wurde ein Namen für das Gateway bestimmt. Von hier ab wird sich auf die Gateway, vor allem in Kombination mit der eigenen Firmware, als Measurement and Diagnostic Device (MDD) bezogen.

Der Kern des MDD ist die Firmware für das Gateway. Die meisten Steuergeräteanfragen via OBD und UDS folgen einem Anfrage-Antwort-Schema. UDS bietet zwar einen Service, der automatisch in festlegbaren Intervallen Messwerte sendet, dieser kann aber als Ausnahme betrachtet werden. Daraus ergibt sich, dass das MDD alle benötigten Messwerte in einem definierten Zyklus durch Polling beziehen muss. Die Peak Firmware erlaubt dies nicht auf dem Gateway. Das heißt das MPAS Add-in müsste die entsprechenden Anfragen/Requests generieren und über das Gateway an das Steuergerät übermitteln. Der Polling-Zyklus wäre damit von TCP/IP, CAN, Peak Firmware und MPAS Add-in beeinflusst. Im Gegensatz dazu wäre es mit einer eigenen Firmware möglich, das Gateway das Polling verwalten zu lassen. Dieses Vorgehen kürzt den TCP/IP-Jitter sowie den Windows-Kernellag der MPAS Add-in-Implementierung aus der Gleichung. Vor diesem Hintergrund wirkt die Umsetzung durch eine eigene Firmware vorteilhaft.

Um diesen Vorteil weiter zu untermauern, wurden einige Tests zum OBD Polling durch Custom Firmware durchgeführt.

4.4.1 POSIX-Timer und Realtime-Scheduling

Präzises Polling durch Software setzt einen präzisen Timing-Mechanismus voraus. Unter Linux bietet sich hierzu die Implementierung der POSIX-Timer an. Portable Operating System Interface (POSIX) beschreibt eine standardisierte Schnittstelle für Betriebssysteme. Die aktuellste Version des Standards ist die IEEE Std 1003.1-2017 [Jos20]. Viele UNIX-Derivate halten sich zum Großteil an POSIX, wie Linux seit Version 2.6, sind aber nicht offiziell als 100% kompatibel getestet. Das Standardinterface, das Linux garantiert, ist die Linux Standard Base (LSB). Bezüglich der hier vor allem relevanten C-Systemaufrufe, kann LSB als Superset von POSIX betrachtet werden [Lin15]. Die sogenannten Man-Pages (Manual), wie sie in vielen UNIX-Derivaten, auch Linux, als Kommandozeilentool vorzufinden sind, beschreiben diese C-Systemaufrufe ausführlich. Weiter gibt es viele Webseiten, die die unterschiedlichen Versionen dieser Man-Pages im HTML-Format hosten (Beispiel für Linux[11]).

4.4.1.1 POSIX-Timer

Aus diesen Dokumentationen ergeben sich neue Entscheidungen, die bezüglich der Timer getroffen werden müssen. Einerseits ist das die Art der dem Timer zugrundeliegenden Uhr. Hier bieten sich drei Varianten an:

`CLOCK_REALTIME` Diese Uhr entspricht der systemweiten Uhr und ist von Sprüngen und Ratenanpassungen durch z.B. Network Time Protocol (NTP) betroffen.

`CLOCK_MONOTONIC` Diese Uhr kann nicht gesetzt werden und bezieht sich auf einen unbestimmten aber fixen Zeitpunkt in der Vergangenheit. Sie ist nicht von Sprüngen, wie `CLOCK_REALTIME`, betroffen, aber von Ratenanpassungen.

`CLOCK_MONOTONIC_RAW` Diese Uhr verhält sich wie `CLOCK_MONOTONIC`, ignoriert allerdings Ratenanpassungen. Das heißt `CLOCK_MONOTONIC_RAW` kann wie der Wert des unkorrigierten Oszillator der CPU betrachtet werden.

Zuerst schien `CLOCK_MONOTONIC_RAW` die beste Option, da sie verhindert, dass Korrekturen während einer Messung das Zeitraster verfälschen. Allerdings hat sich beim Testen der entsprechenden Systemaufrufe gezeigt, dass die Kombination aus Linux-Kernelversion und uClibc für Timer, die auf dieser Uhr basieren, fehlerhaft ist. Recherche in einschlägigen IRC-Chats ergibt, dass `CLOCK_MONOTONIC_RAW`-basierte Timer in vielen Linux-Versionen vorhanden sind, aber verschwiegen werden, da sie kaum genutzt/empfohlen werden. Deshalb wurde auf `CLOCK_MONOTONIC` zurückgegriffen. Da entsprechende

[11] https://man7.org/linux/man-pages/

Timer keine Sprünge machen, sind sie in dieser Anwendung genauso unproblematisch. Eine Ratenanpassung könnte in Kombination mit anderen Geräten zur Synchronisation sogar hilfreich sein, insofern überhaupt ein NTP-Client auf dem Gateway aktiviert wird.

Die zweite Entscheidung liegt in der Art und Weise wie der Timer die Applikation benachrichtigen soll. Hier gibt es die folgenden relevanten Möglichkeiten:

- Keine asynchrone Benachrichtigung: Diese Option erfordert es, den Timer manuell synchron zu Pollen und so herauszufinden, ob er abgelaufen ist.
- Benachrichtigung via POSIX-Signal: Wenn der Timer abläuft wird ein definierbares POSIX-Signal generiert und an die Applikation gesendet. Diese kann dann entscheiden, wie sie darauf reagiert.
- Starten eines definierbaren Threads: Wenn der Timer abläuft, wird ein POSIX-Thread (pthread) gestartet.

Asynchrones Polling des Timers erfordert eine regelmäßige Abfrage. Da der Timer dazu benutzt werden soll, einen Zyklus für einen Polling-Vorgang vorzugeben, würde das bedeuten, dass der Timer per Polling in bestimmtem Zeitraster geprüft werden muss, um selber ein Polling zu generieren. Damit wäre also nichts gewonnen. Wenn der Timer bei Ablauf einen Thread startet, ähnelt das der Möglichkeit asynchron auf ein Signal mit einem sogenannten Signalhandler zu reagieren. Signalhandler sind allerdings kein eigener Thread sondern unterbrechen den empfangenden Thread. Dies kann problematisch werden, je nachdem was im Signalhandler getan wird (ähnlich wie Interrupts). Das trifft vor allem dann zu, wenn sich mehrfach anfallende Signale und Signalhandler gegenseitig unterbrechen. Je nachdem was bei Ablauf des Timers getan werden soll, sind separat gestartete Threads mit ihrem eigenen Kontext deswegen sinnvoller, obwohl sie einen deutlich größeren Overhead haben. Wenn allerdings beispielsweise alle 10 ms ein Timer abläuft und deshalb alle 10 ms ein neuer Thread gestartet wird, kostet das Performance. Eine weitere Möglichkeit ist es, das ausgelöste Signal vom Empfang zu blockieren und dann durch einen blockenden Systemaufruf an einer bestimmten Stelle im Code auf das Signal zu warten. Dieser Ansatz hat weder das Problem, dass regelmäßig der Status des Timers gepollt werden muss, noch ein extra Thread oder ein Signalhandler gestartet werden muss. Indem der entsprechende Systemaufruf blockiert, ist sichergestellt, dass es zu keiner, dauerhaft CPU-Zeit benötigenden, ununterbrochenen Dauerschleife kommt.

Der folgende Test (Abbildung 4.1 und Tabelle 4.4) vergleicht die zwei Möglichkeiten eines Signalhandlers und des blockenden Wartens auf das Signal.

Zu sehen ist ein Test, bei dem zwei Timer alle 10 ms ausgelöst wurden. Ein dritter Timer verwaltet die Benchmarkdauer. Es wurde dann ein extra Thread gestartet, der in einer Schleife bis Benchmarkende immer wieder durch einen `sigwait`-Systemaufruf blockiert wird. Dieser wird jedes Mal, wenn das entsprechende Signal anfällt, aufgeweckt. Im Gegensatz dazu löst der zweite Timer ein Signal aus, auf welches ein Signalhandler registriert ist. Beide Auslösemechanismen sorgen dafür, dass ein atomarer Zähler inkrementiert wird sowie ein Protokolleintrag mit dem aktuellen Systemzeitzeitstempel und

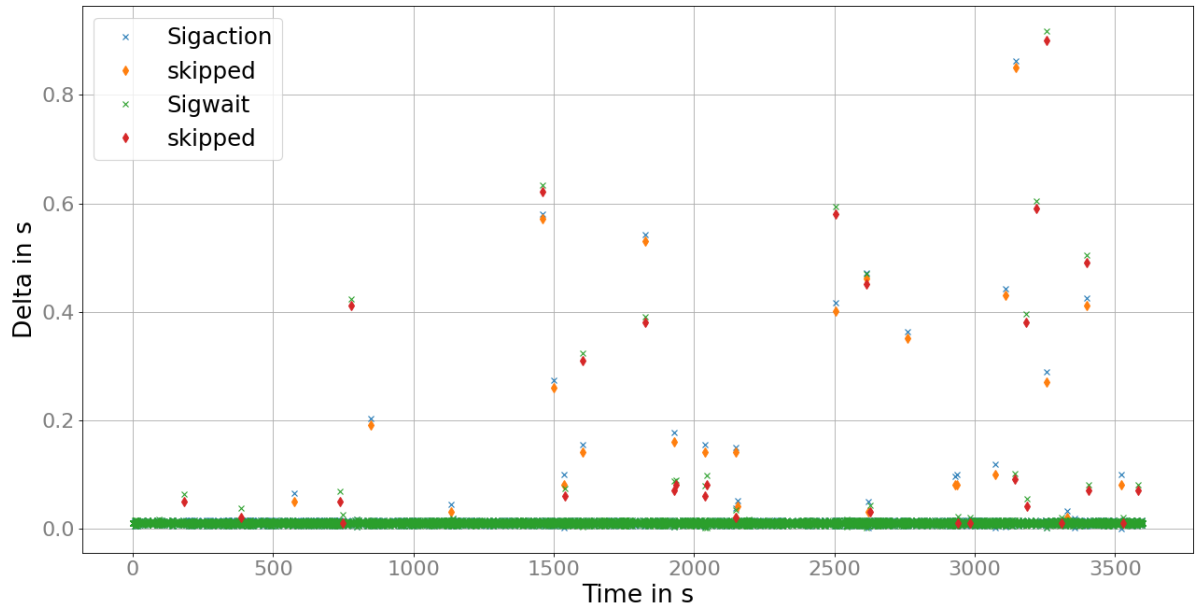


Abbildung 4.1: Signalhandler (Sigaction) vs. blockierendes Warten (Sigwait), getestet auf dem Gateway für 1 h ($f = 1/10$ ms)

Tabelle 4.4: Signalhandler (Sigaction) vs. blockierendes Warten (Sigwait), getestet auf dem Gateway für 1 h

tick (ms)	**strategy**	**mean** (ms)	**max** (ms)	**min** (ms)	**3σ** (ms)	**ticks**	**skipped**	**sum**	**expected**	**time** (s)
10	sigaction	10.016	862.565	0.166	10.669	359408	591	359999	359999	3599.979805
10	sigwait	10.017	918.031	0.574	11.348	359404	595	359999	359999	3599.980148

dem Zählerstand angelegt wird. Je nach Auslösemechanismus wird dies entweder im Signalhandler oder in dem speziellen Thread getan. Die Systemzeit kann nicht durch NTP verfälscht werden, da kein NTP-Client Teil des Buildroot-Projektes ist. Mit den entsprechenden Systemaufrufen kann festgestellt werden, ob das Auslösen der beiden Mechaniken zu lange gedauert hat. Falls der Timer bis zum Zeitpunk, an dem auf ihn reagiert wurde, mehrfach ausgelöst hat wird eine Anzahl von Overruns/Überlaufen ausgegeben.

Abbildung 4.1 zeigt das Delta zwischen je zwei protokollierten Zeitstempeln. Optimal wären Deltas bei stetig konstanten 10 ms. Kernellag bzw. der Scheduler, der den jeweiligen Prozess und dessen Threads auf einem nicht echtzeitfähigen System natürlich u.U. nicht immer sofort wieder aufweckt, sorgen aber für sichtbare Ausreißer. Bemerkenswert ist, dass sowohl in Abbildung 4.1 sowie in Tabelle 4.4 belegt ist, dass das Abrufen und Zählen der Überläufe sich exakt auf die zu erwartenden Ticks/Perioden aufaddiert. In der Abbildung sind erfasste Überläufe durch Rauten markiert, deren Delta mit $Anzahl\_der\_Ueberlaeufe \cdot Delta\_mean$ angegeben ist. Die Tabelle zeigt die Werte des inkrementierten Zählers am Benchmarkende (ticks) sowie die auf-

summierten Überläufe (skipped) und deren gemeinsame Summe (sum). Entsprechend *Benchmarkdauer/Tick_Periode* (expected) ist die Summe selbst nach einem Benchmark über eine Stunde noch exakt.

Die gezeigte Messung wurde auf dem Gateway ausgeführt. Der entsprechende C-Code nutzt die portable POSIX-Schnittstelle, die sowohl in der schlanken uClibc-Bibliothek sowie in der, auf x86 Desktop Systemen verbreiteteren, glibc-Bibliothek zur Verfügung steht. Dementsprechend kann der Code genauso für den Linux-Host kompiliert werden. Erwartungsgemäß fallen dort deutlich mehr Überläufe an (sigaction: 6091 skipped, sigwait: 6072 skipped über eine Stunde). Da der Linux-Host virtualisiert betrieben wird und eine umfangreiche Desktopumgebung, Ubuntu, im Hintergrund läuft, muss der Scheduler das Testprogramm deutlich öfter verzögert fortführen. Wieder ist aber anzumerken, dass selbst in der VM die Hochrechnung mit den Überläufen jede 10 ms-Periode erfasst. In beiden Fällen, Linux-Host und -Target, unterscheiden sich jedoch die zwei Signalhandling-Methoden kaum.

4.4.1.2 Scheduling

Es folgte eine Recherche zum Scheduling unter Linux und den entsprechenden POSIX-Systemaufrufen in den Man-Pages. Dabei wurde offenbar, dass es die Möglichkeit gibt, die Scheduling-Policy des eigenen Prozesses zu beeinflussen. Die Policy `SCHED_FIFO` zählt zu den „real-time“ Optionen. Obwohl es sich dabei um keine echte echtzeitfähige Policy handelt, ist es damit doch möglich, sich zuverlässig stets vor Prozessen, die die Standard-Policy (`SCHED_OTHER`) nutzen, in der Schlange der wartenden Prozesse und Threads einzureihen. Vorsicht ist geboten, falls die eigene Applikation dann in eine aufwendige Berechnung oder einen lang andauernden I/O-Vorgang übergeht. Das kann bedeuten, dass alle anderen laufenden Prozesse keine CPU-Zeit mehr bekommen und abstürzen oder gar das System instabil wird. Deshalb muss der Prozess, der seine Policy auf diese Art ändert, spezielle Rechte besitzen.

Die direkteste Methode einem Prozess zu erlauben seine Policy zu wechseln, ist ihn als Superuser auszuführen. Alternativ dazu kann der Superuser einer Binary explizit das Recht geben, die Policy für sich selbst zu ändern. Dann kann die Binary von anderen Nutzern ausgeführt werden, auch wenn sie ein Umschalten der Policy beinhaltet. Dieses spezielle Recht gehört zu den sogenannten *Capabilities*, die Superuser-Rechte in kleine verwaltbare Gruppen aufteilen. Capabilities können für Nutzer oder Dateien gesetzt werden. Die letztere Möglichkeit erweitert das klassische Rechtesystem nach POSIX, dass nur die Dateirechte `read`, `write` und `execute` kennt. Deshalb muss das genutzte Dateisystem diese zusätzliche Capabilities für Dateien unterstützen. Unter Linux ist dies durch eine entsprechende Flag in der Kernelkonfiguration für z.B. die Dateisysteme *EXT2* und *EXT3* erreichbar. Umfangreiche Linux-Distributionen erlauben dies recht problemlos bzw. integrieren die Option schon von Haus aus. Eine spezifisch angepasste und klein gehaltene Linux-Installation, wie die von Buildroot generierte, benötigt aber

weitere Anpassung. Diverse weitere Konfigurationen im Kernel, am generierten Dateisystem und den genutzten Paketen, für zusätzliche nötige Bibliotheken, sind nötig. Da die Anwendung dieser Arbeit und des resultierenden Produktes in keinem sicherheitskritischen Umfeld geplant ist, wird aufgrund des Umfangs der Anpassungen darauf verzichtet, und die Firmware stets mit Superuser-Rechten gestartet.

Während weiteren Tests mit dem bereits vorher genutzten Programm hat sich das Ändern der Policy als effektiv heraus gestellt. Schnell hat sich aber auch gezeigt, dass es nicht allzu hilfreich ist, die teilweise über 1Mio. Zeilen der Benchmarkdatei zu schreiben, während noch die `SCHED_FIFO`-Policy aktiviert ist. Der genutzte SSH-Server, der die Verbindung offen hält, über die das Programm überhaupt erst auf dem Target gestartet werden konnte, stürzt dann dank fehlender CPU-Zeit einfach ab. Mit ihm natürlich dann auch die SSH-Verbindung und somit das Programm selbst, bevor es den ganzen Benchmark in eine Datei schreiben konnte. Es gilt vor solchen Vorgängen immer wieder auf die Standard-Policy zu wechseln.

Abbildung 4.2 und Tabelle 4.5 zeigen das Ergebnis einer Messung über 3,5 h, während der `SCHED_FIFO` aktiv war. Bei dieser Benchmarkdauer ist der verfügbare RAM auf dem Gateway der limitierende Faktor. Messschriebe müssen im RAM gesammelt werden, da das Schreiben in eine Datei nicht signal-sicher ist. Das heißt es darf nicht in einem Signalhandler stattfinden. Eine Mechanik, die periodisch den Buffer von Messungen aus dem RAM liest und in Dateien speichert, während der Test weiterläuft, entspräche in ihrer Komplexität nicht dem Umfang dieses Tests.

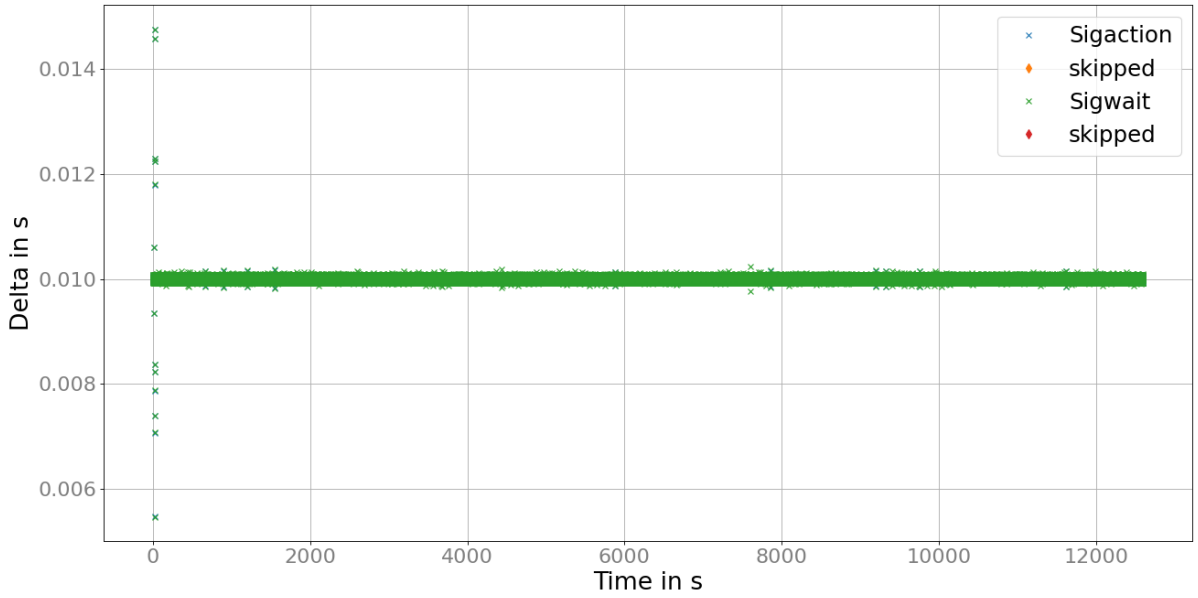


Abbildung 4.2: Signalhandler (Sigaction) vs. blockierendes Warten (Sigwait), getestet auf dem Gateway für 3,5 h ($f = 1/10\,\text{ms}$) mit FIFO-Scheduling

Tabelle 4.5: Signalhandler (Sigaction) vs. blockierendes Warten (Sigwait), getestet auf dem Gateway für 3,5 h mit FIFO-scheduling

tick (ms)	**strategy**	**mean** (ms)	**max** (ms)	**min** (ms)	**3σ** (ms)	**ticks**	**skipped**	**sum**	**expected**	**time** (s)
10	sigaction	10.000	14.740	5.477	10.051	1259999	0	1259999	1259999	12599.979961
10	sigwait	10.000	14.744	5.472	10.073	1259999	0	1259999	1259999	12599.979941

Auf dem ARM-Target hat ein C-`int` 4 B und ein POSIX-`timespec` je zwei C-`long` für Sekunden und Nanosekunden, die ebenfalls je 4 B groß sind. Ein entsprechend ausgerichtetes Struct hat pro Periode also $4\,\mathrm{B} + 8\,\mathrm{B} + 4\,\mathrm{B} = 16\,\mathrm{B}$ für den aktuellen Zählerstand (`int`), den Zeitstempel (`timespec`) und die mögliche Anzahl der Überläufe (`int`). Bei effektiv etwa 55 MB zur Verfügung stehendem RAM ergeben sich zwei Messschriebe, für die zwei Signalhandling-Varianten, mit zusammen $3{,}5\,\mathrm{h} \cdot 1/10\,\mathrm{ms} \cdot 16\,\mathrm{B} \cdot 2 = 40{,}32\,\mathrm{MB}$. Es bleiben zur Sicherheit und für die Systemstabilität noch über 10 MB freier RAM.

Es ist eindeutig zu sehen, dass dank der geänderten Scheduling-Policy kein einziger Überlauf mehr zu Stande kam. Bis auf Ausreißer im Bereich von ± 5 ms, die nur direkt zu Beginn der Messung, womöglich durch Nutzerinteraktion mit dem System, auftreten, sind alle Messpunkte mehr als millisekundengenau. Wird dieses Ergebnis wieder mit dem Linux-Host verglichen, so erreicht dieser bereits nach 10 min Messung, trotz FIFO-Scheduling, über 1000 Überläufe. Ein komplexeres Linux-System profitiert scheinbar also weitaus weniger von der Änderung der Scheduling-Policy.

4.4.1.3 Ergebnis

Mit diesem Ergebnis wird die Wahl, eine eigene Firmware für das Polling auf dem Linux-Target zu schreiben, noch einmal deutlich untermauert. Die Varianten mit Signalhandler und separatem Thread, der auf das Signal wartet, unterscheiden sich kaum. Deshalb wird die zweite Variante mit dem wartenden Thread in Betracht gezogen, da sie nicht so fehleranfällig wie ein Signalhandler ist.

4.4.2 Polling-Strategie

Der zweite wichtige Teil der Architektur ist die Polling-Strategie. Wie bereits erwähnt, werden die meisten OBD- und UDS-Anfragen nach dem Frage-Antwort/Request-Response-Schema behandelt. Hierzu kann physikalisch oder funktional adressiert werden (siehe Kapitel 2.2.4 in den Grundlagen). OBD und UDS sind prinzipiell synchrone Protokolle, das heißt nach einer Anfrage darf die nächste Anfrage erst nach dem erfolgreichen Empfangen einer Antwort gestellt werden. Beide Standards definieren hierfür die Protokollparameter $P2$ und $P2_{max}$. Diese behandeln die Timeouts für Antworten. Nach einer Anfrage muss ein Testgerät maximal die $P2$-Zeit warten. Falls bis dahin keine Antwort empfangen wurde, darf von einem Timeout ausgegangen werden und die nächste Anfrage

kann geschickt werden. Falls eine negative/fehlerhafte Antwort empfangen wurde, muss maximal $P2_{max}$ auf eine weitere positive Antwort gewartet werden.

Die ISO 15765-4 für OBD beschreibt $P2$ als 50 ms während die ISO 15765-3 für UDS bei $P2$ nur von der Summe $P2_{CANServer} + \Delta P_{CAN}$ spricht. Dabei wird nur $P2_{CANServer}$ als 50 ms spezifiziert, ΔP_{CAN} aber als unbestimmter Delay durch z.B. Gateways im Fahrzeugnetz beschrieben. Um die Verwirrung hierbei noch weiter zu steigern, ist in vorliegenden Beispiel-ODX-Dateien $P2$ als Protokollparameter mit 50 ms für OBD und mit 150 ms für UDS angegeben.

Es gilt also diesen Parameter, als Timeout für die zu erwartende Antwort, je nach Anwendung parametrisierbar zu lassen. Besonders relevant wird dieser Parameter außerdem, wenn Anfragen funktional adressiert werden. Hierbei weiß das Testgerät nicht, wie viele Steuergeräte antworten werden. Das heißt es muss auf jeden Fall $P2$ gewartet werden, falls nicht alle acht möglichen Steuergeräte sofort antworten, was unwahrscheinlich ist, da selten alle Steuergeräte dieselben Anfragen unterstützen.

4.4.2.1 Asynchrones Polling

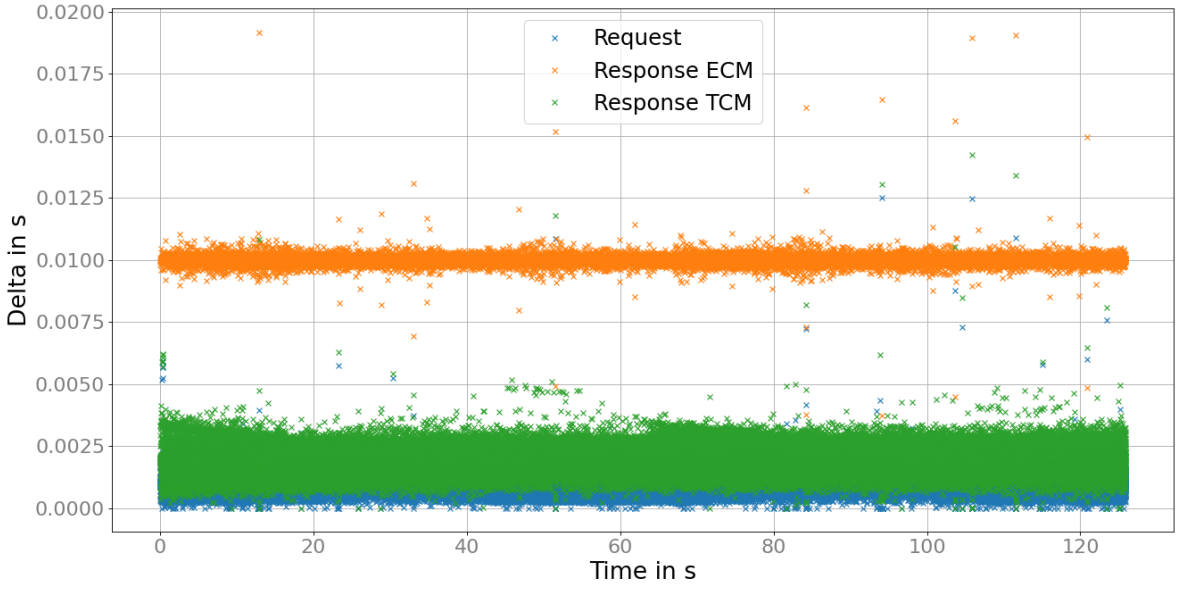


Abbildung 4.3: Asynchrones Polling von ECM und TCM ($f = 1/1\,\text{ms}$)

In der Praxis hat sich gezeigt, dass unterschiedliche Steuergeräte unterschiedlich schnell antworten. Abbildung 4.3 zeigt die zugehörige Messung. Hierbei wurden asynchron, also ohne Antworten abzuwarten, alle 1 ms eine funktionale OBD-Anfrage (Request) nach der Versorgungsspannung der Steuergeräte abgesendet. Der Plot zeigt hierbei je das

Delta zwischen zwei Anfragen. Weiter wurden dann alle Antworten (Response) der beiden Steuergeräte aufgezeichnet und die Zeitstempel, der einzelnen Antworten, von der jeweils vorangegangenen Antwort abgezogen. So ergibt sich, dass das Engine Control Module (ECM) durchschnittlich alle 10 ms antwortet und das Transmission Control Module (TCM) durchschnittlich alle 2 ms. Im Bezug auf die Polling-Rate von 1 ms unterstreicht dies das synchrone Verhalten von OBD, da viele Anfragen unbeantwortet bleiben, wenn das jeweilige Steuergerät die Polling-Rate nicht einhalten kann.

4.4.2.2 Synchrones Polling

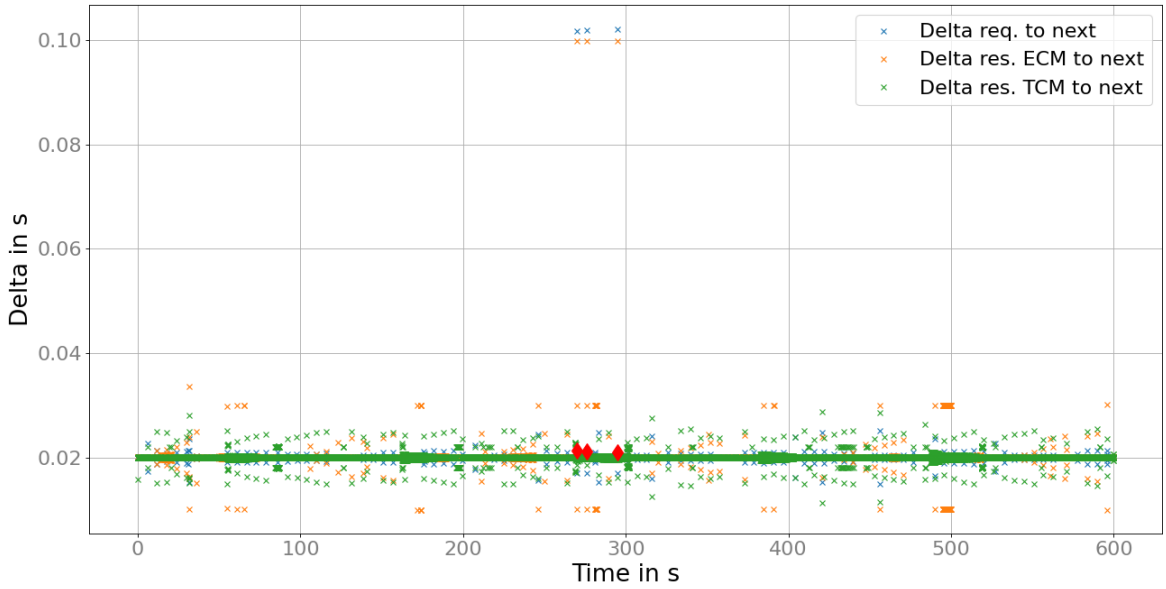


Abbildung 4.4: Synchrones Polling von ECM und TCM ($f = 1/20\,\text{ms}$)

Sinnvoller ist es deshalb, synchrones Polling zu betreiben. Für diesen Test wurde ein Programm mit einem Timer, wie in Kapitel 4.4.1 besprochen, benutzt. Jedes Mal, wenn der Timer auslöst (hier alle 20 ms), wird ein funktional adressierter Request nach der Versorgungsspannung abgesetzt. Da bekannt ist, dass beide verbundenen Steuergeräte auf diesen Request antworten werden, kann dann gewartet werden, bis beide Steuergeräte geantwortet haben. Dann wird wieder auf den Timer gewartet und der nächste Request abgesetzt. Abbildung 4.4 zeigt die Ergebnisse erneut als Zeitperioden zwischen je zwei Requests bzw. je zwei Antworten des jeweiligen Steuergerätes. Rote Rauten markieren Timeouts. Entsprechend der, in der ODX-Datei angegebenen, $P2$-Zeit von 50 ms, wird beim Empfangen der beiden Antworten nach dieser Zeit ein Timeout ausgelöst. Das Ergebnis ist eine, bis auf gelegentliche Ausreißer, konstante Polling-Rate. 99,7 % der Deltas liegen für beide Steuergeräte unter 24 ms.

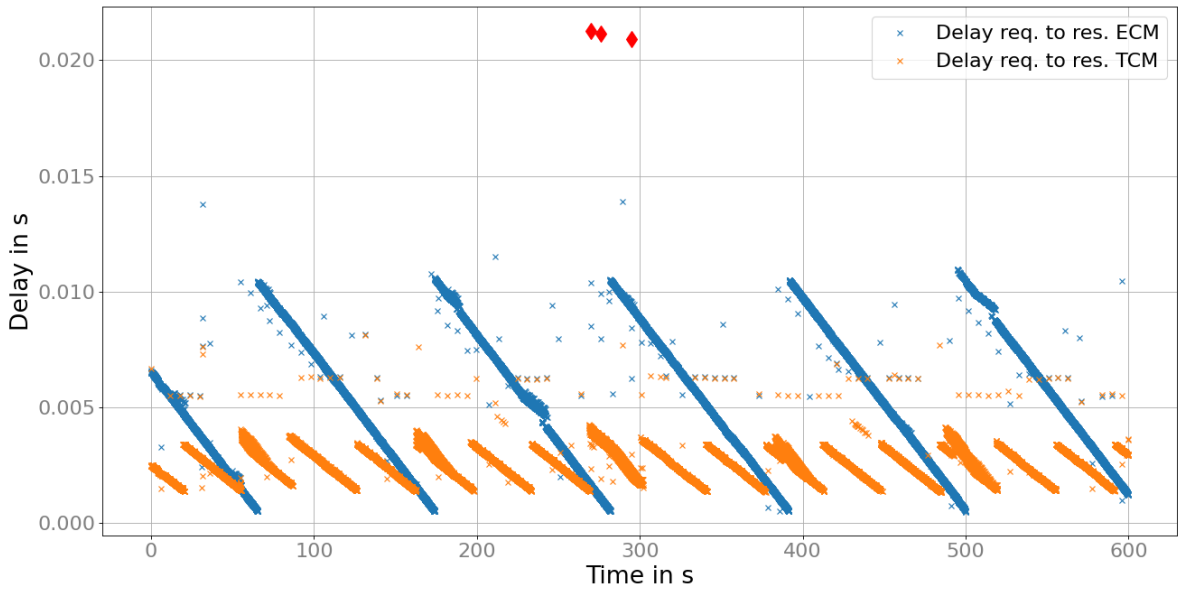


Abbildung 4.5: Synchrones Polling von ECM und TCM ($f = 1/20\,\text{ms}$)

Aus dieser Messung lässt sich ein weiterer Plot ableiten. Abbildung 4.5 zeigt den Delay, zwischen Anfrage und Antwort. Indem der Zeitstempel des abgesetzten Requests mit den Zeitstempeln der Antworten von ECM und TCM verglichen wird, lässt sich bestimmen, wie lange die Steuergeräte im Mittel brauchen, um zu antworten. Hier zeigt sich: Das TCM antwortet mit 2,5 ms im Durchschnitt deutlich schneller als das ECM mit 5,4 ms. Beide Zeitspannen liegen aber im Normalfall weit unter dem $P2$-Timeout.

Eine besonders markante Auffälligkeit ist die „Treppchenbildung“. Diese könnte darauf zurückgeführt werden, dass die Firmware der Steuergeräte in einem fixen Zeitraster arbeitet. Das würde dafür sorgen, dass Polling mit einer Periodenlänge, die kein genaues Vielfaches der Periode eins Rechenzeitrasters der ECU ist, sich bei jedem Anfrage-Antwort Takt, um den Teilerrest der beiden Perioden zum Zeitraster der ECU verschiebt.

4.4.2.3 Ergebnis

Die Tests zur Polling Strategie haben gezeigt, dass synchrones Polling in dieser Anwendung deutlich sinnvoller ist, als asynchrones Polling. Es wird damit vermieden, dass der Bus und die Steuergeräte mit unnötig vielen Anfragen, auf die nie eine Antwort folgen wird, belastet wird. Zudem ist es durch synchrones Polling möglich, Messraster zu realisieren, die deutlich nachvollziehbarer und konstanter sind. Der einzige Fallstrick bei synchronem Polling sind funktionale Anfragen. Hier wäre es optimal stets zu wissen, welche Steuergeräte antworten werden, um sich explizit auf diese zu synchronisieren. In Fällen, in denen das nicht möglich ist, muss der $P2$-Timeout greifen.

4.4.3 Entwurf

Alle bis hierher gesammelten Testergebnisse und Evaluationen, lassen einen Architekturentwurf, vor allem für die Firmware für das Target/Gateway, zu. Die Ergebnisse der Evaluationen zeigen sich vor allem in zwei Designentscheidungen. Ein jeweils eigener Thread für die maximal acht ECUs, die über OBD und UDS erreichbar sind, ermöglicht es, sich unabhängig auf einzelne Steuergeräte zu synchronisieren. Weiter soll eine Liste von *Jobs* dafür sorgen, dass alle aktuellen Mess-/Polling-Aufgaben verwaltet werden können. Ein Job beschreibt alle nötigen Informationen für die Messung eines einzelnen Wertes wie Anfrageadresse, zu erwartende Antwortadressen, die eigentliche Anfrage als ISO TP-Frame, das zu benutzende CAN-Interface, eine eindeutige Identifizierung und die Frequenz für das Polling. Abbildung 4.6 zeigt die schematische Darstellung dieses Entwurfs während Abbildung 4.7 die Struktur der Threads und ihr Zusammenspiel zeigt.

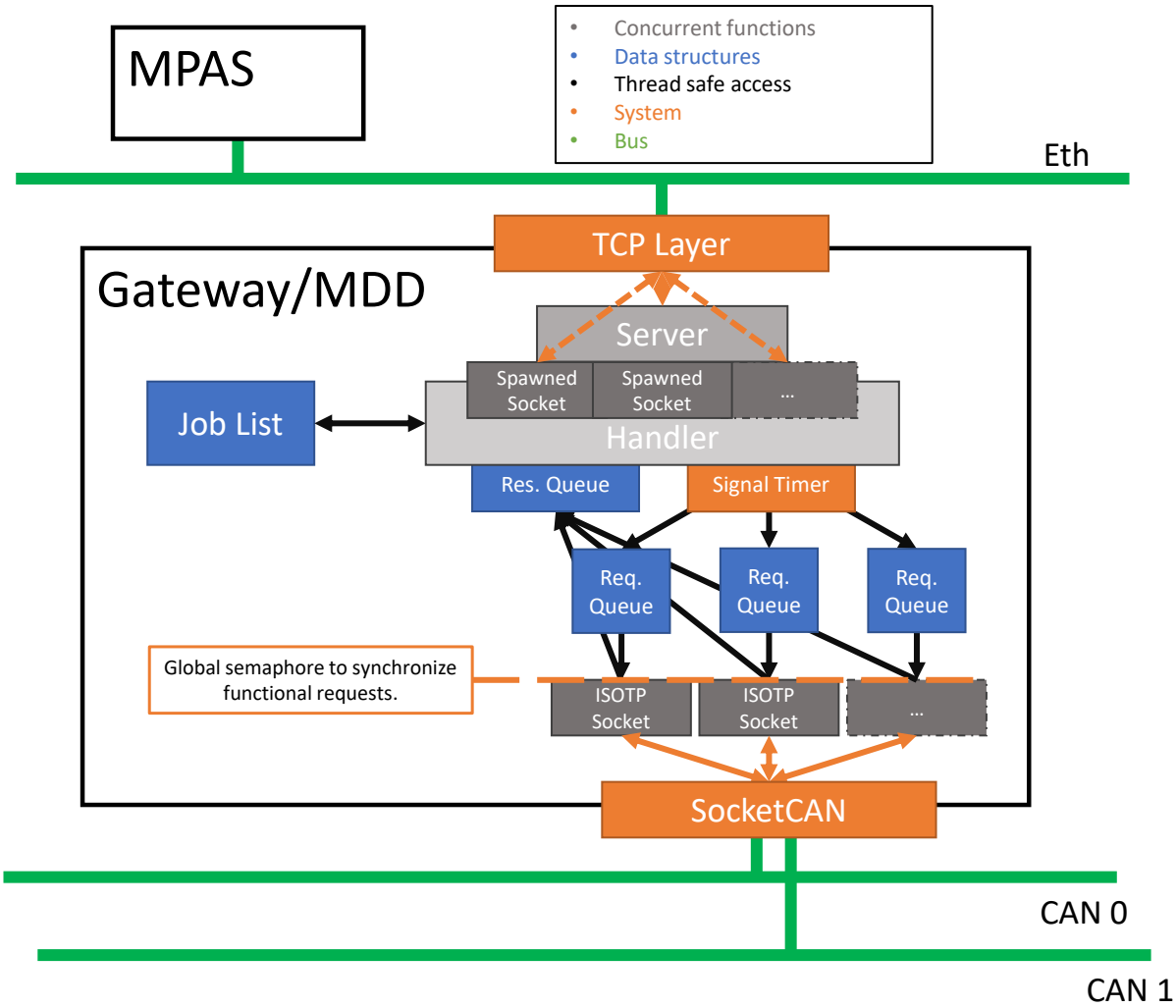


Abbildung 4.6: Konzeption Architekturentwurf

Das MDD implementiert einen parallelen/multi-threaded TCP-Server. Dieser kann über ein einheitliches Socket (im *TCP Server Thread*) von Clienten angesprochen werden und leitet diese dann in zusätzlich generierte Sockets (in *TCP Client Handler Threads*) weiter. Dank der separaten Sockets und Threads können Clienten unabhängig voneinander bearbeitet werden. Über die TCP-Verbindung können Jobs angelegt, editiert, gelesen und gelöscht werden. Spezielle Befehle erlauben es dann, von einem Verwaltungsmodus in einen Messungsmodus zu wechseln, wobei gewählte Jobs scharf geschaltet werden. Ab diesem Moment muss der Client davon ausgehen, gefilterte ISO TP-Nachrichten über die TCP-Verbindung zu empfangen, die seinen aktivierten Jobs entsprechen. Ein entsprechender weiterer Befehl kann die Messung wieder stoppen, woraufhin das Socket wieder die CRUD-Befehle (Create Remove Update Delete) für die Jobs erlaubt.

Ein Timer löst in einem definierten Takt aus, und entblockt einen Thread (*CAN Producer Thread*) entsprechend der Methodik wie in Kapitel 4.4.1.1 besprochen. Dieser Thread liest dann alle aktiven Jobs und wählt diejenigen aus, die entsprechend ihrer eingestellten Frequenz an der Reihe sind. Je nach den im Job hinterlegten zu erwartenden Antwortadressen, werden diese Jobs dann in FIFO-Queues (First In First Out) der einzelnen Polling-Threads (*CAN Polling Threads*) für die betroffenen ECUs gelegt. Die ECU-Threads blocken auf ihrer jeweiligen threadsicheren Queue, bis ein Job anfällt, der gepollt werden muss. Fällt dieser an, wird die Anfrage entsprechend der Daten im Job abgesetzt und auf die Antwort gewartet, oder entsprechend $P2$ in den Timeout gelaufen. Die empfangene Antwort wird in Job-Form mit der entsprechenden ID, aber mit der ISO TP-Antwort anstatt des Anfrage-Frames, in eine weitere FIFO-Queue gelegt. Ein letzter, unabhängiger, Thread (*TCP Response Router Thread*) blockt auf der Queue mit den Antworten. Wenn ein Socket einen oder mehrere Jobs aktiv schaltet, und so in den Messungsmodus fällt, wird in den aktivierten Jobs eine Referenz auf das Socket hinterlegt, das sie aktiviert hat. Wenn nun eine Antwort in der entsprechenden Queue anfällt, kann der dafür verantwortliche *TCP Response Router Thread* nachvollziehen, welcher Socket den zugehörigen Job aktiviert hat und über diesen die Antwort absenden.

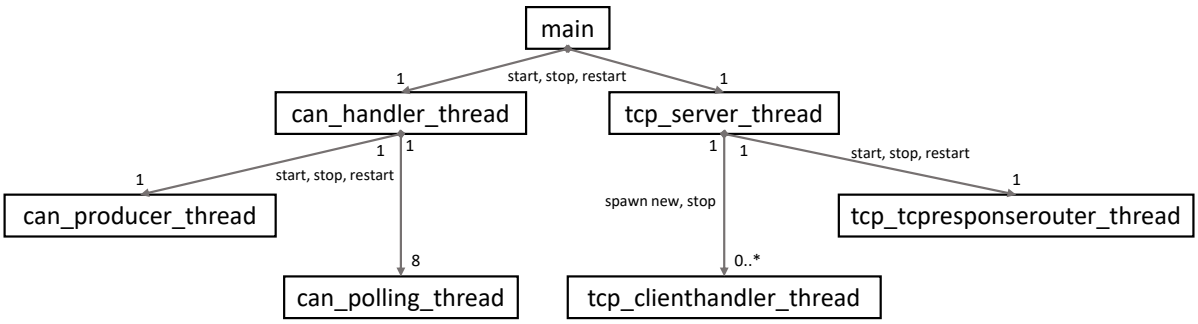


Abbildung 4.7: Threading-Struktur MDD-Firmware

Der Umgang mit funktional adressierten Requests ist eine Besonderheit in diesem Entwurf. Wenn ein entsprechender Job aktiv ist, wird er durch den *CAN Producer Thread*

nicht nur in eine Queue sondern in alle Queues der ECUs gelegt, von denen eine Antwort erwartet wird. Wenn die entsprechenden *CAN Polling Threads* dann auf einen solchen Job stoßen, rufen sie eine Semaphore ab, die sich alle *CAN Polling Threads* teilen. Mit der Information über die Anzahl der zu erwartenden Antworten, kann so nachvollzogen werden, ob alle beteiligten *CAN Polling Threads* bereit sind. Der letzte Thread, der diese Schranke erreicht, setzt die Anfrage auf den CAN-Bus ab. Danach können die Threads wieder unabhängig ihre eigenen Queues bearbeiten. Die Semaphore schützt zudem einen speziellen Job, der als Variable im globalen Scope liegt. In dieser Variablen kann der erste Thread, der auf ein funktionale Anfrage stößt, den zugehörigen Job als Referenz ablegt. Damit kann sichergestellt werden, dass stets alle auf denselben gemeinsamen Request warten.

Weiter muss es möglich sein, auch einzelne, nicht pollende Requests abzusetzen. Hierzu können die *TCP Client Handler Threads* direkt in die Request-Queues schreiben. Indem sie dabei ihre eigene Socket-Referenz in den Job legen, weiß später in der Kette der *TCP Response Router Thread*, welcher Socket die Antwort versenden muss.

Die zusammengefassten Ergebnisse aus der Evaluation lassen sich wie folgt mit dem Design in Verbindung bringen:

- Es wird TCP/IP genutzt, da es kaum langsamer als UDP ist. Je nach Einsatzort/Anwendungsfall des MDD kann sich dazu entschieden werden, Nagle's-Algorithmus zu aktivieren oder zu deaktivieren. Damit können entweder höhere Datenmengen für umfangreichere Messungen oder geringere Latenzen für Anwendungen, in denen das Feedback der Messwerte zeitnäher nötig ist, realisiert werden. Indem sich mehrere Clienten verbinden können, ist es möglich Sockets zu entlasten und nur die an der jeweiligen Stelle relevanten Messwerte über eine Verbindung zu verarbeiten. Tools, wie das geplante MPAS Add-in, können so mit mehreren Instanzen und Sockets für spezifische Aufgaben (z.B. Messen oder Jobverwaltung) arbeiten.

- Die Threads für das Polling und die Generierung der Requests können die sogenannte `SCHED_FIFO`-Policy nutzen um ein genaues konstantes Messraster zu erreichen. Alle Threads und ihre Aufgaben sind so ausgelegt, dass sie mindestens einen blockende Systemaufruf/Queue-Abfrage beinhalten. Damit ist sichergestellt, dass keiner der Threads übermäßig viel CPU-Zeit beansprucht.

- Entsprechend der synchronen Natur der OBD- und UDS-Protokolle sind, wie bereits angesprochen, eigene Threads für jede ECU im Einsatz, um sich spezifisch auf diese zu synchronisieren.

- Threadsichere Queues sorgen dafür, dass selbst im Falle von Verzögerungen nicht gleich Anfragen oder Antworten verloren gehen, da diese gepuffert sind.

5 Umsetzung

Diese Kapitel konzentriert sich auf die Umsetzung der im vorangegangenen Kapitel 4 geplanten Firmware. Zudem wird die Entwicklung eines Add-in für MPAS angesprochen, das mit dem entwickelten MDD kommunizieren soll. Abschließend wird auf das ODX-Dateiformat eingegangen und wie es genutzt wird, um die Toolkette aus Add-in und MDD zu konfigurieren.

5.1 MDD Daemon

Die entwickelte C-Implementierung der eigenen Firmware für das MDD wird als MDD Daemon bezeichnet. Daemon bezieht sich hierbei auf den Zustand des laufenden Programms, da es sich von allen Elternprozessen löst. Das heißt wird das Programm über ein Terminal, SSH oder seriell, gestartet, löst es sich davon und läuft dann als „Dämon“im Hintergrund weiter. Unter Linux bedeutet das, dass der direkte Elternprozess des Programms der `init`-Prozess ist. Dieser ist der Einstiegspunkt für alle Userspace-Anwendungen und wird vom Kernel sofort nach dem Bootvorgang gestartet.

Der `init`-Prozess erlaubt es zudem, nach dem Bootvorgang eine Sammlung von Skripten auszuführen. Über diese kann der MDD Daemon automatisch nach dem Bootvorgang gestartet werden. Falls das Gateway also zurückgesetzt wird, die Spannungsversorgung ausfällt oder Vergleichbares, startet der MDD Daemon automatisch wieder. Zudem können so die CAN-Interfaces des Gateway bereits beim Bootvorgang auf eine bestimmte Bitrate gestellt werden und die Länge der hinterlegten TX-Queues für den CAN-Treiber angepasst werden. Letzteres ist gerade bei der Benutzung des ISO TP-Modul relevant. Hier kann es bei langen Nachrichten passieren, dass das Modul diese in viele CAN-Frames zerlegt und dann alle Frames gleichzeitig in die TX-Queue des Treibers schreibt, ohne dass der physikalische Bus sie alle auf einmal verschicken kann.

5.1.1 Entwicklungsumgebung

Für eine möglichst einfache Entwicklung einer Cross-Plattform-Anwendung wurde eine geeignete Toolkette bzw. Entwicklungsplattform aufgesetzt. Basis hierfür ist der umfangreich durch Extension erweiterbare Texteditor VSCode[1]. Die C/C++ Erweiterung für VSCode bringt bereits viele Features mit, die die Entwicklung für mehrere Zielsysteme

[1] https://code.visualstudio.com/

mit unterschiedlichen Compilern erleichtern. In einer entsprechenden Konfigurationsdatei können mehrere Targets angelegt werden. Für jedes Target kann der Pfad zu Compiler, Linker, Header-Dateien etc. angegeben werden. Wird während der Entwicklung umgeschaltet, sorgt das dafür, dass z.B. Referenzen, automatische Codevervollständigung und Linter die entsprechenden Ressourcen lesen. Der Entwickler wird somit gleich darauf hingewiesen, dass ein bestimmter Systemaufruf oder eine bestimmte Typendefinition für die zwei Targets (hier 1. die Linux-VM mit CAN-USB-Adapter und 2. das Gateway/MDD) aufgrund der unterschiedlichen Kernelversionen oder C-Bibliotheksversionen unpassend ist.

Als Buildsystem wird ein einfaches Makefile genutzt. Damit ist sichergestellt, dass die Firmware später direkt in den Buildroot-Buildprozess eingebunden werden kann. Entsprechende Targets im Makefile unterstützen das Umschalten zwischen den Compilern und Linkern für die x86 VM bzw. das ARM Target auf Basis der SDK, die von Buildroot generiert wurde.

Um laufende Programme zu debuggen kommt der unter Linux verbreitete GNU Debugger (GDB) zum Einsatz. Während GDB klassisch über ein Command Line Interface (CLI) genutzt wird, erlaubt es die C/C++ Erweiterung für VSCode GDB grafisch zu bedienen. Features wie sie aus umfangreichen IDEs (z.B. Visual Studio oder Eclipse) bekannt sind, wie z.B. Breakpoints im Code setzen, Start/Stop/Pause und Variablen Werte beobachten, sind somit direkt im Editor verfügbar. Weiter erlaubt es Buildroot einen GDB-Server zu kompilieren. Dieser ist Teil des Image für das Gateway. Zusätzlich wird in der SDK ein GDB-Client für die Architektur des Entwicklungsrechner/der VM hinterlegt. Die Kombination aus Client und Server erlaubt es, dass der GDB-Server eine Binary auf dem Gateway starten kann und sich der GDB-Client auf dem Entwicklungsrechner dann damit verbinden kann. Der Client benötigt dabei die ARM Binary, die auf dem Gateway läuft. Nun ist es möglich, dass mit der grafischen Oberfläche von VSCode auf dem Entwicklungsrechner das Programm, das auf dem Gateway läuft, remote gedebuggt werden kann. Wie schon für den Linter und das Buildsystem können so zwei Konfigurationen hinterlegt werden, um entweder lokal oder remote zu debuggen.

Zwei einfache Shell-Skripte vervollständigen die Entwicklungsumgebung:

Entwicklungsrechner - lokales x86 Target In VSCode muss das Linter- und Debugger-Target auf die Konfiguration für das lokale Target umgestellt werden. Das Skript nutzt dann das Buildsystem um Objektdateien, die ggf. noch für ARM Architektur sind, zu löschen und daraufhin die Binary neu für x86 zu erzeugen. Danach wird GDB lokal mit der erzeugten Binary gestartet.

Gateway - Remote ARM Target In VSCode muss das Linter und Debugger Target auf die Konfiguration für das Remote-Target umgestellt werden. Das Skript nutzt dann das Buildsystem um Objektdateien, die ggf. noch für x86 Architektur sind, zu löschen und daraufhin die Binary neu für ARM zu erzeugen. Nun wird das Tool `scp` durch das Script gestartet. Es kopiert über eine SSH-Verbindung die erzeugte Binary auf das Gateway. Abschließend ruft das Skript remote über SSH den Befehl

zum Starten des GDB-Servers auf dem Gateway mit der Binary auf. In VSCode kann nun lokal auf der VM der GDB-Client gestartet werden. Er verbindet sich dann mit dem Server für die Debugging-Session.

Eine Entwicklungsumgebung, die es erlaubt remote zu debuggen, ist natürlich besonders hilfreich. Vor allem klassische C-Fallstricke wie Segmentation-Faults und nicht freigegebener Speicher können so einfach untersucht und behoben werden.

5.1.2 Struktur

Bereits in Kapitel 4.4.3 und Abbildung 4.7 wurde eine Struktur für das MDD dargestellt. Diese wurde, bis auf einige Ergänzungen, genauso umgesetzt. Das Programm kann also als die Kombination aus drei Subsystemen betrachtet werden:

Main Hier werden die Subsysteme gestartet und gestoppt, Kommandozeilenargumente behandelt und dafür gesorgt, dass der Prozess, falls erwünscht, als Daemon startet.

TCP-Subsystem Hauptbestandteil ist der *TCP Server Thread*, mit dessen Socket sich Clienten verbinden können. Für jeden Client wird ein separater *TCP Client Handler Thread* gestartet. Angefallene CAN-Frames werden über die entsprechenden Sockets im exklusiv dafür verantwortlichen *TCP Response Router Thread* versendet.

CAN-Subsystem Ein *CAN Handler Thread* ist dafür verantwortlich, die Threads des CAN-Subsystems zu starten und zu stoppen. Für jede ECU nach OBD/UDS gibt es einen separaten *CAN Polling Thread*, um synchron, und ECU für ECU, Anfragen zu versenden und Antworten zu empfangen. Der zeitkritische *CAN Producer Thread* ist abgekapselt und nutzt einen POSIX-Timer, um die Anfragen für die *CAN Polling Threads* im richtigen Moment zu generieren.

Diese Struktur verschweigt jedoch einen logischen Teil sowie die Datenstrukturen für den Austausch zwischen den Threads.

5.1.2.1 Datenstrukturen

Die Queues, in die der *CAN Producer Thread* Anfragen für die *CAN Polling Threads* ablegt, basieren auf einer Bibliothek. Da diese FIFO-Queues performant und threadsicher sein müssen, liegt es nahe, dafür eine bereits etablierte Umsetzung zu nutzen. `rpa_queue`[2] ist eine Umsetzung der `apr_queue`, die im quellenoffenen Apache-Projekt[3] zu finden ist. Da dort aber Apache eigene Wrapper-Funktionen für Threading und Locking genutzt werden, wird die `apr_queue` Implementierung in `rpa_queue` insofern angepasst, dass

[2] https://github.com/chrismerck/rpa_queue
[3] https://www.apache.org/

diese Aufrufe durch POSIX-Äquivalente ersetzt werden. Das heißt die POSIX-Threadbibliothek mit ihren Funktionen zur Verwaltung von Threads und diversen Synchronisationsmechanismen kommt zum Einsatz. Diese Bibliothek fügt sich somit problemlos in das MDD Programm ein, dass ebenso die POSIX-Threadbibliothek nutzt. Die Queue für den *TCP Response Router Thread* basiert auch auf einer `rpa_queue`.

Um die MDD-Jobs zu verwalten ist eine weitere Datenstruktur nötig. MDD-Jobs beschreiben, welche ISO TP-Nachrichten an welche CAN-IDs zyklisch auf den Bus geschrieben werden sollen. Verbundene TCP-Clienten können die Jobs entsprechend der CRUD-Operationen editieren. Da hierzu eine eindeutige ID und schneller Zugriff auf alle Elemente nötig sind, wäre ein einfaches Array, oder, um dynamisch zu sein, eine Linked-List sinnvoll. Ein einzelner Job muss die ISO TP-Nachricht beinhalten, die auf den Bus geschrieben wird. Das heißt bei maximal möglichen 4095 B Nutzdaten wird die Datenstruktur für die Jobs relativ groß, vor allem wenn sie statisch angelegt wird. Dieser Fakt spricht für eine Linked-List. Der Nachteil der Linked-List ist der Zugriff. Alle Elemente müssten nach der je gegebenen eindeutigen ID durchsucht werden. Da vor allem der *TCP Response Router Thread* schnell überprüfen muss, welche Job-ID zum zu versendenden CAN/ISO TP-Frame gehört, um das entsprechende Socket zu ermitteln, fiel trotz des Speicherbedarfs die Wahl auf ein statisches Array. Eine entsprechende Bibliothek wurde umgesetzt, die die eigentlichen Nutzdaten, hier der MDD-Job, in einem Struct mit einer `used`-Flag vereint. Die Array-Indizes fungieren als IDs, so ist der lesende Zugriff maximal performant. Lediglich beim Schreiben neuer Jobs oder dem Löschen und Editieren von bereits Bestehenden, muss das Array nach freien Plätzen durchsucht werden. Durch die `used`-Flag ist dies auf einen einfachen booleschen Vergleich pro Eintrag reduzierbar, wobei die Chance hoch ist, dass nie das ganze Array durchlaufen werden muss. Nachdem diese Datenstruktur als Bibliothek zur Verfügung stand konnte sie auch für den *TCP Server Thread* genutzt werden, der durch eine solche Datenstruktur alle aktiven *TCP Client Handler Threads* verwalten und ggf. stoppen kann.

5.1.2.2 MDD-Befehle/Parser

Der angesprochene logische Teil, der bisher verschwiegen wurde, ist die Verwaltung der MDD-Jobs durch die entsprechenden Befehle an die *TCP Client Handler Threads*. Für diesen Zweck wurde ein menschenlesbares, ASCII-codiertes, Protokoll erarbeitet. Da die Kommunikation über TCP läuft, also ein Stream basiertes Protokoll, müssen einzelne Nachrichten als solche markiert werden. Die ASCII-Zeichen Start of Text (STX) (`0x02`) und End of Text (ETX) (`0x03`) dienen deshalb als Trennzeichen am Anfang und am Ende eines jeden Befehls. Ein Befehl beginnt immer mit drei Buchstaben, A bis Z. Danach folgt, separiert durch Leerzeichen, die zu übertragenden Informationen. Die Informationen sind hierbei immer rein numerisch. Während im Falle eines MDD-Jobs das zu nutzende Polling-Intervall ein einzelner Wert ist, gibt es aber auch Werte, wie beispielsweise die eigentliche ISO TP-Nachricht, die als Array aus Werten betrachtet werden müssen. Für diesen Zweck wird das Komma als Trennzeichen eingeführt. Abschnitte

in einem Kommando, die ein Array bezeichnen, werden so durch Leerzeichen von anderen Elementen im Kommando getrennt, und in sich selbst durch Kommas, die die Array-Werte voneinander trennen. Folgend ist beispielhaft ein Kommando dargestellt.

\0x02 APJ 0 0,1 1,42 10 \0x03

Die drei Zeichen, die den Zweck des Befehls beschreiben, sind ein Akronym für `Add Polling Job`. Es folgen die spezifischen Elemente, wie sie bei `APJ` Kommandos erwartet werden.

1. Der Index des zu nutzenden CAN-Channels. Das genutzte Gateway hat beispielsweise zwei.
2. Die ECUs, die gepollt werden sollen. Falls hier ein kommasepariertes Array angegeben ist, wird der Job als funktionell adressierte Anfrage abgesendet und auf allen gelisteten ECU-Pollern/*CAN Polling Thread* synchronisiert.
3. Die eigentliche ISO TP-Nachricht. Wie im Kontext von CAN oft gebräuchlich, werden diese Nutzdaten mit Basis 16, hexadezimal, interpretiert. `1,42` ist hierbei ein Beispiel für die OBD-Abfrage mit SID 1 und PID 66. Die Antwort ist die aktuelle *Control Module Voltage* des jeweiligen Steuergerätes.
4. Das Polling-Intervall. Es beschreibt, dass die ISO TP-Nachricht alle 100 ms abgesetzt werden soll. Dies ergibt sich aus einem hinterlegten Grundintervall von 10 ms, wobei parametrisierte Jobs immer Vielfache des Grundintervalls, hier also $10 \cdot 10\,\text{ms} = 100\,\text{ms}$, festlegen.

Auf (fast) alle Befehle folgt eine Antwort des MDD zurück an den Clienten. Diese ist wie der eigentliche Befehl aufgebaut, also von STX und ETX limitiert und angeführt von drei Buchstaben. Die drei Buchstaben entsprechen immer denen des jeweiligen empfangenen Befehls. Das erste folgende numerische Element ist immer eine Statusrückmeldung, ob der empfangene Befehl verstanden wurde ($status = 0$) oder es einen Fehler beim Parsen gab ($status < 0$). Nach dem ersten Element folgen wieder befehlsspezifisch ein oder mehrere weitere Elemente. Eine Antwort auf das oben genannte Kommando könnte sein:

\0x02 APJ 0 12\0x03

1. Der Status-Code. Die Null zeigt an, dass der Befehl erfolgreich verarbeitet wurde.
2. Befehlsspezifisches Feedback. Für `APJ`-Befehle beschreibt das zweite Element die ID des angelegten Jobs. Damit können sich später editierende `EPJ-` (`Edit Polling Job`) oder löschende `DPJ`-Befehle (`Delete Polling Job(s)`) auf diesen Job beziehen.

Damit Clienten nicht selbst einen Überblick über alle Jobs intern aktuell halten müssen, kann mit `LPJ` (`List Polling Jobs`) eine Liste aller Jobs in Form ihrer IDs angefordert werden. Danach können mit den IDs mehrere `RPJ` (`Read Polling Jobs`) Kommandos genutzt werden, um alle Informationen über einen Job mit seiner ID abzurufen.

Ein besonderer Fall tritt auf, wenn eine eigentliche Messung gestartet wird. Erst in diesem Moment werden Jobs „scharf“ geschaltet. Das entsprechende Kommando, `BPM` (`Begin Polling Measurement`), kann mehrere Job-IDs beinhalten und liefert keine Kommandoantwort. Stattdessen muss der Client ab dem Moment des Absendens eines solchen Kommandos davon ausgehen nun einen Stream von ISO TP-Frames zu empfangen. Diese Abtrennung ist nötig, da sonst der Client dynamisch auswerten müsste, ob er gerade eine Kommandoantwort oder ein Frame empfangen hat. Bei Polling-Raten von 100 Hz für mehrere Messungen, wäre dies ein unnötiger Engpass. Umgekehrt wären es Leistungseinbusse auf beiden Seiten, wenn CAN-Frames in einem Format versendet werden würden, das den Kommandos ähnelt. Deshalb wird für diese eine direkte binäre Übertragung umgesetzt.

Die gewählte binäre Struktur für ein Frame orientiert sich an der, die durch die originale Peak Firmware realisiert wird. Damit wäre es möglich Softwarekomponenten, die diese Struktur erwarten, zwischen MDD und Peak Firmware austauschbar zu nutzen. Da die originale Struktur einige ungenutzte Bytes aufweist, ist es zudem einfach möglich, diese mit MDD-spezifischen Informationen, z.B. zu welcher Job-ID dieses ISO TP-Frame gehört, zu füllen. Dies erleichtert die Auflösung und Zuweisung des Messwerts für Clienten.

Peak's Struktur sieht zudem einen Zeitstempel mit Nanosekundenauflösung vor. Mit dem entsprechenden Systemaufruf ist es unter Linux möglich einen Hardwarezeitstempel für den genauen Zeitpunkt des Empfangens einer Nachricht, hier des CAN-Frames auf dem Bus, zu ermitteln. Auch wenn die im Vorfeld durchgeführten Messungen zeigen, dass eine genaue Einhaltung des Messrasters durchweg eingehalten wurde, ist dennoch nicht ausgeschlossen, dass es zu Ausreißern kommen kann. Indem die Zeitstempel mit dem Frame übertragen werden, kann der Client dies zusätzlich überprüfen und im Ausnahmefall korrigieren.

Abschließend sei angemerkt, dass es neben den Kommandos für die Jobs und Messungen auch diverse Kommandos gibt, die Einstellungen lesen und editieren. Ein spezielles Kommando erlaubt z.B. eine einmalige, nicht zyklische Anfrage an ein oder mehrere Steuergräte abzusetzen, wobei die Antwort stets ein CAN-Frame für jedes erwähnte Steuergerät ist. Editierbare Einstellungen können die zu nutzende CAN-ID-Länge oder -Baudrate sein.

Durch diese Befehle ergibt sich der angesprochene zusätzliche logische Teil, der zwar kein separater Thread ist, aber doch entsprechend seiner Parser-Funktionalität als extra Submodul namens Parser betrachtet werden kann.

5.1.3 Implementierter Polling-Vorgang

Zusammenfassend ist hier noch einmal dargestellt, wie Clienten das MDD für einen Polling-Vorgang nutzen können.

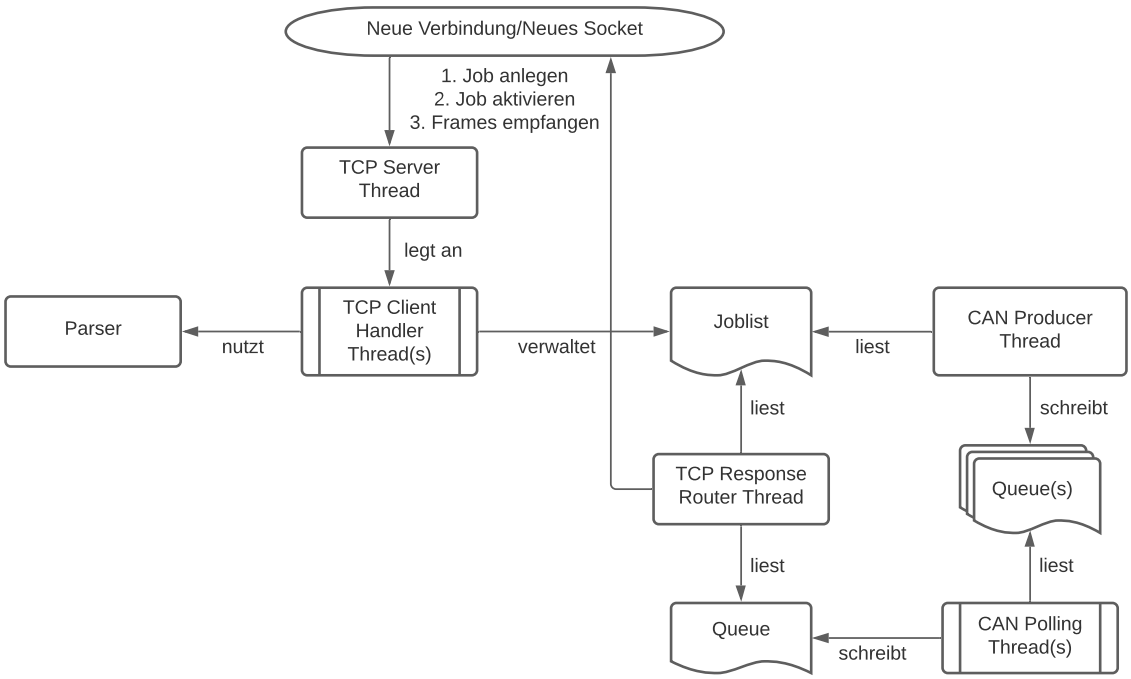


Abbildung 5.1: Schematische Darstellung des Polling-Vorgangs

Der implementierte Polling-Vorgang beginnt mit einem MDD-Job, der über eine TCP-Verbindung angelegt und aktiviert wird, vgl. Abbildung 5.1. Ein TCP-Client verbindet sich mit dem *TCP Server Thread* des MDD und ein neuer *TCP Client Handler Thread* wird für diesen angelegt. Der *TCP Client Handler Thread* empfängt Befehle und interpretiert sie mit Hilfe des *Parsers*. Je nach Befehl wird die *Joblist* gepflegt. Wird ein Befehl zum Aktivieren eines Jobs zur Messung entgegen genommen, wird die entsprechende Flag des Jobs in der *Joblist* gesetzt. Der *CAN Producer Thread* prüft entsprechend seines Timers regelmäßig die aktiven Jobs aus der *Joblist* und schreibt sie, falls ein Polling-Vorgang fällig ist, in die Queues der entsprechenden *CAN Polling Threads*. Die *CAN Polling Threads* blockieren auf ihrer Queue bis ein Job anfällt. Es folgt das Senden der Anfrage via ISO TP-Socket, dann das Empfangen der Antwort. Gibt es beim Empfangen entsprechend $P2$ einen Timeout, wird als Antwort-Frame ein leeres Frame ohne Nachricht und mit Zeitstempel 0 generiert. Das leere Timeout-Frame oder das gültige Antwort-Frame wird dann mit der ID des durchgeführten Jobs und dem aktuellen Zeitstempel der Queue für den *TCP Response Router Thread* übergeben. Der *TCP Response Router Thread* wartet auf anfallende Antworten in seiner Queue. Fällt eine Antwort an,

kann er dank Job-ID nachvollziehen, welcher Socket diesen Job aktiviert hat und das Frame an den verbundenen TCP-Client senden.

5.1.4 Besonderheiten

Während der Umsetzung und abschließenden Validation des Daemon sind diverse Stolpersteine aufgetreten, wie sie bei derartigen Projekten eigentlich immer zu erwarten sind. In diesem Abschnitt werden diese Probleme, etwas eleganter als Besonderheiten betitelt, diskutiert und gelöst. Diese Besonderheiten stellen somit einen Ausreißer aus dem V-Modell dar. Da einige Anpassungen im Zuge der Besonderheiten nicht Teil der ursprünglichen Konzeption waren, haben sie zu zusätzlichen Iterationen über die Konzeptions- und Umsetzungsschritte des V-Modells geführt. Besonders auffällig ist dies bei der Realisierung der funktionalen Anfragen. Nicht nur musste für sie ein Kernelmodul erweitert, sondern auch ein zusätzlicher, dedizierter Thread für diese Anfragen eingeführt werden (siehe Abschnitt 5.1.4.3).

5.1.4.1 Daemon

Zur Wiederholung: Ein Daemon unter UNIX-Betriebssystemen ist ein Prozess, dessen Elternprozess der `init`-Prozess ist. Er ist von jeglichen anderen Prozessen losgelöst. Eine Binary, die aus C-Code kompiliert wurde und nur eine Main-Routine hat, ist standardmäßig ein Kindprozess zum Prozess, der die Binary gestartet hat. Oft wird dies eine Eingabeaufforderung/Shell in einer lokalen oder remote Konsole sein, die die Binary/das Programm startet. Das MDD soll sowohl manuell über eine Shell, als auch automatisch beim Betriebssystemstart durch ein `init`-Skript gestartet werden können. Es wird deshalb eine Flag `-d` eingeführt, die beim Ausführen der Binary des MDD angehängt werden kann, wodurch diese als Daemon startet. Indem der Daemon beim Start eine bestimmte Datei prüft und ggf. sperrt, ist sichergestellt, dass immer nur eine Instanz läuft. Diese Datei kann zusätzlich dazu genutzt werden die PID der laufenden Instanz abzulegen. Der damit sichergestellte leichte Zugriff auf die PID erlaubt es Systemtools die laufende Instanz einfach ausfindig zu machen und ihr Signale, z.B. zum Beenden, zu schicken. Der MDD Daemon erlaubt selbst Flags, um möglichen laufenden andere Instanzen die `SIGTERM` (`-t`) oder `SIGQUIT` (`-q`) Signale zu schicken. Ersteres löst ein geplantes Beenden des MDD Daemon aus, Letzteres erzwingt es.

Um einen Prozess als Daemon zu starten, muss dieser einige prozessspezifische Einstellungen für sich selbst vornehmen und sich durch ein `fork` vom Elternprozess lösen, woraufhin sich der Elternprozess selbst beenden muss. Dieses Abkoppeln und Verwaisen, in Kombination mit besagten Einstellungen, wie z.B. dem Anpassen der Maske für POSIX-Signale, ist ein weit verbreitetes Konzept, für das es vorgefertigte Code-Snippets gibt. Der MDD Daemon implementiert eine angepasste Mischung aus den Vorlagen von [Ste90] und [WW06].

Eine Problematik mit Prozessen, die im Hintergrund ohne direkte Nutzerinteraktion agieren, ist das Logging. Applikationen, die in einem Terminal laufen oder eine UI aufweisen, können den Nutzer über diese mit Informationen über den aktuellen Zustand der Applikation informieren. Ein Daemon hat diese Möglichkeit nicht. Auch hier greift wieder POSIX. Es fordert Systemaufrufe/eine API um Nachrichten an den Systemlogger zu übermitteln. Eine Applikation kann die entsprechenden Systemaufrufe nutzen, um sich mit dem Systemlogger zu verbinden und Nachrichten an ihn zu übermitteln. Der Systemlogger ist meist ein lokal ausgeführter Daemon, der die Anforderungen des `syslog`-Protokolls erfüllt. Die Buildroot-Umgebung des MDD Daemon ist so konfiguriert, dass die Rsyslog[4] Implementierung genutzt wird.

Eine Nachricht an den Systemlogger beinhaltet die eigentliche Nachricht sowie eine Facility und einen Loglevel. Der Loglevel, auch Severity genannt, hilft den Überblick bezüglich der Relevanz von Logeinträgen zu behalten. `syslog` unterscheidet u.a. zwischen `Debug`-, `Informational`-, `Notice`-, `Warning`-, `Error`-Nachrichten und `Critical`-Nachrichten. Kritischere Level wie `Alert` und `Emergency` werden im MDD Daemon nicht genutzt.

Folgend wird beschrieben, wie die einzelnen Loglevel im MDD-Kontext genutzt werden.

Critical Ein kritischer Fehler, der darauf hinweist, dass der MDD Daemon nicht ausgeführt werden kann. Prinzipiell nicht behebbare Fehler, beispielsweise unzureichender RAM für alle internen statischen Datenstrukturen, kein CAN-Interface etc.

Error Ein Fehler, der darauf hinweist, dass der MDD Daemon nicht ausgeführt werden kann. Prinzipiell manuell behebbare Fehler, z.B. verfügbare aber uninitialisierte Interfaces für CAN und Ethernet, zu stark ausgelasteter CPU durch andere Anwendungen etc.

Warning Eine Warnung über einen internen Fehler, den die Applikation selbst beheben kann, der aber beobachtet werden sollte. Hierzu gehören z.B. Fehler beim Anlegen von Sockets auf bereits belegten Ports. Es kann passieren, dass TCP-Ports einige Sekunden lang belegt bleiben, auch wenn die zugehörige Applikation bereits beendet wurde. Falls dies der Fehler ist, kann der MDD Daemon ihn selbst beheben, indem er regelmäßig versucht, sich erneut an den Port zu binden. Natürlich kann es sich genauso gut um einen Konfigurationsfehler handeln, deshalb sollten diese Warnungen beobachtet werden.

Notice Eine Meldung über die Änderung des Systemzustands, die in der normalen Benutzung selten vorkommt, aber festgehalten werden sollte. Hierzu gehören beispielsweise das Neuladen der Konfiguration, das Neustarten des CAN und TCP-Submoduls etc.

[4] https://www.rsyslog.com/

Informational Gelegentliche Meldungen zum Systemzustand, wie die Anzahl der aktiven Messungen oder verbundenen TCP-Clienten.

Debug Umfangreiche Meldungen, die beim Debugging des Systems helfen. Sie sollten während der normalen Benutzung stets deaktiviert sein, da sie so häufig vorkommen, dass sie die Performance des Programms beeinträchtigen können. Hierzu gehören z.B. Nachrichten beim Empfang jedes ISO TP-Frames.

Die Facility einer Lognachricht kann laut RFC 3164 z.B. `kernel messages`, `user-level messages` oder `mail system` sein. Für spezifische Userspace-Applikationen, die zu keiner der definierten Facilitys passen, stehen die Facilitys `local use 0` bis `local use 7` zur Verfügung.

Dank der umfangreichen Konfigurationsmöglichkeiten von Rsyslog können Lognachrichten nach Wunsch gefiltert, formatiert und sogar über das Netzwerk weitergeleitet werden. So sieht die Konfiguration in der Buildroot-Umgebung für das MDD z.B. vor, dass alle Nachrichten des MDD Daemon in einer separaten Logdatei abgelegt werden. Hinzu kommen Hilfsfunktionen in der MDD Daemon-Implementierung, die es erlauben Nachrichten von bestimmten Submodulen, z.B. dem Parser, auszublenden. In Summe ist so umfangreich festlegbar, welche Lognachrichten in den unterschiedlichen Entwicklungsschritten und später in der Anwendung beim Kunden ausgegeben und gespeichert werden sollen.

5.1.4.2 CAN-Channel

Das Gateway von Peak verfügt über zwei separate CAN-Interfaces, an die zwei unabhängige CAN-Busse angeschlossen werden können. Während der Entwicklung des MDD Daemon wurde sich zuerst nur auf ein Interface konzentriert, da Fahrzeuge auf dem Prüfstand meist nur über eine OBD/UDS fähige, frei zugängige CAN-Buchse verfügen. Dennoch schien es nachlässig, das zweite Interface völlig zu ignorieren. Eine vergleichsweise kleine Anpassung wurde deshalb vorgenommen. Hierzu müssen Jobs, die per Kommando angelegt werden, definieren, welches Interface genutzt werden soll. Der MDD Daemon legt dann intern nicht nur acht Polling-Threads für acht mögliche ECUs an, sondern $Anzahl\_Interfaces \cdot Anzahl\_ECUs$ Polling-Threads. Damit wäre es theoretisch möglich den MDD Daemon für Hardware mit beliebig vielen Interfaces zu kompilieren.

5.1.4.3 Funktionale Anfragen

Eine Besonderheit, die vor allem im Kontext von OBD genutzt wird, sind funktionale Anfragen. Hierbei wird nicht eine spezifische ECU mit einer Anfrage zu einer Antwort angeregt, sondern mehrere ECUs durch eine Anfrage stimuliert, woraufhin mehrere Antworten folgen. In der vorab entwickelten MDD Daemon-Architektur wurde dies zwar bereits bedacht, das Problem ist es aber wert umfangreicher beleuchtet zu werden.

Funktionale Anfragen bereiten zwei Herausforderungen:

Broadcast ISO TP-Sockets sind entsprechend der ISO TP-Definition prinzipiell 1-to-1 Verbindungen mit einer spezifischen Quell- und Zieladresse. Eine funktionale Anfrage ist als Broadcast und somit als 1-to-n Verbindung zu verstehen. Eine funktionale Anfrage über ein ISO TP-Socket kann dementsprechend nur so realisiert werden, dass die Zieladresse die funktionale OBD/UDS-CAN-ID ist und die Quelladresse eine beliebige, aber nicht anderweitig benutzte CAN-ID. Es ist nicht unmöglich eine solche beliebige, aber nicht benutzte CAN-ID zu bestimmen und festzulegen. Das Fehlerpotential ist aber dennoch deutlich.

Eine weitere Möglichkeit wäre ein CAN-RAW-Socket. Diese benötigt keine feste Quell- und Zieladresse, sondern nur die CAN-ID für das zu versendende Frame, effektiv also nur die Zieladresse.

Da das ISO TP-Modul in seiner Dokumentation explizit auf die Verwendung von ISO TP für OBD und UDS hinweist, wirkt es aber wie ein Versäumnis, dass die Implementierung des Moduls solche Broadcasts, bei denen keine Quelladresse definiert sein muss, nicht unterstützt. Deshalb wurde sich mit dem Entwickler des Moduls über die entsprechende Kernel-Mailliste in Verbindung gesetzt und die Implementierung dieser zusätzlichen Funktionalität angeregt. Da das Modul zum Zeitpunkt dieser Arbeit kurz davor steht in den Mainline-Kernel aufgenommen zu werden, ist die Kommunikation um das Modul besonders aktiv und so konnte die Lösung schnell implementiert werden. Neben der Anregung zu dieser Erweiterung[5] konnte der Autor dieser Arbeit einen kleinen Patch dazu beitragen, der verhindert, dass die normalerweise stattfindenden Checks der Empfangsadresse bei einem Broadcast übersprungen werden.

Mit dem nun gepatchten Kernelmodul wird neben den acht Polling-Threads je CAN-Interface ein zusätzlicher Broadcast-Thread eingeführt.

Synchronisation Dank der Broadcast-Funktionalität kann jetzt eine Anfrage abgesetzt werden. Nun müssen die Antworten verarbeitet werden. Die MDD Daemon-Architektur sieht separate Sockets für alle ECUs, die antworten könnten, vor. Im Falle einer funktionalen Anfrage müssen alle diese Sockets bereit sein und synchron auf die Antworten der einen einzelnen Anfrage warten. Dieser gemeinsame Job muss unter allen beteiligten Polling-Threads synchronisiert sein, da er sonst mit weiteren aktiven, physikalisch adressierten, Jobs kollidiert.

Der nächste Abschnitt beschreibt das Vorgehen für diese Synchronisation. Es werden drei Synchronisationsmechanismen aus der POSIX-Threadbibliothek genutzt, die hier als kurzer Einschub erläutert werden sollen.

Mutex Ein Mutex, oder auch Lock, ist die klassische Variante zur Synchronisation und für Threadsicherheit. Das Sperren und Entsperren eines Mutexes ist eine atomare

[5] https://lore.kernel.org/linux-can/20201206144731.4609-1-socketcan@hartkopp.net/T/#u

Operation und kann somit helfen Zugriff auf andere Variablen threadsicher zu gestalten. Das Vorgehen hierfür ist, dass stets der Mutex gesperrt werden muss, bevor auf die Variable oder den kritischen Codebereich zugegriffen wird. Ist der Mutex bereits durch einen anderen Thread gesperrt, muss gewartet werden, bis er wieder entsperrt ist.

Barrier Eine klassische Barriere, eng. Barrier, lässt Threads an ihr warten, bis eine bestimmte Anzahl von wartenden Threads erreicht ist. So kann sichergestellt werden, dass die Threads aufeinander warten um dann synchron weiterzulaufen.

Condition Eine Bedingung, eng. Condition, ist eine Kombination aus einem klassischen Lock/Mutex und einem Benachrichtigungsmechanismus. Erreicht ein Thread einen Conditional-Lock, versucht er einen gegebenen Mutex zu sperren und dann eine Bedingung zu prüfen. Diese Bedingung kann durch jeden beliebigen Codeabschnitt realisiert werden und ist dabei dank des Mutexes threadsicher. Natürlich sollte der Mutex selbst in diesem Codeabschnitt nicht verändert werden. Ist die Bedingung wahr, wird der Mutex gehalten und ein, durch die Bedingung und Mutex geschützter, Codeabschnitt kann ausgeführt werden. Danach muss der Mutex wieder freigeben werden. Ist die Bedingung unwahr, wird der Thread blockiert und wartet auf der Bedingung. Währenddessen gibt er den Mutex wieder frei. Nun kann ein anderer Thread den Mutex ergreifen um mögliche Variablen, die z.B. Teil der Bedingung sind, zu verändern. Danach sendet der Thread eine Benachrichtigung an alle Threads, die auf der Bedingung warten. Die geweckten Threads können die Bedingung dann erneut prüfen und ggf. den kritischen Codeabschnitt ausführen. Diese Mechanik erlaubt es threadsicher auf einer beliebigen Bedingung zu warten. Danach ist zudem sichergestellt, dass der geweckte Thread, der die Bedingung als wahr auswertet, den geteilten Mutex hält.

Eine Synchronisation aller Polling-Threads beeinträchtigt den großen Vorteil der separaten Polling-Threads, die unabhängig voneinander schnell pollen können. Deshalb muss ein Job, der funktional adressiert, festlegen, welche ECUs antworten werden. Falls dies bekannt ist, kann damit die bremsende Wirkung auf die anderen Polling-Threads minimiert werden (vgl. Kapitel 4.4.3). Falls unbekannt ist, welche ECUs antworten werden, müssen einfach alle Acht im Job angegeben werden. Falls kein ECU angegeben wird, handelt es sich um einen Broadcast ohne Antwort. Dieser Fall ist nützlich für die UDS-Nachricht „Tester Present“.

Die Synchronisierung bezieht sich immer auf den Broadcast-Thread des jeweiligen CAN-Interface sowie die Polling-Threads für die geforderten ECUs. Zuerst wird sichergestellt, dass alle Threads bereit sind. Dazu wird die funktionale Anfrage in die Queues aller beteiligten Threads gelegt. Sobald ein Thread einen funktionalen Request aus seiner Queue nimmt, vergleicht er diesen mit einer Variablen im globalen Scope, die einen Request beinhaltet. Diese Überprüfung ist durch eine Condition geschützt. Damit ist sichergestellt, dass alle Threads auf denselben funktionalen Request warten. Falls der glo-

bale Request nicht dem soeben der Queue entnommenen entspricht, wartet der Thread. Falls es derselbe Request ist, läuft der Thread an die synchronisierende Barriere. Der Broadcast-Thread ist dafür verantwortlich, den globalen Request zu beschreiben, sobald bei ihm ein funktionale Request in der Queue anfällt. Nach dem Schreibvorgang wird die Condition-Benachrichtigung ausgelöst. Dementsprechend werden alle wartenden Polling-Threads aufgeweckt, worauf sie wieder prüfen, ob dies nun der richtige Request für sie ist. Jeder bereite Thread läuft an die Barriere und inkrementiert diese. Die Barriere wurde vorab durch den Broadcast-Thread auf die Menge der zu erwartenden, antwortenden ECUs plus sich selbst initialisiert. Öffnet sich die Barriere, sendet der Broadcast-Thread die Anfrage und alle beteiligten Polling-Threads überspringen ihren normalen schreibenden Vorgang und gehen sofort zum Lesen mit $P2$-Timeout über.

5.1.4.4 Subsystem Neustarts

Eine letzte umgesetzte Besonderheit des MDD Daemon ist das Neustarten einzelner Subsysteme. Dieses wurde bereits von Anfang an angedacht, so dass sich der MDD Daemon bei bestimmten Fehlern selbst wieder in funktionellen Zustand bringen kann. Weiter wird vom MDD Daemon gefordert, dass er zwischen 11 bit CAN-Adressierung und 29 bit wechseln kann. Hierzu müssen die ISO TP-Sockets neu initialisiert werden. Falls die Baudrate gewechselt werden soll, trifft dasselbe zu. Hierzu bietet es sich an, einfach das ganze CAN-Subsystem neu zu starten, bzw. zu stoppen, die Anpassungen vorzunehmen und dann wieder zu starten.

Threads, die für Subsystemneustarts neugestartet werden sollen, sind im Kontext des MDD Daemon stets solche, die eine Initialisierung haben, bevor sie in eine dauerhafte Schleife übergehen. Um sie neu zu starten, muss jeder Thread eine Möglichkeit bieten, diese Schleifen zu verlassen. Die direkteste Lösung sind binäre Flags, die bei jedem Schleifendurchlauf geprüft werden. Dies ist allerdings problematisch, wenn der Thread, wie einige der TCP-Threads, auf einer empfangenden Operation ohne Timeout wartet. Die POSIX-Threads bieten hierzu die Möglichkeit Threads zu unterbrechen. Hierbei ist darauf zu achten, dass Threads die so unterbrochen werden, ihre Ressourcen, z.B. Sockets, richtig freigeben. Hierzu können bestimmte Aufräumfunktionen definiert werden, die bei Beendigung eines Threads, ob durch `exit`, `return` oder externes Unterbrechen/`cancel`, aufgerufen werden.

Im Falle der CAN-Threads ist dies einfacher, da diese stets nach spätestens $P2$ in Timeouts laufen und so nicht explizit unterbrochen werden müssen. Ansonsten blockieren die CAN-Polling-Threads nur beim Lesen ihrer Queue. Die genutzte Queue-Implementierung besitzt bereits explizite Funktionen, um auf ihr wartende Threads aufzuwecken.

Die Kombination aus `cancel`, Lesen mit Timeout und die Queue-Implementierung, die es erlaubt auf ihr wartende Threads aufzuwecken, erlaubt es also alle Subsysteme des MDD Daemon separat von einander, und vor allem ohne mögliche Speicherlecks, neu zu starten. Das korrekte Freigeben der Ressourcen liegt natürlich in der Hand des Programmierers, hiermit soll aber festgehalten sein, dass alle Methoden zum Unterbrechen

der Threads eine Möglichkeit bieten, explizite Aufräumaktionen vorzunehmen. Im Falle des Unterbrechens durch `cancel` wird durch die speziellen Funktionen aufgeräumt. Im Falle der einfachen Flag für Schleifen mit Timeout wird durch simple Aufrufe nach der Schleife, bevor `exit` aufgerufen wird, aufgeräumt.

5.2 MDD MPAS Add-in

Der zweite große Teil dieser Arbeit ist das MPAS Add-in, das das MDD steuern soll. Entsprechend der Architektur des MDD Daemon muss das Add-in eine TCP/IP-Verbindung mit dem MDD aufbauen und den dort laufenden Daemon mit den entsprechenden Befehlen parametrisieren. Wenn eine Messung gestartet wird, muss das Add-in den entsprechenden Befehl absetzen und dann den darauf folgenden Stream aus ISO TP-Frames auflösen. Das MDD wurde so realisiert, dass es prinzipiell kein Protokollwissen über OBD oder UDS hat, sondern lediglich Anfrage-Antwort-Polling-Operationen auf bestimmten CAN-IDs ausführen kann. Damit ist das MDD Add-in für MPAS dafür verantwortlich, die entsprechenden SIDs und PIDs bzw. LEVs zu ermitteln und die Anfrage-Frames zu generieren. Ob es sich bei den Antwort-Frames dann um positive oder negative Antworten handelt muss das Add-in ermitteln. Mit dieser Designentscheidung liegt die Protokollkomplexität auf der Seite des Leitrechners, der deutlich performanter ist als der einkernige ARM-Prozessor des MDD. Außerdem können so zukünftige neue Protokolle, die auch auf das Anfrage-Antwort-Prinzip setzen und ISO TP nutzen, u.U. direkt im Add-in erweitert werden, ohne dass die Software des MDD verändert werden muss.

5.2.1 MDD Tool

Bereits während der Entwicklung des MDD Daemon schien es sinnvoll ein Tool zu entwickeln, um diesen zu testen. Um Teile des Tools später direkt für das MPAS Add-in wiederverwenden zu können, wurde das Tool im .Net-Framework entwickelt. In Zukunft ist durch PA-Systems vorgesehen, MPAS in die aktuellere Programmiersprache C# zu übersetzen. Sowohl Visual Basic als auch C# nutzen das .Net-Framework und die Microsoft Common Language Runtime und können daher gemischt in einem .Net-Framework Projekt auftauchen. Das MDD Add-in bietet sich deswegen als Proof-of-Concept an, ob ein neuer Gerätetreiber, der in C# geschrieben ist, problemlos in das Visual Basic basierte MPAS einzubinden ist. Dementsprechend wurde das MDD Tool in C# entwickelt. Um dem Trend der aktuellsten Technologien für das MDD Tool zu folgen, wird die UI des MDD Tool in Windows Presentation Foundation (WPF) umgesetzt, dass das ältere Windows Forms ersetzen kann. WPF zeichnet sich vor allem dadurch aus, dass die UI im XAML-Format (Extensible Application Markup Language) beschrieben wird. Da XAML, ähnlich wie Hypertext Markup Language (HTML), auf XML basiert, ist

die Idee hinter WPF, dass die Entwicklung von Desktopapplikationen mit UI mehr und mehr der Entwicklung von Webanwendungen ähneln sollte.

Die Kombination aus C# und WPF wird oft im sogenannten Model View ViewModel (MVVM) Entwurfsmuster entwickelt. Das MVVM ist Microsofts Abwandlung des Model View Controller (MVC)-Musters. Prinzipiell geht es, ähnlich wie in der Webentwicklung, darum, dass die UI möglichst von den Daten der Applikation getrennt wird. Der View, die UI, sollte nie direkt auf die Daten im Modell (eng. Model) zugreifen. Die Logik zwischen beiden, Controller oder ViewModel genannt, stellt die Schnittstelle zwischen UI und Modell dar. Sie ist damit eine zusätzliche Abstraktionsebene. Falls die UI also über Datenänderungen informiert werden muss oder sich Daten in der UI ändern, die im Model widergespiegelt werden sollen, kümmert sich der Controller/das ViewModel um die Synchronisation. Ändert sich die UI im Quellcode, muss sich nur der Controller anpassen, nicht das Model. Umgekehrt muss sich bei Modelländerungen, z.B. Datenbankwechsel, auch nur der Controller anpassen, nicht aber die UI. Eine Applikation kann mehrere Views, Modelle und Controller beinhalten, wobei jede Implementation eines Views, Modells oder Controllers in einer separaten Klasse umzusetzen ist.

Das MDD Tool soll entsprechend dieses Entwurfsmusters entwickelt werden.

5.2.1.1 Modelle

Die ersten beiden Modelle, die für den Austausch mit dem MDD nötig sind, sind ein Modell um die Informationen über einen Polling-Job und ein Modell um ein ISO TP-Frame zu halten. Das erste Modell (Klassenname `MDDJobModel`) muss dazu im Stande sein, die Information zu einem Polling-Job in ein *Add*/*Edit*/*Remove*-Kommando für das MDD umzusetzen und umgekehrt empfangene Kommandoantworten zu verstehen. Falls eine Messung läuft, wird vom ASCII-codierten Kommandoformat zu binär kodierten ISO TP-Frames gewechselt. Die `MDDTcpCanFrameModel` Klasse muss den Bytestream dieser Frames verstehen und in einfach verarbeitbare Klassenfelder umsetzen.

Ein drittes Modell ist weniger direkt und vielleicht eher als Controller einzuordnen. `MDDConnectionModel` beschreibt die Verbindung zu einem MDD. Neben dem Adress-Tupel implementiert das Modell die verschiedenen Methoden zum Auf- und Abbauen der Verbindung. Ein implementiertes Event bietet die Möglichkeit auf Verbindungsabbrüche zu hören. Weiter setzt die Modellklasse Methoden um, die ASCII-codierte Anfragen absetzen und ASCII-codierte Antworten empfangen können. Spezielle Controller können diese Methoden dann nutzen, um die nötigen Operationen auf dem MDD auszuführen. Da nur diese Controller, beschrieben in Kapitel 5.2.1.2, direkt von den Views angesprochen werden, wurde die Entscheidung getroffen `MDDConnectionModel` als Modell und nicht als Controller einzuordnen, obwohl es aufgrund seines hohen Logikanteils auch als Controller gezählt werden könnte.

5.2.1.2 Controller

Wie bereits in Kapitel 5.2.1.1 angesprochen, sind zwei wichtige Controller des MDD Tools der `MDDJobHandleController` und der `MDDMeasurementController`. Beide erben von `MDDConnectionModel`. Der erste Controller ist vor allem für die CRUD-Befehle auf die hinterlegten Jobs gedacht. Ein gemeinsamer `MainController` verwaltet eine interne Liste von `MDDJobModels`. Der separate `MDDMeasurementController` kann diese Liste nutzen um bestimmte Jobs auf dem MDD zu aktivieren. Er beinhaltet die Implementierung eines extra Threads, der während einer Messung ISO TP-Frames empfängt und entsprechend `MDDTcpCanFrameModel` parst und ablegt.

Ähnlich wie bei vielen UI-Frameworks wird die UI von WPF-Anwendungen in einem Hauptthread ausgeführt. Gibt es eine Userinteraktion, die ein Event mit angehängtem Code auslöst, wird auch dieser im Hauptthread ausgeführt. Falls der Code eine lange Laufzeit hat, kann währenddessen nicht mit der UI interagiert werden. Wenn eine Messung aktiv ist empfängt das Tool zyklisch neue Frames über TCP. Damit das Empfangen den Hauptthread nicht bremst, wird dafür durch den `MDDMeasurementController` ein separater Thread angelegt. TCP-Verbindungen sind allgemein als eine Interaktion anzusehen, die nicht im Hauptthread ausgeführt werden sollte. Das heißt auch die CRUD-Operationen auf die Jobliste des MDD sollten entkoppelt werden. Hierzu gibt es diverse asynchrone Designansätze. Der modernste im .Net-Framework ist die Benutzung der `async` und `await` Schlüsselwörter. Asynchrone Methoden, die solche Schlüsselwörter nutzen, verwenden keine separaten Threads sondern sind entkoppelte Tasks. Tasks in der .Net-Welt sind Aufgaben mit eigenem Kontext, die von einem Thread nach Bedarf und bei verfügbarer Prozessorzeit jederzeit ausgeführt werden können. Eine asynchrone `Connect` Methode kann beispielsweise auf der eigentlichen Verbindungsoperation das `await` Schlüsselwort nutzen. Der dahinter arbeitende Systemaufruf ist so von der Methode entkoppelt und der ausführende Kontext des Thread wechselt, bis zur Beendigung des Systemaufrufs, zurück zur aufrufenden Methode. Ziehen sich also die asynchron implementierten Methoden von einem UI-Event, beispielsweise ein Knopfdruck, bis zu einem blockenden Systemaufruf, springt der Kontext bis zurück in die Message-Queue. Die Message-Queue bezeichnet die Softwarekomponente, die Teil von WPF ist, die UI rendert und auf Userinteraktion wartet. Asynchrone Methoden mit `await` Schlüsselwort sind also eine einfache Möglichkeit länger andauernde Aufrufe vom Hauptthread zu entkoppeln und so die Message-Queue nicht zu blockieren. Der Vorteil gegenüber älteren asynchronen Designvorgehen, wie Asynchrones Programmiermodell (APM) mit seinen klassischen `BeginMethode` und `EndMethode` Aufrufen, ist, dass Methoden mit `async` und `await` sich in der Anwendung nur minimal von synchronen Methoden unterscheiden. Wieder, entsprechend dem Paradigma, die neusten Designmuster und Technologien zu verwenden, werden alle CRUD-Methoden als asynchrone Methoden entsprechend dem Task-based Asynchronous Pattern (TAP) mit `async` und `await` umgesetzt.

5.2.1.3 Views

Der für den Nutzer vor allem sichtbare Teil der Applikation ist der `MainView` (siehe Abbildung 5.2).

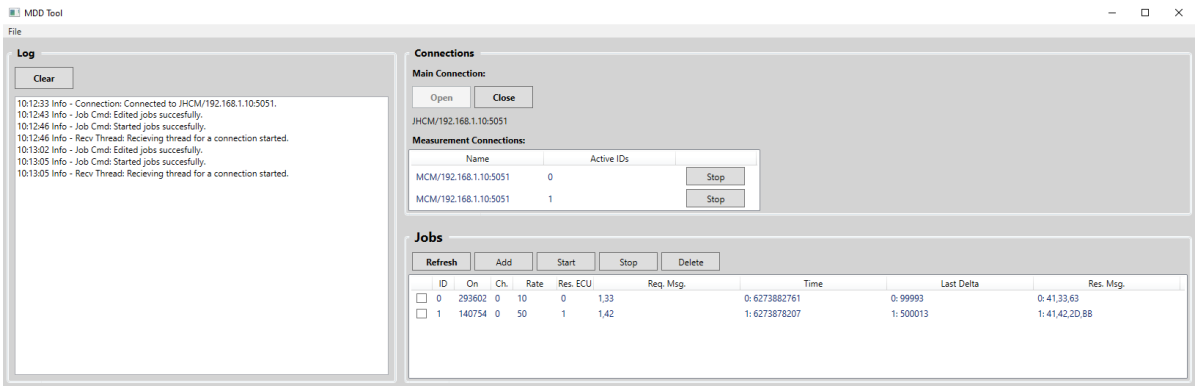


Abbildung 5.2: MDD Tool

Hier wird eine aktuelle Jobliste, ein Log, die momentanen Verbindungsdetails (IP-Adresse und TCP-Port) sowie die letzten empfangenen Werte einer Messung angezeigt. Die visuelle Jobliste erlaubt alle vier CRUD-Operationen und synchronisiert sich mit dem MDD. Ein extra View, `JobEditDialogView`, öffnet sich in einem separaten Fenster, falls ein Job aus der Liste editiert oder ein Neuer angelegt werden soll. Es können ein oder mehr Jobs aus der Liste gleichzeitig angewählt werden, um eine Messung zu starten oder zu stoppen. Da das Tool nur als Testtool fungieren soll, werden die eigentlichen Messwerte nur als Moving-Average dargestellt. Dazu werden je Job die letzten fünf Werte gespeichert und dann verworfen. Neben dem gemittelten Messwert wird die Differenz zwischen den Zeitstempeln der letzten beiden Werte in Mikrosekunden dargestellt. Während des Testens sind so Verbindungsabbrüche bzw. verlorengegangene Messwerte und Timer-Überläufe aufspürbar.

5.2.2 Add-in

Die Funktionalität des MDD Daemon wurde durch das MDD Tool sichergestellt. Der letzte große Schritt dieser Arbeit ist nun das Add-in für MPAS. Die folgenden Abschnitte beschreiben das Vorgehen für dessen Entwicklung und wie die bereits realisierten Klassen des MDD Tools weiter genutzt werden können.

5.2.2.1 Anpassung der MDD Tool-Klassen

Der vorerst einfachste Schritt der Transition zum Add-in war die Integration der Modell- und Controller-Klassen des MDD Tools. Deren Funktionalitäten für die Steuerung des MDD können fast unverändert so auch für das Add-in genutzt werden. MPAS nutzt

eine ältere .Net-Version als das MDD Tool. Dies war der Hauptgrund für einige kleinere Adaptionen. Hierzu gehören Methodenaufrufe in bestimmten überladenen Varianten, die in der älteren .Net-Version noch nicht verfügbar waren. Obwohl die ältere .Net-Version von MPAS zwar schon die relativ aktuellen `async` und `await` Schlüsselwörter bereitstellt, waren diesbezüglich auch Anpassungen zu tätigen. Das TAP-Modell sieht vor, dass alle Methoden im Stack, die eine asynchrone Methode aufrufen, selbst als asynchron deklariert sind. MPAS setzt derzeit auf kein spezifisches Modell für asynchrone Aufrufe, weshalb diese Anforderung nicht gegeben ist. Zeitkritische Aufrufe und Events, die durch die UI ausgelöst werden, werden bereits durch das MPAS-Hauptprogramm in separaten Threads durchgeführt. Dementsprechend sind asynchrone Methodenaufrufe nicht nötig. Dank der Möglichkeit asynchrone Methoden explizit synchron auszuführen, ist allerdings auch diese Anpassung von nicht allzu großem Aufwand. Ein Vorteil ergibt sich zudem daraus, dass in Zukunft bereits asynchrone Versionen der Methoden des MDD Add-in zur Verfügung stehen. Falls die MPAS-Codebasis, entsprechend des neuen TAP-Modells, überarbeitet werden sollte, stehen sie schon bereit.

5.2.2.2 MPAS Device-Aufbau

Ein MPAS Add-in beinhaltet mindestens ein MPAS Device, den Gerätetreiber, und wird in eine separate DLL kompiliert. Damit ein Device als solches genutzt werden kann, muss es gegen ein bestimmtes Interface und eine bestimmte Basisklasse implementiert werden. Danach liegt dem Device ein Zustandsautomat zugrunde. Dieser beschreibt den Status des Gerätes, der sich durch verschiedene Ereignisse und Benutzerinteraktionen ändern kann. Abbildung 5.3 zeigt die schematische Darstellung dieser Zustände und einige der wichtigsten Übergänge.

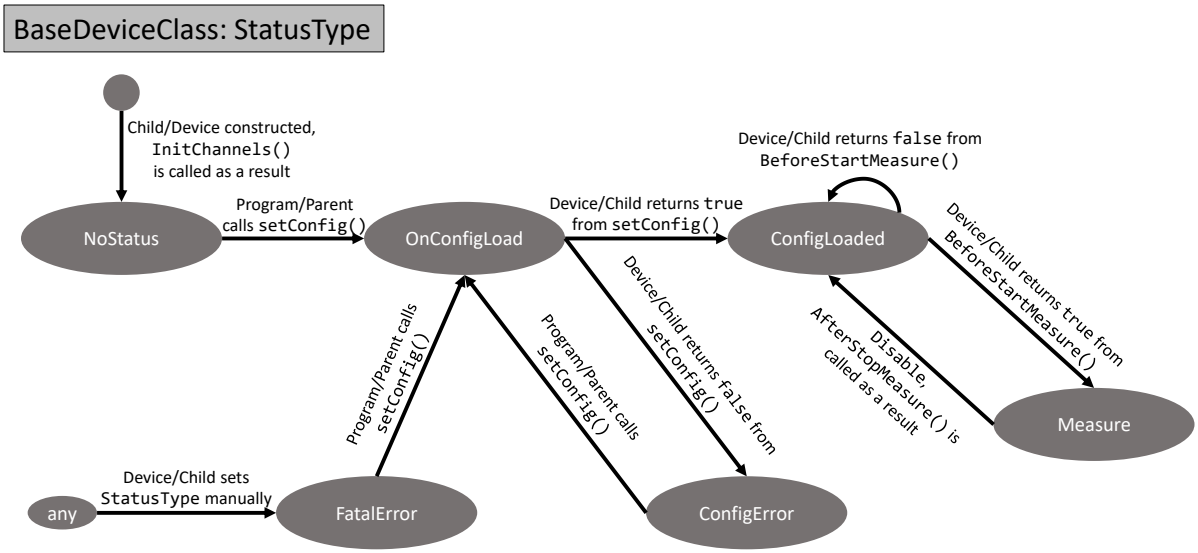


Abbildung 5.3: Zustände des Gerätetreiber im MPAS Add-in

Prinzipiell wird ein Gerätetreiber initialisiert, sobald MPAS startet und der Gerätetreiber zu einem Gerät gehört, das im Gerätemanager angelegt ist. Auch nach dem Start von MPAS können noch Geräte hinzugefügt oder gelöscht werden. Der Konstruktor und Dekonstruktor der Geräteklasse sind hierbei die Mechanismen, über den eine Geräteimplementierung weiß, ob sie genutzt wird oder nicht. Im Gerätemanager können angelegte Geräte dann aktiviert und deaktiviert werden. Geschieht Letzteres, wird der Gerätetreiber über eine Deinitialisierungsmethode gerufen, die praktisch dieselben Maßnahmen ergreifen sollte, wie der Dekonstruktor. Wird ein angelegtes Gerät aktiviert, erfolgt sofort der Versuch in den `ConfigLoaded`-Zustand überzugehen. Hierzu wird das Gerät erst per `setConfig` gerufen. Nun kann das Gerät diverse Maßnahmen vornehmen, um sich einsatzbereit zu machen, z.B. Verbindungen aufbauen. Während die Methode ausgeführt wird, ist das Gerät im Zustand `OnConfigLoad`. Dieser spezielle Zustand hilft MPAS zu überblicken, welche Geräte noch Zeit brauchen, bis sie bereit sind. Sobald das Gerät aus `setConfig` zurückkehrt und dabei Wahr oder Falsch zurück gibt, wechselt es in den messbereiten Zustand `ConfigLoaded` oder in einen Fehlerzustand `ConfigError`. Falls wegen eines Fehlers der Fehlerzustand erreicht wird, sollte dem Nutzer über eine Meldung mitgeteilt werden, wieso die Konfiguration fehlerhaft war. Dieser kann dann, nachdem er den Fehler behoben hat, mit einer Schaltfläche auslösen, dass erneut `setConfig` gerufen wird. Ein letzter wichtiger Zustand ist `Measure`. Er indiziert, dass das Gerät gerade Messwerte generiert und in entsprechenden Messkanälen ablegt. Die entsprechenden Methoden `BeforeStartMeasure` und `AfterStopmeasure` teilen dem Gerät den Übergang mit. Ähnlich wie bei `setConfig` hat die Implementierung so die Möglichkeit, die entsprechenden Vorkehrungen zu treffen.

Eine relevante Besonderheit tritt bei `BeforeStartMeasure` auf. MPAS übergibt bei Aufruf dieser Methode einen Zeitstempel an die Geräteimplementierung. Der Zeitstempel gibt an, wann genau die Messung gestartet werden soll. Damit ist sichergestellt, dass alle aktiven Geräte möglichst synchron ihre Messung starten. Die Implementierung des Add-in muss deshalb sicherstellen, dass sie bei diesem Aufruf den Messvorgang auf dem MDD mit der gewünschten Verzögerung startet.

5.2.2.3 Erweiternde Klassen

Natürlich sind die Klassen, die aus dem MDD Tool übernommen wurden, nicht ausreichend, um das Add-in gänzlich umzusetzen. Zum Verständnis der Add-in-Architektur wird hier auf einige weitere wichtige Klassen eingegangen, die entsprechend des MVC-Musters eingeordnet sind.

Modelle Der MDD Daemon kennt nur Jobs. Dabei ist ein Job ein Polling-Vorgang, der alle x Sekunden ein beliebiges ISO TP-Frame als Anfrage versendet und dann auf eine Antwort wartet. Das Protokollwissen zu OBD und UDS ist im MDD Daemon nicht hinterlegt. Ein MDD-Job ist entsprechend der Klasse aus dem MDD Tool bereits umgesetzt. Um die Umsetzung auf Seiten des Add-ins protokollspezifischer zu gestalten,

werden die Modelle `BaseRequestModel`, `OBDRequestModel` und `UDSRequestModel` eingeführt. `BaseRequestModel` erbt von `MDDJobModel` und somit all dessen Funktionalitäten. `OBDRequestModel` und `UDSRequestModel` erben von `BaseRequestModel`. Auf ihrer Ebene sind generische Werte, wie die Payload des Frames als ganzes Byte-Array, nicht mehr sichtbar. Stattdessen sind nur noch die einzelnen Bytes sichtbar, die, je nachdem ob OBD oder UDS, SID, PID, LEV oder DID beschreiben. Zusätzlich ist in einem `BaseRequestModel` nicht nur der Job hinterlegt, sondern auch wie die Antworten auf die Jobs vom MDD, `MDDTcpCanFrameModel`, einzuordnen sind. MPAS fordert, dass Messwerte in sogenannte Channels/Kanäle geschrieben werden. Das Modell für diese Channels ist vorgegeben, kann aber in eigenen Implementierungen durch Vererbung angepasst werden. Hierzu gibt es das neue Modell `BaseValueModel`, das von der MPAS Channel-Klasse erbt. Von `BaseValueModel` gibt es dann weitere, noch spezifischere Ableitungen. Ein `BaseRequestModel` erbt von `MDDJobModel` und beschreibt somit immer nur einen Job. Bestimmte OBD- oder UDS-Anfragen können aber mehrere Messwerte in einer Antwort beinhalten. Deswegen besitzt ein `BaseRequestModel` eine Liste von `BaseValueModels`. Jedes Kind von `BaseValueModel` muss eine Methode implementieren, die bestimmte Bytes der Antwort in einen Messwert umrechnen kann. Jedes Kind von `BaseRequestModel`, also `OBDRequestModel` und `UDSRequestModel`, muss eine Methode implementieren, ein `MDDTcpCanFrameModel` entgegen zu nehmen und es entsprechend seiner Liste von `BaseValueModels` aufzulösen und in die entsprechend darunter liegenden MPAS Kanäle abzulegen.

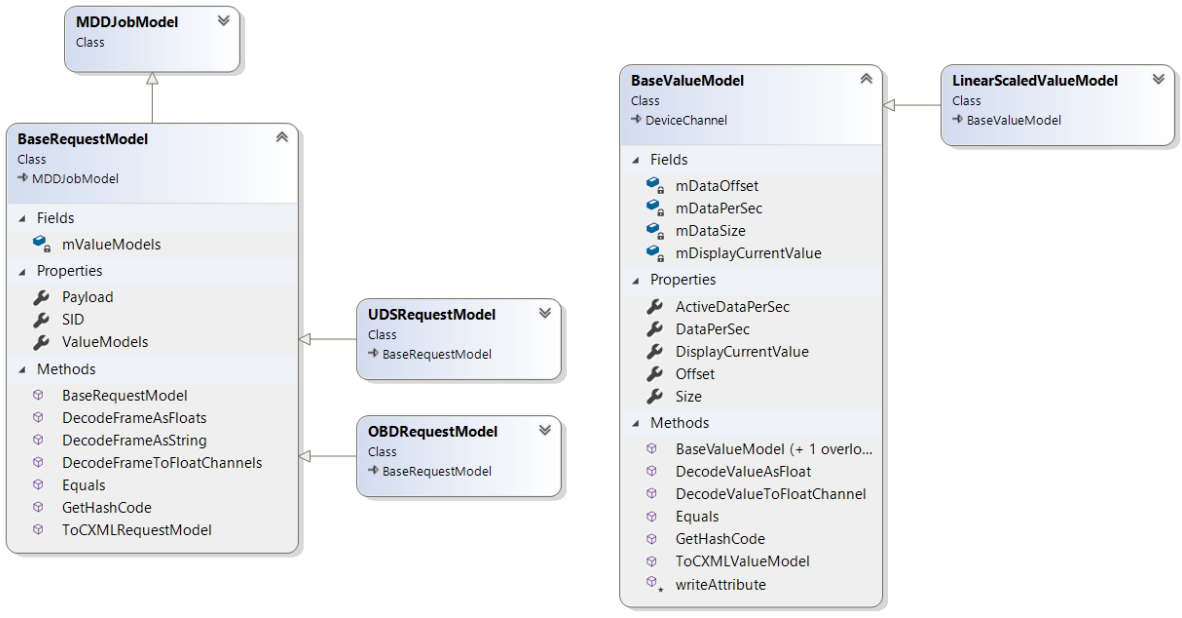


Abbildung 5.4: Grundlegende Datenmodelle im MPAS Add-in

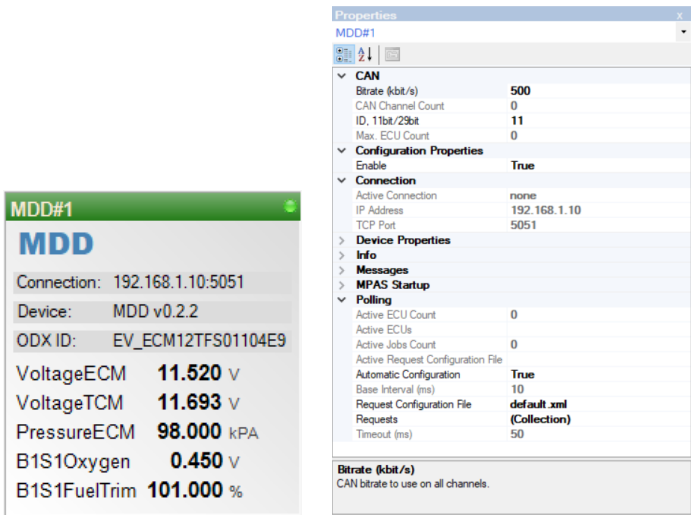
Abbildung 5.4 soll diese Abhängigkeiten noch einmal in einem vereinfachten Klassendiagramm darstellen. Ein neuer Request wird angelegt. Hierzu werden die `OBDRequestModel` oder `UDSRequestModel` Ableitungen genutzt. Dort wird für OBD die SID und PID, für UDS die SID und LEV/DID sowie u.a. die Polling-Frequenz hinterlegt. Parameter wie die `Payload`, bzw. die Nachricht des Requests, werden in abgeleiteten Klassen ausgeblendet, insofern diese spezifischere Implementationen davon zur Verfügung stellen. Die Requests beinhalten zudem die Information darüber, wie die empfangene ISO TP-Frames zu interpretieren sind. Dazu werden `BaseValueModels` genutzt. Ein mögliches `BaseValueModel` ist das `LinearScaledValueModel`. Dieses beschreibt, wie x Bytes entsprechend eines Polynoms vom Grad x in eine Gleitkommazahl umgerechnet werden, wozu es x Faktoren und einen Offset speichert.

Ein angelegtes `BaseRequestModel` hat ein darunterliegendes `MDDJobModel`. Es kann also den Controllern, wie sie schon aus dem MDD Tool bekannt sind, übergeben werden. Diese legen die Anfrage als Job auf dem MDD an und empfangen bei aktiver Messung Frames. Empfangene Frames, als Instanz von `MDDTcpCanFrameModel`, können dann der Methode `DecodeFrameToFloatChannels` übergeben werden, die jede Variante des `BaseRequestModels` besitzen muss. Diese Methode nutzt die hinterlegte `mValueModels`-Sammlung, um alle Bytes des Antwort-Frames den richtigen `BaseValueModels` zuzuordnen. Diese ihrerseits interpretieren die Bytes entsprechend der Methode `DecodeValueToFloatChannel` und fügen sie dann dem darunterliegenden `DeviceChannel` hinzu.

Der Aufbau aus Anfragen, die Byteinterpretationen für ihre Antworten referenzieren, spiegelt grob die Darstellung solcher Anfragen in ODX-Dateien wieder. Diese Ähnlichkeit soll später helfen, dass `BaseRequestModels` aus ODX-Dateien konfiguriert werden.

Controller Bereits aus dem MDD Tool sind die Controller `MDDJobHandleController` und `MDDMeasurementController` bekannt. Sie können, wie in Kapitel 5.2.2.3 angedeutet, Requests nutzen und sie auf dem MDD als Job über Kommandos anlegen sowie Antwort-Frames empfangen. Das Add-in erweitert lediglich die `MDD` und die `PollingController` Klasse. `MDD` ist die Ausgangsklasse für das Add-in und realisiert ein MPAS Device. Im MVC-Konzept kann sie als Controller betrachtet werden. Sie implementiert all die Mechaniken und Methoden die im Kapitel 5.2.2.2 zum MPAS Device beschrieben sind. Die `PollingController` Klasse verwaltet alle aktiven Requests. Viele in ihr umgesetzten Logiken könnten genauso gut direkt Teil der `MDD` Klasse sein. Aus Gründen der allgemeinen Übersichtlichkeit und um ggf. später Klassen zu erweitern, die das MDD für automatische zyklische UDS-Nachrichten oder manuelles Polling nutzen, wurde sie abgekoppelt.

Views Das MDD Add-in realisiert zwei Views. Der erste View, `MDDView`, beschreibt das kleine Fenster, welches im Gerätemanager von MPAS angezeigt wird. Es stellt einige wichtige Informationen und aktuelle Messwerte dar, vergleiche Abbildung 5.5(a).



(a) Übersichtsfenster (b) Einstellungen

Abbildung 5.5: MDD-Ansichten im Gerätemanager

Wird im Gerätemanager ein Gerät angewählt, so öffnet sich eine zusätzliche Ansicht um diverse Einstellungen/Konfigurationen für das Gerät vorzunehmen, vergleiche Abbildung 5.5(b). Alle konfigurierbaren Werte eines Gerätes können in der entsprechenden Klasse speziell deklariert werden, damit sie dann in dieser Ansicht verfügbar sind. Für das MDD Add-in sind das beispielsweise das Tupel aus IP-Adresse und Port für die Verbindung, zu nutzende CAN-Bitrate etc.

Um manuell konfigurieren zu können, welche OBD- und UDS-Requests durch das MDD gepollt werden sollen, ist ein zweiter View, `RequestEditView`, umgesetzt.

Abbildung 5.6 zeigt dieses Fenster. Der Editor erlaubt alle CRUD-Operationen auf der Liste von Requests und deren Werte. In tabellarischer Ansicht können die verschiedenen Werte und Requests angewählt werden. Seitlich angeordnete Eigenschaftselemente erlauben es dann die verschiedenen Einstellungen an den Anfragen und Werten vorzunehmen. Dazu gehören z.B. die MDD spezifischen Einstellungen, die OBD SID und PID, die gewünschte Frequenz des Polling-Vorgangs und wie Antworten in Werte aufgeteilt werden sollen. Da jedem Wert, wie in Kapitel 5.2.2.3 beschrieben, ein MPAS Channel zugrunde liegt, werden in den Werten auch Einstellungen vorgenommen, die diesen Channel betreffen. Das kann u.a. die Farbe für Messschriebe oder das Format für die Darstellung einzelner Werte sein. In der momentanen Implementierung des MDD Add-in wird zwischen UDS- und OBD-Anfragen unterschieden. In Zukunft steht es offen hier noch weitere Varianten zu ergänzen. Ähnlich verhält es sich mit den verschiedenen Varianten der Werte. Durch die Wahl von Buttons mit mehreren Auswahlmöglichkeiten wird die UI der Anzahl der Optionen gerecht. Der Button für Request-Tests erlaubt es einen einmaligen Polling-Vorgang auszulösen, um zu verifizieren, dass der parametrisierte Request die

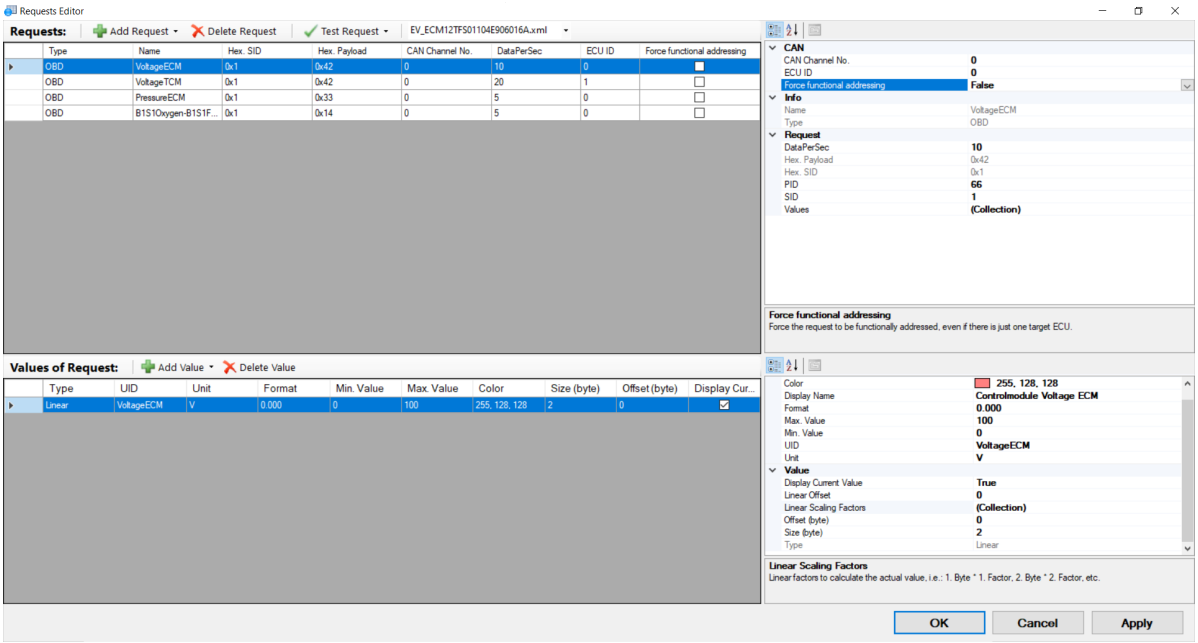


Abbildung 5.6: MDD-Fenster um Requests anzulegen und zu editieren

richtigen Werte liefert. Die hier angelegte Konfiguration wird in separaten XML-Dateien gespeichert. Das Vorgehen diesbezüglich steht im Zusammenhang mit den ODX-Dateien und wird deshalb im folgenden Kapitel 5.2.3 genauer beleuchtet.

5.2.3 ODX-Parser

Wie bereits in den Grundlagen in Kapitel 2.4 angesprochen, kann eine ODX-Datei bzw. ein Archiv solcher Dateien (PDX) helfen herauszufinden, welche OBD- und UDS-Anfragen ein Fahrzeug und dessen Steuergeräte benutzt. Über eine bestimmte UDS-Anfrage, die laut Standard immer vom Fahrzeug unterstützt werden muss, kann eine ODX-ID erfragt werden. Diese ID entspricht dem Dateinamen einer ODX-Datei, die das antwortende Steuergerät beschreibt.

Das ODX-Format besteht aus verschiedenen XML-Elementen. Einige der hier relevanten sind beispielsweise `REQUEST`, `DATA-OBJECT-PROP` oder `POS-RESPONSE`. Diese Elemente können auf andere verweisen. Hierzu gibt es Referenzen und Vererbungskonzepte wie im Grundlagenkapitel 2.4 besprochen. Um den Aufbau näher zu evaluieren wurde deshalb zuerst ein separater *ODX Service Explorer* entwickelt. Grundlage für das Tool sind Klassen, die dem Aufbau der entsprechenden ODX-Elemente entsprechen. Für jedes Element gibt es eine Klasse. Jede Klasse definiert wie ihr jeweiliges ODX-Element genannt wird und wie die entsprechende Abkürzung (`SHORT-NAME`) für das Element in ODX-Dateien heißt. Weiter hält sie Standarddatentypen wie Strings, Gleitkomma- und Ganzzahlen

sowie ggf. Verweise auf andere Elemente. Die Verweise sind eine geparste Klasseninstanz oder eine noch nicht aufgelöste Referenz auf eine ID.

Listing 5.1: Beispielhafter Ausschnitt aus einer ODX-Datei

```
<?xml version="1.0" encoding="UTF-8"?>
<ODX>
    <DIAG-LAYER-CONTAINER ID="DLC_EV_ECM14TFS01104E906014K_001">
        <!-- ... -->
        <ECU-VARIANTS>
            <ECU-VARIANT ID="EV_ECM14TFS01104E906014K_001">
                <DIAG-COMMS>
                    <DIAG-SERVICE ID="DiagnServi_RequeCurrePowerDiagnData">
                        <REQUEST-REF ID-REF="Req_RequeCurrePowerDiagnData" />
                        <POS-RESPONSE-REFS>
                            <POS-RESPONSE-REF ID-REF="Resp_RequeCurrePowerDiagnData" />
                        </POS-RESPONSE-REFS>
                        <!-- ... -->
                    </DIAG-SERVICE>
                    <!-- ... -->
                </DIAG-COMMS>
                <REQUESTS>
                    <REQUEST ID="Req_RequeCurrePowerDiagnData">
                        <PARAMS>
                            <PARAM SEMANTIC="SERVICE-ID" xsi:type="CODED-CONST">
                                <!-- ... -->
                            </PARAM>
                            <PARAM SEMANTIC="ID" xsi:type="VALUE">
                                <!-- ... -->
                                <DOP-REF ID-REF="DOP_TEXTTABLEParamIDs"></DOP-REF>
                            </PARAM>
                        </PARAMS>
                        <!-- ... -->
                    </REQUEST>
                </REQUESTS>
            </ECU-VARIANT>
        </ECU-VARIANTS>
    </DIAG-LAYER-CONTAINER>
</ODX>
```

Das Listing 5.1 zeigt einen beispielhaften und vereinfachten Ausschnitt aus einer ODX-Datei für das Steuergerät mit der ODX-ID `ECM14TFS01104E906014K`. Gut zu sehen ist, dass der Aufbau eher flach ist. Das heißt einem `DIAG-SERVICE` sind die zugehörigen `REQUEST` und `POS-RESPONSE` nicht direkt unterstellt, sondern werden stattdessen als separate Elemente referenziert. Damit können sich mehrere Services auf denselben Request beziehen, ohne dass es Duplikate in der Beschreibung geben muss. Um allerdings eine vollständigen Request aus ODX-Dateien zu bestimmen, wie er in das `BaseRequest-Model` des MDD Add-In passt, müssen alle diese Elemente zusammengefasst werden. Dazu werden besagte Klasseninstanzen gefüllt, womit sich Listen an Services, Requests, Pos. Antworten etc. ergeben. Danach können dann bestehende Referenzen aufgelöst und die referenzierten Elemente als ihre geparste Klasseninstanz richtig zugeordnet werden. In Abbildung 5.7 ist dies nochmal grafisch dargestellt.

Die Kette an ODX-Elementen hört nicht bei der Request-Definition auf. Darauf folgen Parameterdefinitionen und Einträge aus Datentabellen oder sogenannte `CODED-CONST`

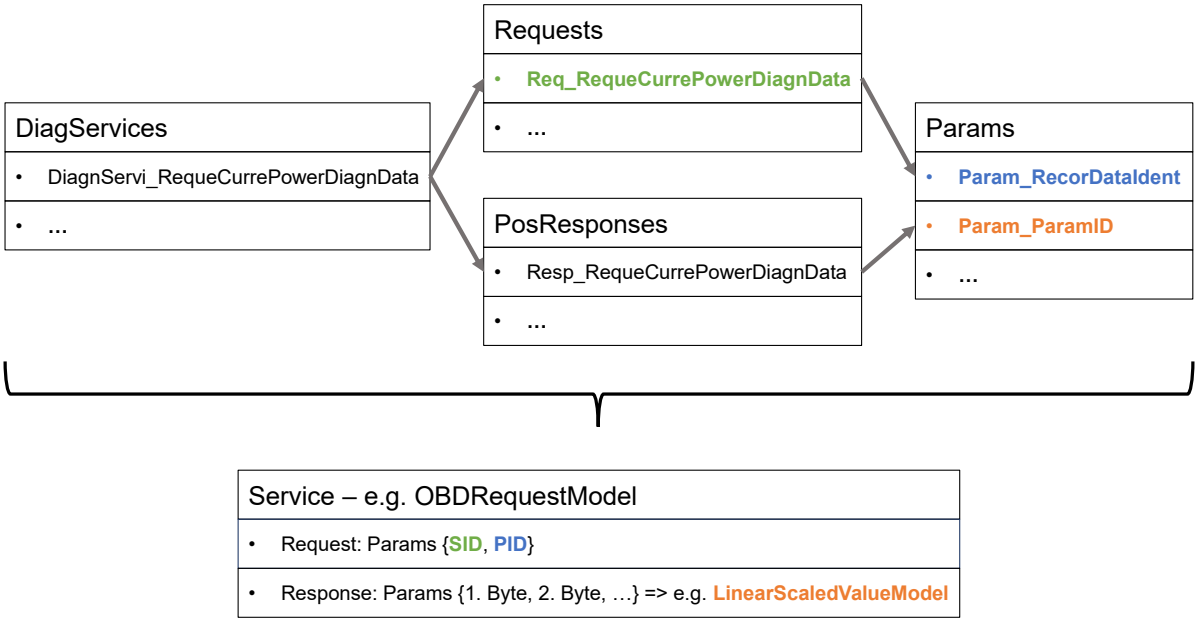


Abbildung 5.7: Beispielhaftes Interpretieren der ODX-Struktur mit bereits aufgelösten Referenzen

Werte. Für das prinzipielle Verständnis, woher die später relevanten Werte wie die OBD PID und SID kommen, sollte diese Darstellung allerdings ausreichend sein.

Das entwickelte Tool *ODX Service Explorer* kann die Services wie hier beschrieben auflösen, siehe Abbildung 5.8. Allerdings wird dabei immer nur direkt die ODX-Datei des Steuergerätes gelesen. Wie in den Grundlagen angesprochen, kennt ODX aber das Prinzip der Vererbung. Die direkte Definition für ein Steuergerät, `ECU-VARIANT/EV`, steht im Eltern-Kind-Verhältnis zu einer `BASE-VARIANT/BV`, die z.B. für alle ECMs genutzt wird. Diese wiederum untersteht einer `FUNCTIONAL-GROUP/FG`, die z.B. alle Steuergeräte, die im Zusammenhang mit dem Antriebsstrang stehen, zusammenfasst. Zuletzt untersteht die `FUNCTIONAL-GROUP` dann einem `PROTOCOL/PR`, das alle Steuergeräte vereint, die z.B. OBD sprechen. Durch all diese Ebenen können sich Elemente wie Services, Requests, Parameter etc. vererben, enterben und überschreiben. Hinzu kommen noch losgelöste Datenbibliotheken, die eingebunden werden können. Die Komplexität erlaubt den Herstellern enorme Flexibilität, macht die Implementation eines schnellen und zuverlässigen Parsers aber aufwendig.

Abseits der Komplexität durch die Vererbung, haben sich weitere Herausforderungen ergeben (teilweise direkt in der Norm nachvollziehbar [ASA08, SZ11a]):

- Von den zwei im Testaufbau verbauten Steuergeräten, ein ECM und ein TCM, antwortet nur das ECM auf die Anfrage nach seiner ODX-ID.
- Die speziellen Bezeichnungen für Referenztypen variiert von Hersteller zu Hersteller, z.B. einerseits `ID-REF`, andererseits `ODX-LINK`.

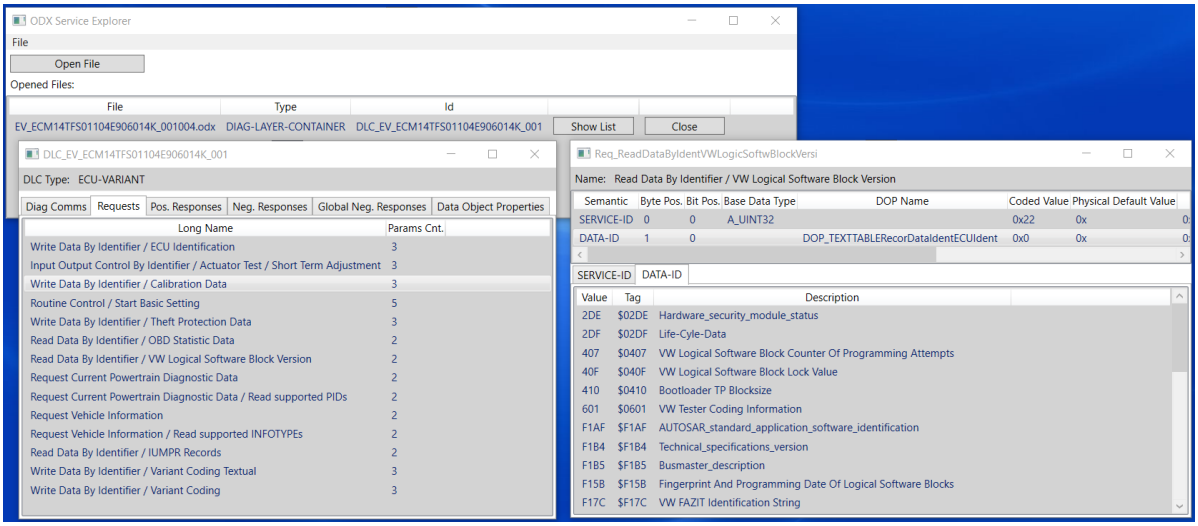


Abbildung 5.8: Ausschnitt einer ECU-Beschreibung im *ODX Service Explorer*

- Die Dateiendungen sind nicht auf zwangsläufig immer `*.odx` genormt. Einige Hersteller nutzen z.B. `*.odx-d` speziell für Dateien mit Diagnoseinformationen oder `*.odx-fd` für funktionelle Bibliotheken.
- Die Grundidee, dass das Parsen hilft, wenn bei einem Fahrzeug die aktuelle Versorgungsspannung z.B. über PID `0x42` und beim anderen `0x04` abzufragen ist, kann problematisch werden. Dieses Vorgehen funktioniert nämlich nur, wenn in den entsprechenden ID-Tabellen dieser Messwert immer als *Control Module Voltage* oder dergleichen bezeichnet wird. Falls diese Bezeichnung nicht eindeutig ist, hat die Automatisierung keine Möglichkeit zu wissen, dass *Control Module Voltage* in Fahrzeug A dasselbe ist, wie *Board Voltage* in Fahrzeug B. Es bräuchte bei jedem neuen Fahrzeug mit neuem ODX-Datensatz eine Person, die aus den geparsten möglichen Werten die Interessanten auswählt und mit einem eindeutigen Namen homologiert.
- Im verfügbaren ODX-Datensatz zu den Steuergeräten im Testaufbau gibt das ECM an bestimmte Anfragen zu unterstützen, reagiert aber auf diese in der Praxis nicht.

Die Summe des Aufwands in der Umsetzung des Parsers und das Untersuchen dieser Schwierigkeiten und Unstimmigkeiten hat dazu geführt, dass beschlossen wurde den Ansatz in dieser Arbeit nicht weiter zu verfolgen. Stattdessen werden Schnittstellen und ein eigenes Dateiformat für Request-Konfigurationen angedacht, die in Zukunft durch einen ODX-Parser gefüllt werden könnten, im Zuge dieser Arbeit aber nur manuell über den `RequestEditView` editiert werden. Indem dieses eigene Dateiformat eine ODX-ID beinhalten, können mehrere solche Dateien automatisch bei Teststart durchsucht und diejenige ausgewählt werden, die zum gerade getesteten Fahrzeug passt.

Das neue Dateiformat ist wie ODX, und viele andere Konfigurationen in MPAS, XML-basierend. Die Standardbibliotheken von .Net erlauben es, speziell deklarierte Klassen automatisch in XML-Format zu serialisieren. Aus diesem Grund werden weitere Modell-Klassen angelegt. Nach demselben Schema wie schon bei den Add-in-Modellen (Kapitel 5.2.2.3) beschrieben, gibt es neue serialisierbare Klassen für Requests und Values, wobei es hiervon die Ableitungen für OBD- und UDS-Requests sowie linear skalierte Werte gibt. Jede dieser Klassen kann in und aus ihrem Äquivalent aus den Add-in-Modellen übersetzt bzw. generiert werden. Die Einstellungen, die das Add-in offenlegt, erlauben es, sowohl beliebige Konfigurationen einzulesen, als auch automatisch entsprechend der erfragten ODX-ID auszuwählen.

5.2.4 Besonderheiten

Wie schon bei der Realisierung des MDD Daemon wurden diverse Besonderheiten und Fehlerquellen während der Implementation aufgedeckt. Im Zuge der Konzeption wurden bereits Mechaniken wie Timeout-Frames und TCP als verbindungsorientiertes Protokoll bedacht, um Verbindungsabbrüchen entgegen zu wirken. Erst während einiger Tests mit der umgesetzten Toolkette, Validation im V-Modell, hat sich gezeigt, wie spezielle Verbindungsabbrüche unerwartete Fehler hervorrufen können. Die folgenden Kapitel vereinen die Beobachtungen aus der Validation, daraus abgeleitete Konzeptionen für die Anpassungen an bereits realisierter Mechaniken und deren Umsetzung. Damit ergeben sich erweiternde Iterationen über eigentlich einmalige Schritte im V-Modell.

5.2.4.1 Halboffene Verbindungen

Ein klassisches Problem mit TCP/IP sind halboffene Verbindungen. Wenn ein Teilnehmer die Verbindung nicht explizit schließt, sondern die Verbindung aufgrund von beispielsweise getrennter physikalischer Verbindung abbricht, kann es passieren, dass die Teilnehmer dies nicht sofort erkennen. Wenn der entsprechende Code gerade auf einer lesenden Operation blockiert kann das problematisch werden, da lesende Operationen rein passiv sind. Ist die Verbindung abgebrochen und keine Daten werden mehr übertragen, bleibt das Programm auf unbestimmte Zeit an der Operation blockiert. Schreibende Operationen hingegen enden in detektierbaren aktiven Fehlern, wenn die Verbindung unerwartet abgebrochen ist. Abhilfe schaffen lesende Operationen mit Timeout und/oder Verbindungen, die nur geöffnet bleiben, wenn sie aktiv benutzt werden.

Im Falle des MDD Add-in werden beispielsweise zwei Verbindungen zum MDD geöffnet. Die eine Verbindung legt neue Jobs an, löscht alte bzw. editiert sie auf dem MDD. Während einer Messung, wenn dauerhaft Frames empfangen werden, wird dafür eine separate zweite Verbindung geöffnet. Die erste Verbindung zum Verwalten der Jobs wird aber nicht geschlossen, sondern verweilt. Auf Seiten des MDD Daemon bedeutet das ein Verweilen auf einer lesenden Operation die neue Kommandos erwartet. Das kann bei Verbindungsabbrüchen problematisch werden. Deshalb wird die Verbindung, die die

Jobs verwaltet, so angepasst, dass sie immer nur kurz geöffnet und dann sofort wieder geschlossen wird. Die Verbindung für die Messung hat dieses Problem nicht, da sie entsprechend der angelegten Jobs und hinterlegten Requests weiß, wie regelmäßig sie Werte zu erwarten hat, und deshalb einen Timeout etablieren kann, der auslöst falls die Messwerte ausbleiben. Auf Seiten des MDD Daemon ist die Verbindung für die Messung eine, auf die der Daemon regelmäßig schreibt und damit Verbindungsabbrüche sofort erkennt.

5.2.4.2 Timeouts

Das zweite Problem hat einen ähnlichen Kontext wie das Vorangegangene, tritt aber bei Verbindungsabbrüchen des CAN-Bus auf. Falls der Bus einen Fehler aufweist, kommt der bereits angesprochene $P2$-Timeout ins Spiel. Dieser beträgt meistens 50 ms. Das MDD erlaubt das Polling von Werten mit einer beliebig hohen Frequenz entsprechend eines Grundintervalls. Die ECUs des Testaufbaus erlauben sinnvollerweise (siehe Kapitel 4) Frequenzen bis 100 Hz. Das Grundintervall ist deshalb standardmäßig auf 10 ms parametrisiert, das heißt das MDD erlaubt nur Polling-Raten mit Intervallen, die ein Vielfaches des Grundintervalls sind. Der Worst-Case in diesem Beispiel ist also ein Job, der auf 100 Hz mit $1 \cdot Grundintervall$ konfiguriert ist. Falls nun ein $P2$-Timeout nach 50 ms für die ECU mit dem 100 Hz-Job auftritt, so hat sich nach dem Timeout deren Queue mit fünf offenen Requests gefüllt. Da die Leitung unterbrochen bleibt, tritt der nächste Timeout auf. Die Queue wurde zwar um einen Request geleert, es kamen aber wieder fünf dazu, sie hält nun neun. Dieser Zyklus wiederholt sich, bis die Queue an ihre maximale Kapazität stößt. Ab diesem Zeitpunkt gehen Requests verloren. Auf Seiten des MPAS Add-in ist es wichtig, dass die Messkanäle, die keine Zeitstempel haben, stets mit der richtigen Anzahl von Werten entsprechend ihrer Abtastfrequenz gefüllt werden. Der MDD Daemon verschickt bereits erläuterte spezielle Timeout-Frames, woraufhin das Add-in NaN-Werte (Not a Number) in den Channel ablegt. Die Anzahl dieser Werte geht aber nicht auf, sobald Requests auf dem MDD wegen der vollen Queue verloren gehen, bzw. die ganzen aufgestauten Requests in einer Welle von viel zu schnell gesampelten Werten eintreffen, sobald der CAN-Bus wieder verbunden und fehlerfrei ist.

Um diese Problematik zu beheben wird bei einem Timeout nach $P2$ die Queue für den betroffenen ECU gänzlich geleert und für jeden angefallenen Eintrag ein Timeout-Frame versendet. Damit das Verwerfen dieser Requests die Synchronisation möglicher funktionaler Anfragen nicht unmöglich macht, werden diese Requests nach dem normalen Vorgehen synchronisiert und erst dann verworfen. Als zusätzliche Absicherung nutzt das MPAS Add-in die Zeitstempel, die jedem empfangenen Frame angehängt sind, um nachzuvollziehen, ob womöglich doch Werte abhanden gekommen sind. Falls dies der Fall ist, berechnet das Add-in selbst die fehlende Menge an Werten und trägt entsprechend viele NaNs ein.

In der Praxis zeigt sich, dass dieses Vorgehen selbst bei längeren Messzyklen, beispielsweise dem WLTC-Zyklus mit 1800 s, effektiv ist. Abbildung 5.9 zeigt einen Messschrieb in MPAS, während dem es einen Aussetzer auf dem Bus gab. Zu sehen ist, dass trotz

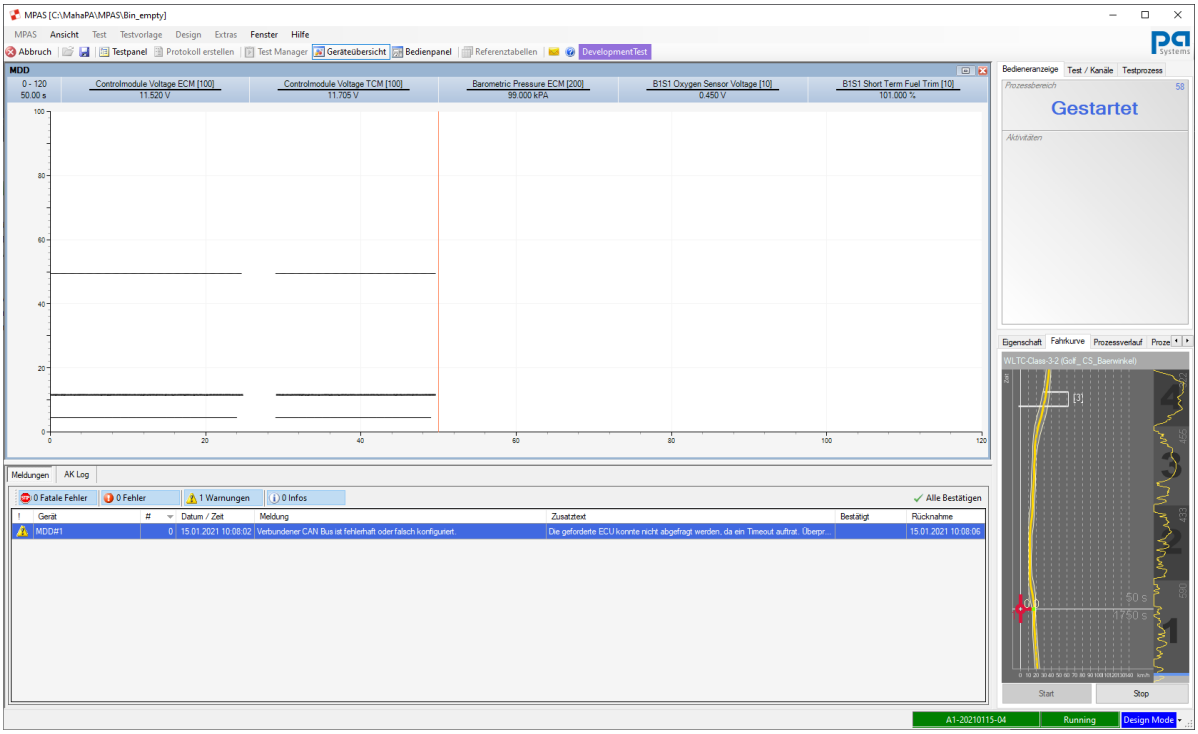


Abbildung 5.9: MPAS-Messschrieb mit CAN-Verbindungsabbruch

der Lücke mit NaNs, die Anzahl der eingetragenen Messwerte noch zum aktuellen Messungs-Cursor aufschließt.

Selbst mit vielen langen und kurzen zufälligen Verbindungsabbrüchen und Wacklern im CAN-Bus, weicht die Anzahl der Werte am Ende nach entsprechenden Tests lediglich um maximal ± 20 Werte ab. Ein 50 Hz Channel sollte $50\,\text{Hz} \cdot 1800\,\text{s} = 900000$ Werte generieren. Es ergibt sich eine Abweichung von $\frac{20}{90000} \approx 0{,}2\,‰$. Diese Abweichung ist genau genug für eine eigentlich sowieso ungültige Messung.

6 Ergebnis

Das abschließende Kapitel dieser Arbeit zieht ein rückblickendes und bewertendes Fazit in Kapitel 6.1 und betrachtet einige fortführende Gedanken im Abschnitt 6.2.

6.1 Zusammenfassung

Das MDD in Kombination mit dem MPAS Add-in erfüllt die Anforderungen an eine Toolkette zum Erfassen von Mess- und Diagnosewerten über OBD und UDS. Die umgesetzte Benutzeroberfläche des Add-in erlaubt es, das MDD in MPAS Testprozesse zu integrieren und festzulegen, welche Messwerte aufgezeichnet werden sollen. Die Parametrisierung der OBD- und UDS-Anfragen kann durch ein separat entwickeltes Softwaretool zum Lesen von ODX-Dateien teilweise unterstützt und dann in einem XML-Format gespeichert werden. Für verschiedene Testprozesse und Fahrzeuge können mehrere Konfigurationen angelegt werden. Im Test kann dann festgelegt werden, ob die Konfiguration automatisch gewählt werden soll oder manuell angegeben wird. Abbildung 6.1 zeigt die gesamte Architektur der Toolkette zur Messwertaufzeichnung.

Das MDD läuft auf der Basis eines Buildroot-Projektes, um ein angepasstes Linux-basiertes Betriebssystem zur Verfügung zu stellen. Der integrierte Linux-Kernel wurde u.a. dahingehend angepasst, dass ein neues Kernelmodul eingefügt wurde. Sowohl OBD als auch UDS nutzen als Transportprotokoll ISO TP (siehe Kapitel 4.3.1.1). Das PCAN-Ethernet Gateway, das dem MDD zu Grunde liegt, ist auf Linux Kernel der Version 2.6.31 ausgelegt. In dieser Kernelversion ist CAN in Form der SocketCAN-Schnittstelle verfügbar, nicht aber darauf aufbauende Transportprotokolle. Indem ein Kernel-Modul für diese Funktionalität genutzt wird, im Gegensatz zu einer Userspace-Implementierung, sind zeitliche und Multi-Nutzer Einschränkungen keine Problematik [Har20b]. Eine durchdachte Entwicklungsumgebung wurde erstellt, die sowohl für die native, als auch für cross-plattform Entwicklung geeignet ist. Die Umgebung beinhaltet zudem einen (remote) Debugger und ist dank generierter SDK portabel (siehe Kapitel 5.1.1).

Die Funktionalität zum Erfassen und Weiterleiten von ISO TP-Nachrichten obliegt dem entwickelten MDD Daemon. Ziel der Entwicklung der Software war es, möglichst viel Protokollwissen bezüglich OBD und UDS auszulagern und die Software dahingehend offen zu lassen. Deshalb ist der MDD Daemon als generische Software umgesetzt, die per TCP-Verbindung konfiguriert werden kann und dann Anfragen einmalig oder zyklisch auf den CAN-Bus schreibt. Werden auf dem CAN-Bus passende Antworten empfan-

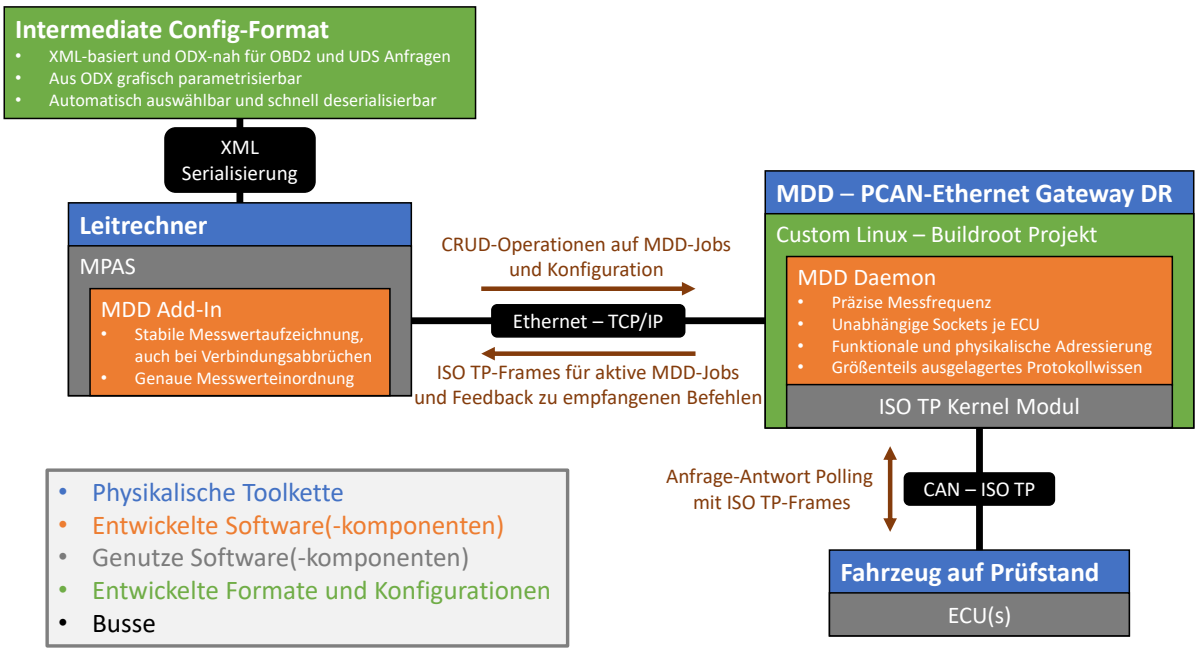


Abbildung 6.1: Projektumsetzung als MPAS Add-in und MDD

gen, so werden diese über die TCP-Verbindung an den verbundenen Clienten geschickt. Hierzu werden mehrere ISO TP-Sockets in separaten Threads genutzt, die auf protokollspezifische Adressen für mehrere ECUs konfiguriert werden. Prinzipiell arbeiten die Threads unabhängig voneinander, womit Anfrage-Antwort-Abläufe, Polling, in Frequenzen durchgeführt werden können, die praktisch nur durch die Geschwindigkeit der ECUs limitiert sind (siehe Kapitel 4.4.3). Da sowohl OBD als auch UDS gewisse Anfragen funktional, also an mehrere ECUs gleichzeitig, adressieren, ist ein Synchronisationsmechanismus für die Polling-Threads umgesetzt worden. Dank des Betriebssystems, das für den embedded Einsatz ressourcensparend optimiert wurde, können in Kombination mit FIFO-Scheduling hoch präzise Timer und Timer-Abfragen für das Polling benutzt werden (siehe Kapitel 4.4.1). Ein entwickelter Befehlssatz erlaubt es die Polling-Jobs und andere Parameter auf dem MDD zu konfigurieren. Der Befehlssatz erlaubt einmalige Anfragen, zyklische Polling-Jobs und das Konfigurieren diverser Parameter durch ASCII-codierte Zeichenfolgen. Jeder Befehl wird durch das MDD mit entsprechendem Feedback beantwortet. Es ergibt sich eine einfach zu benutzende und vielseitige Schnittstelle für OBD und UDS in Form des umgesetzten Measurement and Diagnostic Device, das nicht nur durch das realisierte MDD Add-in genutzt werden kann (siehe Kapitel 5.1.2.2).

Damit das MDD in der Prüfstandsautomatisierungssoftware MPAS genutzt werden kann, wurde ein Add-in für diese entwickelt. Neben der automatischen Auswahl aus diversen vorgefertigten Konfigurationen stellt eine Implementierung, die robust gegenüber Verbindungsabbrüchen auf TCP und CAN Seiten ist, sicher, dass ungeschultes Personal das MDD in MPAS Testabläufen nutzen kann (siehe Kapitel 5.2.4.1 und 5.2.4.2). Ein im

Add-in integrierter Editor für die Konfigurationen erlaubt es einfach neue OBD- oder UDS-Anfragen zu hinterlegen, die automatisch in die entsprechenden Jobs und Parametrisierungen für das MDD umgesetzt werden (siehe Kapitel 5.2.2.3). Der Inhalt der Konfigurationen ist darauf optimiert, entsprechend der Informationen aus ODX-Dateien angelegt werden zu können. Da sich das Format auf die wichtigsten Parameter für OBD- und UDS-Anfragen konzentriert, ist es deutlich schneller lesbar als vergleichbar umfangreiche und vielschichtige ODX-Dateien. Unterstützende Tools wurden im Zuge der Arbeit angedacht, um die Zusammenfassung der Informationen aus ODX-Dateien in das „intermediate“ Konfigurationsformat vorzunehmen (siehe Kapitel 5.2.3).

Das MPAS Add-in und die MDD Daemon-Software wurden in ihrer Grundidee, Struktur und Funktionsweise so dokumentiert, dass zukünftige Weiterentwicklungen einfach möglich sind. Die Dokumentation beinhaltet zudem Beschreibungen des Befehlssatzes um das MDD zu konfigurieren, des XML-Konfigurationsformats und des Byteprotokolls der ISO TP-Frames über TCP. Das Buildroot-Projekt wurde in seinem Aufbau dokumentiert. Artefakte zur Entwicklung von Cross-Plattform-Anwendungen, z.B. ein Remote-Debugger und Cross-Compiler, wurden gesichert abgelegt und beschrieben, um einfach neue Entwicklungsumgebungen einrichten zu können.

6.2 Ausblick

Während der Arbeit sind einige neue Perspektiven entstanden, die für die zukünftige Nutzung des MDD und des Add-ins sowie möglicher Weiterentwicklungen interessant sein könnten.

Neue Hardware / CAN FD Der MDD Daemon wurde absichtlich generisch implementiert. Das ermöglicht viele Parameter zur Kompilierzeit anzupassen. Prinzipiell sollte es deshalb möglich sein den MDD Daemon auf einer Vielzahl Linux-basierter Plattformen mit verschiedenen CAN-Controllern zu nutzen. Dazu gehören auch Plattformen, die CAN FD unterstützen, da das ISO TP-Modul den Unterschied zwischen CAN und CAN FD aus Sicht des MDD abstrahiert. Während der Entwicklung wurde eine Umgebung genutzt, in der der MDD Daemon sowohl testweise für aktuelle Kernelversionen (5.4.x und neuer), als auch für den 2.6.31-Kernel kompiliert wurde. Damit ist davon auszugehen, dass alle diesbezüglich kritischen Stellen durch entsprechende Kernelversion-Guards geschützt sind. Das heißt auch Plattformen mit moderneren Kernelversionen sollten kein Problem sein.

Automatische Verarbeitung von ODX-Datensätzen Wie bereits während der Umsetzung angemerkt (vergleiche Kapitel 5.2.3), wurde sich während der Arbeit dazu entschieden das vollständig automatisierte Parsen von ODX-Dateien nicht weiter zu verfolgen. Natürlich wäre es deshalb dennoch wünschenswert, das MPAS Add-in diesbezüglich

noch umfangreicher zu automatisieren. In der Endstufe dieser Automatisierung müsste für jedes neu zu testende Fahrzeug nur ein Verweis auf ein PDX-Archiv und die Angabe welche Messwerte relevant sind, für eine erfolgreiche Messung hinterlegt werden. Aus den bereits genannten Gründen bedürfte dies aber fortführende und zeitaufwendige Untersuchungen/Tests an einer weitaus größeren Variation von Fahrzeugen, Steuergeräten und PDX-Archiven. Nur dann könnte sichergestellt werden, dass mit all den möglichen Variationen eine stabile Automatisierung möglich ist. Die immense Datenmenge und die entsprechend nötige Geschwindigkeit sind dabei ebenso wenig zu vernachlässigen. Ein Zwischenformat, in dem die Archive vorab zusammengefasst werden könnten, besteht bereits.

Literaturverzeichnis

[AMMC17] A. M. Asprilla, W. H. Martinez, L. E. Muñoz, and C. A. Cortés. Design of an embedded hardware for motor control of a high performance electric vehicle. In *2017 IEEE Workshop on Power Electronics and Power Quality Applications (PEPQA)*, pages 1–5, 2017.

[ASA08] ASAM. *ASAM MCD-2 D (ODX)*, 2008.

[BBB+94] Roland Bent, Jörg Böttcher, Martina Bruland, Thilo Heimbold, Michael Kessler, Thomas Klatt, Werner Kriesel, Henning Nierhaus, Andreas Pech, Peter Roersch, Andreas Schiff, A. Schimmele, Andreas Schmitz, Gerhard Schnell, and Raimund Sommer. *Bussysteme in der Automatisierungtechnik*. Vieweg, 1994.

[Boe84] Barry Boehm. *Verifying and Validating Software Requirements Specifications*. IEEE Software, 1984.

[Bui20] Buildroot Association. Buildroot. https://buildroot.org, abgerufen 2020.

[Cla20] Danny Claus. Scrum: Ein Reality Check. https://blog.doubleslash.de/scrum-ein-reality-check/, abgerufen 2020.

[Cou18] Council on type-approval of motor vehicles. *Commission Regulation (EU) 2017/1151 (Euro 5/6)*. European Parliament, 2018.

[DKS11] M. Diana Marieska, A. I. Kistijantoro, and M. Subair. Analysis and benchmarking performance of Real Time Patch Linux and Xenomai in serving a real time application. In *Proceedings of the 2011 International Conference on Electrical Engineering and Informatics*, pages 1–6, 2011.

[Gen17] J. Gentner. Skriptum Automatisierungstechnik. Technical report, Hochschule Karlsruhe - Technik und Wirtschaft, 2017.

[Har12] Oliver Hartkopp. The CAN networking subsystem of the Linux kernel. In *iCC 2012 CAN in Automation*, 2012.

[Har20a] Oliver Hartkopp. [PATCH 08/17] can: add ISO 15765-2:2016 transport protocol. Technical report, Linux Kernel Mailing List, 2020.

[Har20b] Oliver Hartkopp. can-isotp. https://github.com/hartkopp/can-isotp, abgerufen 2020.

[HTK+20] Oliver Hartkopp, Urs Thuermann, Jan Kizka, Wolfgang Grandegger, Robert Schwebel, Marc Kleine-Budde, Benedikt Spranger, Thomas Gleixner, Andrey Volkov, Matthias Brukner, Klaus Hitschler, Uwe Koppe, Michael Schulze, Pavel Pisa, Sascha Hauer, Sebastian Haas, Markus Plessing, Per Dalen, and Sam Ravnborg. *SocketCAN*. kernel.org, abgerufen 2020.

[ISO05] ISO. *Road vehicles - Diagnostics on Controller Area Networks (CAN) - Requirements for emissions-related systems*, 2005.

[ISO17] ISO. *Road vehicles - Unified diagnosticservices (UDS) - Specification andrequirements*, 2017.

[Jos20] Andrew Josey. POSIX™ 1003.1 Frequently Asked Questions (FAQ Version 1.18). Technical report, Opengroup, 2020.

[ker20] kernel.org. *Kconfig Language*, abgerufen 2020.

[Kha20] Leyla Khamidullina. Scrum. Expectations vs reality. https://medium.flatstack.com/scrum-expectations-vs-reality-ad9891b6d867, abgerufen 2020.

[Kra15] Andreas Krawitz. MCAN Firmware und Treiber mit DBC-Interpreter für einen CAN-Bus-Ethernet-Wandler. Master's thesis, Hochschule Karlsruhe - Technik und Wirtschaft, 2015.

[KRV16] P. Kulkarni, P. K. Rajani, and K. Varma. Development of On Board Diagnostics (OBD) testing tool to scan emission control system. In *2016 International Conference on Computing Communication Control and automation (ICCUBEA)*, pages 1–4, 2016.

[Lin15] Linux Foundation. Linux Standard Base. Technical report, Linux Foundation, 2015.

[LO11] Wolfhard Lawrenz and Nils Obermöller. *CAN Controller Area Network: Grundlagen, Design, Anwendung, Teststechnik*, volume 5. VDE Verlag, 2011.

[Net89] Network Working Group. *RFC 1122 - Requirements for Internet Hosts*. Internet Engineering Task Force, 1989.

[RS19] Steffan Rauh and Andreas Schmidt. Test der Bitratenumschaltung bei CAN-FD. Technical report, GÖPEL electronic GmbH, 2019.

[SAE06] SAE. *SAE J1979/ISO 15031-5*, 2006.

[SBP07] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. The Linux Kernel Module Programming Guide. Technical report, The Linux Documentation Project, 2007.

[Sch04] Ken Schwaber. *Agile project management with Scrum.* Microsoft Press, 2004.

[SNV15] D. Saranyaraj, R. Nandhakishore, and P. Venkatesh. Benchmarking and analysis of can transmision on real-time environment. In *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, pages 399–404, 2015.

[SPP+10] M. Sojka, P. Píša, M. Petera, O. Špinka, and Z. Hanzálek. A comparison of Linux CAN drivers and their applications. In *International Symposium on Industrial Embedded System (SIES)*, pages 18–27, 2010.

[Ste90] W. Richard Stevens. *UNIX network programming.* Prentice Hall software series. Prentice Hall, Englewood Cliffs, N.J., 1990. Literaturverz. S. 737 - 747IMD-Felder maschinell generiert (GBV).

[SZ11a] Jörg Supke and Werner Zimmermann. Diagnosesysteme im Automobil - ODX - Open Diagnostic data eXchange nach ISO 22901-1. Technical report, emotive, 2011.

[SZ11b] Jörg Supke and Werner Zimmermann. Diagnosesysteme im Automobil - Transport- & Diagnoseprotokolle. Technical report, emotive, 2011.

[Tor09] Linus Torvalds. Linux 2.6.31. Technical report, Linux Kernel Mailing List, 2009.

[WW06] Jürgen Wolf and Klaus-Jürgen Wolf. *Linux-Unix-Programmierung - online Version.* Rheinwerk openbook, 2006.

[ZS14] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik.* Springer Vieweg, 2014.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACK Acknowledge
API Application Programming Interface
APM Asynchrones Programmiermodell
BCM Broadcast Manager
BRS Bit Rate Switch
BSP Board Support Packages
BS Block Size
B Byte
CAN Controller Area Network
CA Collision Avoidance
CD Collision Detection
CF Consecutive Frame
CLI Command Line Interface
CRC Cyclic redundancy check
CRUD Create Remove Update Delete
CSMA Carrier Sense Multiple Access
DLC Data length code
DL Data Length
DTC Diagnostic Trouble Code
ECM Engine Control Module
ECU Electronic Control Unit
EOF End Of Frame
EOL End Of Line
ESI Error State Indicator
ETX End of Text
FC Frame Control
FDF FD Format
FD Flexible Data Rate
FF First Frame
FIFO First In First Out
FS Flow Status
GDB GNU Debugger
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
IDE Identifier Extension
IFS Interframe Space
IP Internet Protokoll
ISO International Standards Organization
LEV Subfunction Level Byte
LIN Local Interconnect Network
LSB Linux Standard Base
LWL Lichtwellenleiter
MDD Measurement and Diagnostic Device
MOST Media Oriented Systems Transport
MPAS Multi Process Automation System
MVC Model View Controller
MVVM Model View ViewModel
NRZ Non Return to Zero
NTP Network Time Protocol
NaN Not a Number
OBD On-Board-Diagnose

ODX Open Diagnostic Data Exchange

OSI Open Systems Interconnection

PCI Protocol Control Information

PDX Packaged ODX

PID Parameter ID

POSIX Portable Operating System Interface

RPC Remote Procedure Call

RTR Remote Transmission Request

RTT Round Trip Time

SAE Society of Automotive Engineers

SDK Software Development Kit

SF Single Frame

SID Service ID

SN Sequence Number

SRR Substitute Remote Request

SSH Secure Shell

STC Stuff Count

STX Start of Text

ST Seperation Time

TAP Task-based Asynchronous Pattern

TCM Transmission Control Module

TCP Transmission Control Protocol

TP Transport Protocol

T Troughput

UDP User Datagram Protocol

UDS Unified Diagnostic Services

VB Visual Basic

VG Virtual Gateway

VIN Vehicle Identification Number

VM Virtuelle Maschine

WPF Windows Presentation Foundation

WWH-OBD World-Wide Harmonized OBD

XAML Extensible Application Markup Language

fpp frames per package

fpw frames per window