Linköpings universitet | Institutionen för Datavetenskap
Examensarbete på grundnivå, 16hp | Datateknik
Vårtermin 2020

Effektivare fordonsdiagnostik över CAN-bussen genom UDS

Kandidatarbete

Michael Abraham

Handledare: John Tinnerholm
Examinator: Jonas Wallgren

Linköpings universitet
SE-581 83 Linköping
013-28 10 00, www.liu.se

Upphovsrätt

Copyright

Sammanfattning

Bilar blir allt mer tekniskt avancerade och fler ECU:er utvecklas som medför ökad säkerhet och komfort samt minskad miljöpåverkan. Detta resulterar i ett komplext arbete med att testa och verifiera att alla olika ECU:er fungerar som de skall i olika situationer. Fordonsdiagnostik kräver ofta programvaror från olika aktörer där licenserna ofta är dyra. Idag använder Syntronic AB en programvara med mycket större funktionalitet än de behöver för att utföra fordonsdiagnostik och all denna onödiga funktionalitet i programvaran medför längre körtider. Genom att studera CAN och UDS samt analysera hur de samverkar kunde en programvara skapas genom att systematiskt utveckla programvaran med två interface inkopplade i var sin dator och kontinuerligt testa implementationen mot den teoretiska grunden för att slutligen testa programvaran i en bil. Den skapade programvaran var bättre anpassat för företagets behov och den mer funktionalitetsanpassade programvaran kunde utföra samma diagnostik snabbare än företagets nuvarande programvara. Den UDS-tjänst företaget använde mest hann implementeras men den skapade programvaran möjliggjorde att fler UDS-tjänster kunde läggas till utan modifikation av huvudprogrammet eller dess funktioner.

Abstract

Cars are getting more advanced and more ECUs are being developed that results in increased safety and comfort, and a lower environmental impact. This leads to a complex work to test and verify that all the different ECUs are functioning as intended in various situations. Vehicle diagnostics often requires software from third parties that are often quite expensive. Syntronic AB are currently using software with a much larger functionality than needed to perform vehicle diagnostics and much of the unnecessary functionality in the software leads to a longer runtime for the program. By studying CAN and UDS and analyzing how they interact, a software could be created by systematically developing the software with two interfaces connected to each computer and continuously testing the implementation against the theoretical basis and then finally testing the software in a vehicle. The created software was better suited to the needs of the company and the more functionality-adapted software could perform the same diagnostics faster than the company's current software. The most used UDS service by the company could be implemented but the created software enabled more UDS services to be added without modifications of the main program or its features.

Förkortningar och begrepp

CAN – *Controller Area Network*

CarDiagnosisProgram123 – fiktivt namn för den hemligstämplade programvaran som i dagsläget används vid testning.

CF – *Consecutive Frame*

DID – *Diagnostic Identifier*

DTC – *Diagnostic Trouble Code*

ECU – elektronisk styrenhet

FCF – *Flow Control Frame*

FF – *First Frame*

ISO – *International Organization for Standardization*

Klient – diagnostiskt kommunikationsverktyg

Nod – ECU och/eller sensor

NR_SID – *Negative Response Service ID*

NRC – *Negative Response Code*

OBD – *On Board Diagnostics*

OEM – *Original Equipment Manufacturer*

OSI – *Open Systems Interconnection*

PCI – *Protocol Control Information*

REC – *Receive Error Counter*

RTR – *Remote Transmit Request*

SAE – *SAE International*

UDS-server – en funktion som kan utföra UDS-tjänster och som är en del av en ECU

SF – *Single Frame*

SID – *Service Identifier*

Subfunktion – Vissa UDS-tjänster har möjligheten att kalla på en funktion som är direkt kopplad till en UDS-tjänsts SID värde

TEC – *Transmit Error Counter*

UDS-tjänst – en diagnostisk tjänst för att kunna läsa/skriva/ändra data på en ECU/ UDS-server

UDS – *Unified Diagnostic Service*

Innehållsförteckning

1 Introduktion

Detta är en kandidatuppsats som utförts för Syntronic AB i Linköping. De arbetar med utveckling av nya bilmodeller genom testning av delsystem i en bil.

1.1 Bakgrund

Bilar har blivit allt mer tekniskt avancerade sedan de uppfanns. 1986 introducerade Bosch *Controller Area Network* (CAN), en ny standard som skulle komma att driva utvecklingen än mer. Innan denna standard infördes kommunicerade olika sensorer och elektroniska styrenheter (ECU) (kommer inkluderas i noder framöver) direkt med varandra. Med tiden implementerades flera elektriska komponenter till bilen för att öka säkerhet och komfort samt för att minska miljöpåverkan. Mellan 1990 och 2000 ökade antalet noder från 10 till 40, och idag kan det vara mer än 70 noder i bilarna. Det tidigare tillvägagångssättet var inte hållbart då det blev väldigt dyrt, dessutom var inte kommunikationen mellan de olika noderna tillförlitlig. Lösningen till detta var CAN-bussen, en databuss som onödiggjorde kablage mellan alla olika noder. Utan CAN-bussen är det inte säkert att fordonsutvecklingen hade kunnat fortgå som det gjort sedan 1990-talet [1]. Detta medförde dock nya problem då testningen och felsökningen bland alla noder var ett komplext och tidskrävande arbete.

Innan *Unified Diagnostic Services* (UDS) protokollet standardiserades kunde biltillverkare välja att implementera olika diagnostiska protokoll på fordonets ECU:er vilket innebar att fordonsutvecklare och mekaniker inte kunde använda ett diagnostiskt verktyg för att kommunicera genom CAN-bussen på fordon från olika fordonstillverkare. Med UDS kan ett diagnostiskt verktyg kommunicera med alla noder på CAN-bussen som har stöd för UDS vilket är en majoritet av ECU:erna i en modern bil. UDS protokollet har även definierat vilka diagnostiska tjänster som finns och hur de ska användas.

1.2 Motivering

Syntronic arbetar med testning av ett delsystem i ett fordon. Testningen sker genom att läsa värdena från en ECU när bilen står still. För att få ut datat från ECU:n måste livesignaler (meddelanden på CAN bussen) skickas och läsas. Det krävs omfattande testning i olika situationer för att kunna avgöra om allt fungerar som det skall eller ej. Idag används ett flertal olika diagnostiska verktyg från olika tillverkare för att utföra olika delar av testningen. Dessa olika verktyg har ofta dyra licenser och de är inte skräddarsydda efter Syntronics behov. Idag har Syntronic ett interface från Vector och en hemligstämplad programvara (CarDiagnosisProgram123) som utför diagnostiken enligt UDS protokollet. CarDiagnosisProgram123 är en avancerad programvara som har många funktioner, men den används idag enbart för att skicka UDS förfrågningar och för att ta emot det returnerade svaret. För denna funktionalitet är CarDiagnosisProgram123 för avancerad och overhead tiden vid uppstart och nedstängning av programmet påverkas negativt av detta. Den totala körtiden för att köra en UDS förfrågan har därmed en viss overhead tid och det tar ungefär 30 sekunder från att CarDiagnosisProgram123 startats innan den första UDS förfrågan kan skickas. Övrig funktionalitet har vissa tidsförluster då användaren måste navigera i olika menyer för att kunna utföra de olika testerna. Om testerna kan utföras snabbare kan antingen fler tester köras under samma testperiod vilket innebär en säkrare bekräftelse att allt fungerar, eller så kan bilen bli färdigtestad tidigare vilket innebär att fler modeller kan testas.

1.3 Mål

Målet med detta kandidatarbete är att skapa en programvara som är snabbare än CarDiagnosisProgram123. Detta kommer testas genom att jämföra den egenskapade programvaran mot referenstider som sattes av CarDiagnosisProgram123. Den egenskapade programvaran ska vara uppbyggd så att fler UDS-tjänster ska kunna läggas till i efterhand utan större modifikation. Programmet ska kunna automatisera givna testfall, tillåta användaren att skapa enskilda testfall, automatiskt spara ner svaren från UDS förfrågningarna på en textfil, och vara snabbare än referenstiderna som tagits med CarDiagnosisProgram123. Detta tros kunna göras genom en mjukvara som är specialiserad för Syntronics behov och bara implementera den funktionalitet som faktiskt används samt genom att undvika menyer som används för att växla mellan att skicka enskilda UDS förfrågningar och köra listan med testfall. För att kunna skapa programvaran kommer både CAN protokollet och UDS protokollet behöva undersökas noggrant.

1.4 Problemformulering

1) Hur är UDS protokollet implementerat över CAN-bussen?
2) Hur kan en internt utvecklad mjukvara som ska utföra fordonsdiagnostik enligt UDS protokollet, genom automatiserade tester och automatisk datainsamling konstrueras så att total körtid, inklusive uppstart av mjukvaran, är snabbare än referenstiderna som sattes av CarDiagnosisProgram123?

1.4.1 Önskvärda krav

Mjukvaran ska även:

3) kunna kommunicera på CAN bussen med flera interface.
4) kunna live-visualisera data som kommer från UDS kommandona.
5) visualisera autosparad data.

1.5 Avgränsningar

Denna rapport hanterar UDS över CAN. UDS kan implementeras på andra nätverk som LIN och Ethernet, men detta ligger utanför detta arbetes omfång och inget fokus kommer läggas på det.

Alla UDS-tjänster kommer inte implementeras då det är ett komplext arbete som inte kommer kunna utföras under ett kandidatarbetes omfång. Fokus lades på UDS-tjänster som var av intresse för Syntronic. UDS-tjänster som kräver särskild behörighet kommer inte heller implementeras då ECU:ernas säkerhetsalgoritm inte innehas av författaren.

Fordonstillverkarna, vilket/vilka delsystem som testas, den nuvarande mjukvaran samt värden från konfigurationsfilerna de får in från fordonstillverkarna är konfidentiellt och kommer inte beskrivas i rapporten. Detta gäller även antalet UDS förfrågningar som fanns på den fördefinierade listan. Antalet UDS förfrågningar på listan kommer benämnas med x.

2 Relaterade arbeten

Detta arbete var tidsbegränsat och alla tidigare arbeten inom området kunde inte undersökas, därmed kan relevanta arbeten ha missats. Tre relaterade arbeten som var nära detta arbete eller till och med gjorde delar som detta kandidatarbete behandlade hittades.

Salcianu och Fosalau [4] byggde en diagnostisk verktygssimulator och felgenerator som kommunicerade över CAN-bussen och baserades på UDS protokollet. Syftet med den diagnosiska verktygssimulatorn var att hitta eventuella fel i fordonets ECU men även att hitta kommunikationsfel som kan ha inträffat mellan olika noder. För att få simulatorn att fungera behövde bilen kommunicera via CAN-bussen och vara baserad på UDS protokollet. Felgeneratorn kunde generera olika fel och kunde tillsammans med den diagnostiska verktygssimulatorn testa olika ECU:er på CAN-bussen.

Jinghua och Feng [6] skapade en automatiserad testmetod baserat på UDS protokollet. För att kunna göra detta använde de AutoCAN[1] från ihr. De importerade en ODX-fil för att skapa en script-generator som skapar test baserat på en script-mall samt en importerad databasfil och användarens konfigurationer. Script-generatorn skapar nya testfall som sedan körs. När testet var klart skapades en rapport automatiskt med detaljerat data och resultat.

Assawinjaipetch et al. [14] implementerade UDS på en ECU medan detta kandidatarbete implementerade UDS på en klient. De beskrev hur de vill att deras implementation skulle agera med UDS implementerat samt hur de ville att UDS agerade på nätverkslagret och transportlagret. Detta baserades på ISO25765-2 som detta arbete inte haft tillgång till vilket innebar att en stor del av förståelsen för UDS på nätverkslagret och transportlagret kom från denna artikel.

[1] https://www.ihr.de/index.php/en/produkte-can-english/388-auto-can-english

3 Teori

I detta kapitel kommer teorin som undersökningen vilar på läggas fram. Voss [2] säger att CAN-bussen används inom flera områden, speciellt inom inbyggda system. Exempel på områden är personbilar, hissar, flygplan och maskiner inom industrin, men i och med att undersökningens fokus ligger på personbilar kommer teorin att baseras på användningen i personbilar.

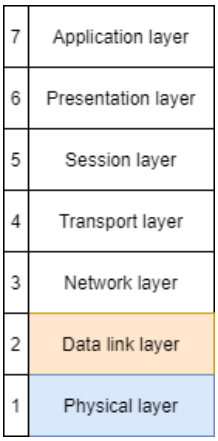
3.1 ECU

En elektronisk styrenhet (ECU) i ett fordon är ett inbyggt system som kontrollerar ett antal sensorer och ställdon. Varje ECU har olika funktioner i fordonet och behöver kunna kommunicera med andra ECU:er över CAN-bussen [21]. ECU:erna har både hårdvara och mjukvara för att kunna utföra sina funktioner där hårdvaran innehåller minst en sorts minne. Mjukvaran innehåller även metadata för ECU:n för att kunna identifiera och utföra olika UDS-tjänster. ISO15765-3 [16] anger att varje ECU ska ha en unik adress sparad i sitt minne. Exempel på komponenter i en modern personbil som styrs av en eller flera ECU:er är en automatisk växellåda, airbagsystemet och ABS.

3.2 CAN

Kommunikationen mellan de olika noderna på fordonet sker genom CAN-bussen. Alla noder kan kommunicera med varandra, och när ett meddelande skickas från en nod har det ingen specifik adress utan meddelandet skickas ut på CAN-bussen. Dekanic et al. [3] hävdar att fördelen med CAN är att den har en hög immunitet mot elektriska störningar och att detta är en av anledningarna till att det fungerar bra i personbilar. Pazul [7] anser att en annan fördel med detta protokoll är att nya noder kan läggas till på databussen utan att övriga noder nödvändigtvis behöver programmeras om. Voss [2], Pazul [7], Assawinjaipetch et al. [14] och Cia [20] säger att CAN är uppbyggt efter ISO/OSI 7 lagers modellen som visas i figur 1. Voss [2] och Cia [20] påstår dock att CAN använder lager 1, 2 och 7 medan Pazul [7] och Assawinjaipetch et al. [14] påstår att CAN använder lager 1 och 2. Båda dessa tolkningar kan dock anses vara korrekta. Lager 1 och 2 är standardiserade enligt standarden för CAN (ISO 11898), men Voss [2] och Cia [20] menar att det finns vissa standardiserade CAN protokoll i applikationslagret såsom CANopen[2]. Voss [2] belyser att applikationslagret interagerar med applikationen eller operativsystemet på noden medan datalänkslagret och det fysiska lagret är integrerat på nodens chip.

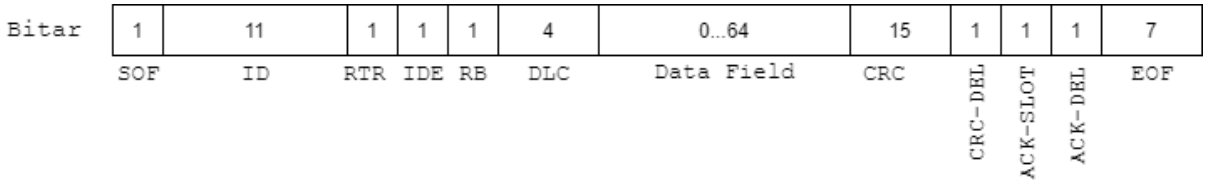
[2] https://www.can-cia.org/canopen/



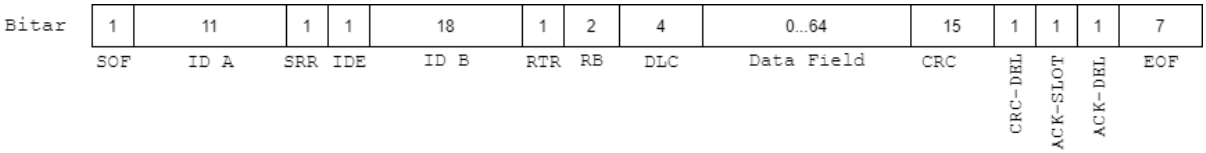
Figur 1. OSI modell över CAN som är implementerat på det första och andra lagret enligt ISO11898-1.

3.2.1 Meddelanden

Det finns två format på meddelandena som skickas över CAN-bussen. Figur 2 illustrerar ett meddelande i standardform, även känt som CAN 2.0A. Det andra formatet, CAN 2.0B, illustreras i figur 3 och är ett utvidgat meddelande där ID fältet är 29 bitar istället för standardformens 11 bitar i ID fältet. Båda dessa meddelandeformat kan skickas på samma databuss. CAN-bussen är ett realtidssystem med ett prioriteringssystem för meddelanden. Det är meddelandets ID, som är unikt för varje meddelande, som bestämmer prioriteten. I CAN protokollet anses en logisk 0:a vara dominant över en logisk 1:a vilket innebär att ju lägre värde i ID fältet ett meddelande har, desto högre prioritet har meddelandet [7]. Meddelandets datafält är 64 bitar långt och det är i detta fält som meddelandets data finns. Meddelandet har även en *checksum* i CRC fältet som kontrollerar att meddelandet är oförvanskat. [3, 7]



Figur 2. Ett CAN meddelande i standardform. I databussen har meddelande i standardform högre prioritet än utvidgade meddelanden. IDE biten är alltid satt till 0 i meddelanden i standardform.



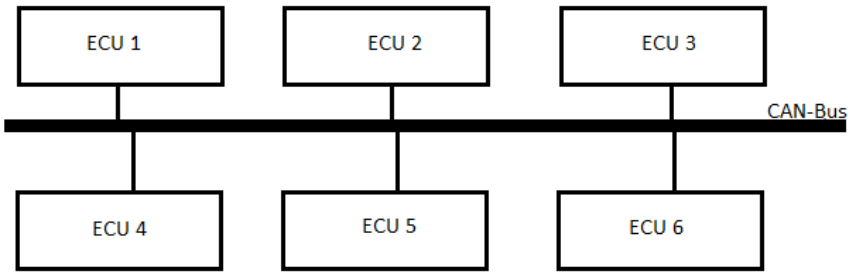
Figur 3. Ett utvidgat meddelande. SRR biten måste vara satt till 1 i och med att meddelanden på standardform har högre prioritet. IDE biten är alltid satt till 1 i utvidgade meddelanden vilket anger att meddelandet är i utvidgad form. ID fältet är uppdelat i två olika fällt där ID A är den första delen av det unika identifikationsnumret och innehåller bit 28 till 18, och ID B är den andra delen av det unika identifikationsnumret och innehåller bit 17 till 0.

Det finns fyra olika meddelandetyper där den vanligaste typen är ett datameddelande som skickas av en nod ut på databussen. För vanliga meddelandetyper är *remote transmit request* (RTR) biten alltid satt

till 0. Den andra meddelandetypen är meddelandebegäran och på dessa meddelanden är RTR biten alltid satt till 1. Meddelandebegäran är när en nod begär information från andra noder. Ett exempel på detta är när en nod vill ha information om oljetemperaturen som en annan nod har tillgång till. Den tredje meddelandetypen är felmeddelanden och dessa skickas om något fel upptäckts på databussen. Exempel på varför ett felmeddelande skickas är att *checksum* i CRC fältet är felaktigt när mottagarnoden jämför sitt beräknade värde jämfört med den skickande nodens *checksum* värde. Den fjärde meddelandetypen är kö-meddelanden och skapas när en nod behöver mer tid för att behandla ett mottaget meddelande. I detta scenario skickas kö-meddelandet ut på databussen och meddelar att noden som skickade nämnda kö-meddelande inte är redo att ta emot nya meddelanden under en viss tidsperiod. [7, 10]

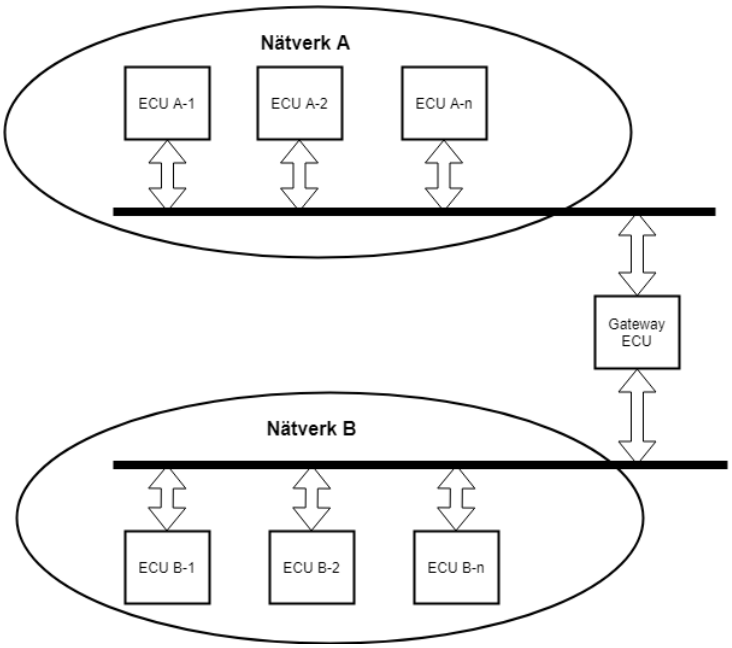
3.2.2 CAN-bussen

När CAN-bussen introducerades försvann behovet av att ha alla olika noder direkt ihopkopplade med kablar mellan varandra. De olika noderna i bilen kopplades istället upp på en seriell databuss och kunde kommunicera med varandra via databussen. Dekanic et al. [3] säger att den maximala signalhastigheten är 1 Mbps och att maximalt 30 noder kan vara uppkopplade på en buss. Figur 4.1 illustrerar idén med CAN-bussen.



Figur 4.1. Exempel på olika noder anslutna till CAN-bussen. Om exempelvis ECU 1 skickar ett meddelande som är av intresse för ECU 3 kommer meddelandet att skickas ut på databussen och inte direkt till ECU 3. Alla noder på bussen kan läsa meddelandet, men det kommer bara att accepteras av noder som är programmerade till att acceptera meddelanden med det specifika identifikationsnumret meddelandet har. När meddelandet accepterats sätts ACK Slot biten till 0 om meddelandet var oförvanskat. Exempelvis kommer ECU 5 kunna se meddelandet, men när den läser identifikationsnumret i meddelandet kommer den att ignorera meddelandet.

CAN-bussen består av ett tvinnat kabelpar där ena kabeln har *CAN-high* signaler och den andra kabeln har *CAN-low* signaler. För att få bra signalstyrka behöver varje ände av CAN-bussen ha ett 120 ohms motstånd [5]. Ett fordon kan ha över 70 noder och maximalt 30 noder kan vara uppkopplade på en buss. För att kunna koppla ihop olika bussar används en *gateway* ECU vars roll är att dirigera trafik mellan olika subnätverk så att meddelanden transporteras till rätt nod. Varje nod i ett subnätverk måste ha samma nätmask och nätverksadress, men nodens adress måste vara unik. Figur 4.2 visar en förenklad bild av *gateway* ECU:ns roll [13, 16].



Figur 4.2. gateway ECU:n dirigerar trafiken mellan de olika subnätverken. Detta möjliggör att exempelvis ECU A-1 kan skicka ett meddelande som ska till ECU B-2. Det kan finnas flera subnätverk, och ett subnätverk kan ha en egen gateway ECU som dirigerar trafik till ett subnätverk inom subnätverket [16].

Pazul [7] säger att CAN protokollet är ett *Carrier Sense Multiple Access with Collision Detection* protokoll vilket innebär att varje nod behöver övervaka databussen en viss tid och detektera att ingen aktivitet skett innan de kan skicka ett meddelande. När denna aktivitetsfria tid är avklarad har alla noder möjlighet att skicka meddelanden. Två eller fler noder kan skicka meddelanden samtidigt, men protokollet är byggt på så vis att inget data förloras genom kollisionshantering. Om två meddelanden skickas samtidigt kommer meddelandet med högst prioritet få tillgång till databussen medan meddelandet med lägre prioritet stoppas. När det högre prioriterade meddelandet är skickat kommer noden som ville skicka det stoppade meddelandet att lyssna på databussen igen och efter att ingen aktivitet noterats på databussen under en viss tid kommer noden försöka skicka meddelandet igen [3, 7].

3.2.3 Fysisk och funktionell adressering

Det finns två metoder för att skicka meddelanden över CAN-bussen. Det sker antingen genom fysisk adressering eller funktionell adressering. Det är avsändaren av meddelandet som bestämmer vilken sorts adressering som sker genom CAN meddelandets ID-fält. Vid fysisk adressering skickas meddelandet till en specifik nod vilket sker när avsändarnoden vet adressen till mottagarnoden. Vid funktionell adressering sänder avsändarnoden meddelandet på CAN-bussen utan att veta mottagarnodens adress. ISO15765-3 [16] anger att mottagaradressen ska vara 0x7FF för att sända ut meddelandet över CAN-bussen [15, 16]. Med funktionell adressering finns möjligheten att ett meddelande hanteras av flera noder.

3.2.4 Diagnostik på CAN-bussen

För att kunna övervaka nodernas status på CAN-bussen integrerades *On board diagnostics* (OBD). Den andra generationen heter OBD II och har funnits sedan 1996. OBD II systemet övervakar varje nod som kan påverka bilens funktionalitet. Det är detta system som får varningslampor i fordonet att lysa om

något fel upptäckts och förutom att tända lampan så sparar systemet viktig information om det upptäckta felet som en diagnostisk felkod (DTC) [8]. OBD II uttaget ska vara placerat på en plats i fordonet man kan nå från förarsätet och när man ansluter ett diagnostiskt verktyg (klient) till uttaget agerar det som en nod på databussen. Klienter som kan anslutas till OBD II uttaget är olika avancerade, de billigare verktygen kan läsa vissa meddelanden och eventuellt släcka något felmeddelande som en servicelampa, medan mer avancerade verktyg har större tillgång till meddelandena på CAN-bussen samt har möjlighet att låsa upp fler funktioner från ECU:erna. Fordonsutvecklare behöver ofta köpa in dyra verktyg från *Orignal Equipment Manufacturer* (OEM), vilket i detta fall är fordonstillverkarna, för att kunna testa olika delsystem i bilarna. Klienter behöver inte alltid kopplas upp via OBD II uttaget utan de kan även kopplas upp direkt på noden som ska testas genom specialutrustning.

Kelkar och Kamal [9] säger att CAN hanterar fel på nodnivå och att felhanteringen beror på nodens beteende i något av de tre tillstånden noden kan befinna sig i, aktiv, passiv eller *bus-off*. En nod som visar felaktigt beteende stängs av för att säkerställa att databussen kan fortsätta fungera som tänkt. Varje nod har två fel-räknare, *Transmit Error Counter* (TEC) och *Receive Error Counter* (REC) och det finns flera regler som bestämmer om hur dessa fel-räknare ska öka eller sänka sitt värde. I korthet ska TEC öka sitt värde snabbare än REC då det är en större chans att det är den sändande noden som har ett fel. När nodens TEC har ett värde över 255 hamnar noden i *bus-off* tillståndet och kan inte längre skicka några meddelanden på databussen.[10]

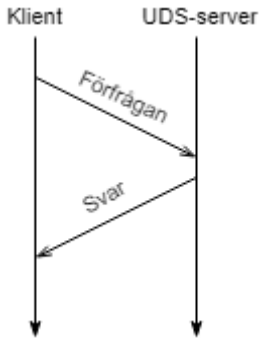
3.3 UDS

UDS är en standard definierad av *International Organisation for Standardization* (ISO). Det är en samling av diagnostiska UDS-tjänster som möjliggör diagnostik på en UDS-server. UDS är implementerat på sessionslagret och applikationslagret, i OSI modellen, men har även speciella förhållningsregler på nätverkslagret och transportlagret. Alla noder i fordonsnätverket har inte stöd för UDS, men det blir vanligare och idag har nästan alla noder i fordonsnätverket UDS implementerat. Fördelarna med UDS är många, det är lättare, både för utvecklare och mekaniker, att felsöka de olika noderna i fordonsnätverket [17]. Embitel [17] sammanfattar fyra kategorier av UDS-tjänster i ISO 14229 [15] som behandlar UDS. Dessa är:

1) dataöverförings möjligheter vilket innebär att data kan läsas och skrivas från/till en UDS-server.
2) feldiagnostik vilket innebär att när ett fel inträffar i en UDS-server så sparas en DTC i ECU:ns minne, denna DTC kan sedan läsas av en klient vilket bland annat underlättar felsökningen för en mekaniker.
3) upp-och nedladdnings möjligheter som innebär att mjukvaran i en UDS-server bland annat kan uppdateras.
4) fjärr-rutin-aktivering som kan användas om ett test behöver köras över en viss tidsperiod, exempelvis vill kanske en mekaniker testa bilens motorfläkt och spara ner datat och då kan detta läge användas.

Embitel [17] missade dock en femte kategori [19]. Kategorin de inte sammanfattade var diagnostiska sessioner som beskrivs i kapitel 3.3.1. UDS är byggt på flera olika ISO standarder som ISO15765-2, ISO15765-3 och ISO14229. Syftet med UDS är att diagnosera olika UDS-servrar i fordonsnätverket via CAN-bussen. Det fungerar genom att en klient skickar en förfrågan till en UDS-server som i sin tur returnerar ett svar vilket illustreras i figur 5. Både förfrågan och svaret följer standarden för meddelanden över CAN-bussen. Både Dekanic et al. [3] och Salcianu och Fosalau [4] bekräftar ISO14229 [15] som anger att standarden för diagnostiska tjänster har ett gemensamt meddelandeformat som innefattar förfrågningsmeddelande, positivt svarsmeddelande och negativt svarsmeddelande. Varje UDS-tjänst i UDS protokollet har en *Service Identifier (SID)* som är en byte stort och kan ha värden i intervallet 0x00

– 0xFF. Alla värden i intervallet är dock inte tillgängliga för fordonstillverkarna och utvecklare, vissa värden är reserverade för framtida bruk medan andra är reserverade av ISO 14229 och dokument [15].



Figur 5. Kommunikation initieras alltid av klienten enligt UDS protokollet medan UDS-servern svarar på klientens förfrågan

3.3.1 Diagnostiska sessioner

UDS protokollet definierar fyra olika diagnostiska sessioner som en ECU kan vara i. Endast en diagnostisk session måste vara implementerat och det är standardsessionen. När en ECU startas ska den alltid hamna i standardsessionen. UDS-tjänsterna som kan utföras i standardsessionen är begränsade och säkerhetskritiska UDS-tjänster kan kräva särskild behörighet. De tre övriga sessionerna är:

1) programmeringssession som används för att uppdatera mjukvaran i en ECU.
2) utvidgad diagnostisk session som kan användas för att få tillgång till fler UDS-tjänster exempelvis som att kalibrera sensorer.
3) säkerhetssystemets diagnostiksession som ger tillgång till säkerhetskritiska UDS-tjänster som airbagens inställningar [15].

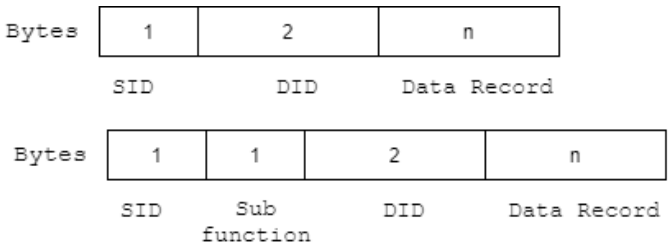
UDS-tjänsten *TesterPresent* (0x3E) används för att hålla en session vid liv. I standardsessionen behövs denna UDS-tjänst inte för att hålla sessionen vid liv, men vid aktivitet i övriga sessioner behöver klienten skicka ett förfrågningsmeddelande periodiskt, ofta med en till två sekunders intervall, med SID värdet 0x3E för att kunna hålla den nuvarande sessionen aktiv. Om en ECU befinner sig i en av de tre övriga sessionerna och 0x3E UDS-tjänsten inte efterfrågas kommer ECU:n att återgå till standardsessionen efter en viss tidsperiod [15, 22].

3.3.2 Säkerhet på UDS

Vissa UDS-tjänster har behörighetsrestriktioner såsom säkerhet mot medvetna attacker, utsläpp och bilens säkerhetssystem såsom airbagsystemet. Ett exempel på en UDS-tjänst som har behörighetsrestriktioner är uppdatering av mjukvaran på en UDS-server vilket kräver särskild åtkomst. Dessa behörighetsrestriktioner finns för att skydda elektroniken och andra komponenter i fordonet. Säkerhetskonceptet över UDS baseras på initiering – nyckel relationer. Klienten skickar ett initieringsförfrågningsmeddelande och UDS-servern returnerar initieringen. Med den mottagna initieringen kan klienten beräkna fram en nyckel genom en krypteringsalgoritm och sedan skicka nyckeln som hör till den mottagna initieringen. Om nyckeln var giltig kommer klienten få tillgång till UDS-tjänsten [15].

3.3.3 Förfrågningsmeddelande

Förfrågningsmeddelanden kan bara skickas i en riktning, från en klient till en UDS-server. Figur 6 visar två typer av förfrågningsmeddelanden. Endast ett fält är obligatoriskt, SID fältet som innehåller UDS-tjänstens id värde och är en byte stort. Detta SID värde är detsamma i klienten och på UDS-servern vilket garanterar att den efterfrågade UDS-tjänsten blir refererad till. Vissa UDS-tjänster har stöd för subfunktioner, några har stöd för två subfunktioner medan andra har stöd för flera. Om UDS-tjänsten stöder subfunktioner måste en subfunktion anges för att förfrågan ska vara giltig. Detta fält är en byte stort. Ett exempel på en subfunktions syfte är att starta eller stoppa en UDS-tjänst. Både SID och subfunktionernas värden är standardiserade i ISO14229 och är detsamma i alla fordon som har UDS implementerat. *Diagnostic ID* (DID) fältet är två byte stort i ISO14229 och dessa värden är inte standardiserade i UDS. DID värdet refererar till olika lokala dataposter i en ECU, exempel på detta är batteriets spänning, en ECU som har en sensor på batteriet kan med hjälp av DID värdet veta vilket värde som efterfrågas och returnera batteriets spänning. Att ange ett DID värde är valbart. *Data Record*, dataposten innehåller metadata och är länkat till DID värdet. Datafältet är obligatoriskt för vissa UDS-tjänster och valbart för andra UDS-tjänster. Både DID och datapostens värden är valbara för OEM. Detta innebär att varje OEM kan sätta egna värden och därmed tvinga utvecklare och mekaniker att köpa deras verktyg eller licenser. Dessa värden kan skilja sig mellan fordonsmodeller från en fordonstillverkare, men även mellan olika årgångar av samma modell. [12, 15, 18]



Figur 6. Exempel på två olika typer av förfrågningsmeddelanden. Det övre meddelandet är utan en subfunktion medan det nedre meddelandet har en subfunktion. Förfrågningsmeddelandet har ingen bestämd storlek. SID är obligatoriskt då det beskriver vilken UDS- tjänst som ska utföras. Detta innebär att övriga fält är obligatoriska/valbara beroende på vilket UDS-tjänst som efterfrågats.

Vissa UDS-tjänster har stöd för subfunktionen *suppressPosRspMsgIndicationBit*. Denna subfunktion meddelar ECU:n att ett positivt svarsmeddelande inte behöver skickas. Detta är lämpligt när det positiva svaret inte innehåller någon datapost utan mest agerar som ett kvitto för att UDS-tjänsten kunde utföras. Om UDS-tjänsten inte kunde utföras kommer ett negativt svarsmeddelande komma och för att spara på trafiken över CAN-bussen kan man därmed meddela att det positiva svarsmeddelandet inte behöver skickas. Standardvärdet i denna subfunktion är alltid satt till *False* (den sjunde biten i subfunktionens byte är satt till 0), vilket innebär att det positiva svarsmeddelandet ska skickas, och behöver inte kallas på i förfrågningsmeddelandet. Om man sätter värdet till *True* (den sjunde biten sätts till 1 i subfunktionens byte) i subfunktionen kommer svarsmeddelandet inte att skickas. Ett exempel på när denna subfunktion är lämplig att sätta till *True* är när UDS-tjänsten 0x3E används. Svaret från denna UDS-tjänst innehåller ingen datapost utan endast förfrågningsmeddelandets SID + 0x40 samt en kopia på värdet i subfunktionen och tar bara upp onödig trafik på CAN-bussen. För att sätta värdet till *True* i subfunktionen till UDS-tjänsten 0x3E ska subfunktionens värde sättas till 0x80 (sjunde biten satt till 1).

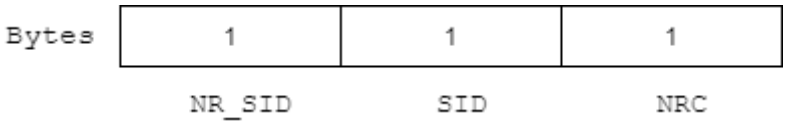
3.3.4 Positiva svarsmeddelanden

Om förfrågningsmeddelandet var korrekt utfört och UDS-tjänsten kan utföras kommer UDS-serven att utföra förfrågan och skicka tillbaka ett positivt svarsmeddelande. Formatet för positiva

svarsmeddelanden liknar formatet som visas i figur 6, men olika UDS-tjänster skickar tillbaka olika data, så formatet på svarsmeddelandet beror på förfrågningsmeddelandet och vilken UDS-tjänst som efterfrågades. Alla positiva svarsmeddelanden skickar tillbaka förfrågningsmeddelandets SID värde men adderar 0x40 till det positiva svarsmeddelandets SID fält för att påvisa att det är ett positivt svarsmeddelande vilket innebär att den sjätte biten alltid är satt till ett, det finns dock ett undantag om svarsmeddelandet är av typ två från *ReadDataByPeriodicIdentifer* UDS-tjänsten som är specificerad i ISO 14229. Om förfrågningsmeddelandet hade SID värdet 0x14 kommer det positiva svarsmeddelandet ha SID värdet 0x54. Om förfrågningsmeddelandet hade en subfunktion och/eller DID värde kommer dessa värden vara identiska i svarsmeddelandet. Om den efterfrågade UDS-tjänstens svar innehåller annat data kommer det finnas i dataposten. Innehållet i dataposten är inte definierat i ISO14229 utan det är fordonstillverkarna som definierar datats betydelse [15].

3.3.5 Negativa svarsmeddelanden

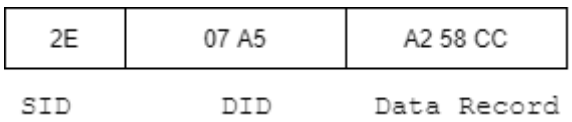
Om förfrågningsmeddelandet inte är korrekt utfört eller om UDS-tjänsten inte kan utföras kommer UDS-servern att skicka tillbaka ett negativt svarsmeddelande som visas i figur 7. Det första fältet *Negative Response Service ID* (NR_SID) är alltid 0x7F enligt UDS protokollet. Det andra fältet innehåller en kopia av förfrågningsmeddelandets SID. Det tredje fältet innehåller *Negative Response Code* (NRC) och är en byte stort. NRC innehåller ett värde som förklarar varför svaret från förfrågan blev negativt. Det finns ett flertal anledningar till att ett svar blir negativt, exempelvis kan det bero på att förfrågningsmeddelandet var felaktigt formaterat, att DID värdet inte stöds av UDS-tjänsten eller att behörighet att utföra UDS-tjänsten saknas [15].



Figur 7. Ett negativt svarsmeddelande är tre byte stort och innehåller ett negativt SID värde, förfrågningsmeddelandets SID värde och en negativ responskod.

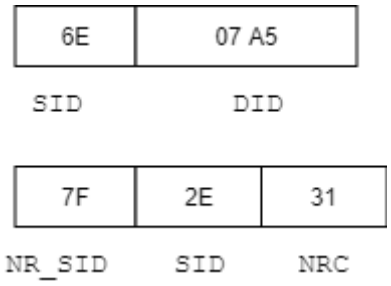
3.3.6 Exempel på ett meddelandeflöde

Detta är ett exempel på hur ett meddelandeflöde kan se ut för UDS-tjänsten *WriteDataByIdentifier* (0x2E). I denna UDS-tjänst kan inga subfunktioner läggas till. Scenariot i detta exempel är att en utvecklare vill ändra en inställning på en sensor i en ECU. SID värdet är definierat av ISO14229, men övriga värden i exemplet är fabricerade då varje fordonstillverkare har egna värden för DID och dataposten. Figur 8 visar hur ett förfrågningsmeddelande från en klient till en UDS-server kan se ut när UDS-tjänsten 0x2E efterfrågas.



Figur 8. Ett exempel på hur ett förfrågningsmeddelande kan se ut. SID för UDS-tjänsten WriteDataByIdentifier är 0x2E enligt UDS protokollet. Övriga värden är fabricerade.

När förfrågningsmeddelandet skickats till UDS-servern kan två potentiella svar komma, ett positivt svarsmeddelande eller ett negativt svarsmeddelande. Figur 9 visar exempel för båda dessa möjliga svarsmeddelanden för förfrågningsmeddelandet i figur 8.



Figur 9. Det övre meddelandet är ett positivt svarsmeddelande för UDS-tjänsten 0x2E. Den returnerar 0x2E + 0x40 som SID värde samt DID värdet från förfrågningsmeddelandet. Den nedre meddelandet är ett negativt svarsmeddelande. Detta syns tydligt då den första byten har värdet 0x7F som är reserverat för negativa svarsmeddelanden oavsett UDS-tjänst. Den andra byten är SID numret från förfrågningsmeddelandet. Den tredje byten är NRC värdet som är fördefinierade och visar varför det blev ett negativt svarsmeddelande.

För 0x2E UDS-tjänsten returneras ingen datapost, om UDS-tjänsten i förfrågningsmeddelandet kunde genomföras returneras SID värdet från förfrågningsmeddelandet + 0x40 samt DID värdet. Om UDS-tjänsten inte kunde genomföras returneras ett negativt svarsmeddelande. De olika UDS-tjänsterna har stöd för olika NRC, 0x31 i detta exempel betyder antingen att DID värdet från förfrågningsmeddelandet inte har något stöd på UDS-servern eller att DID värdet bara har stöd för att läsa och inte skriva.

3.3.7 UDS på nätverkslagret

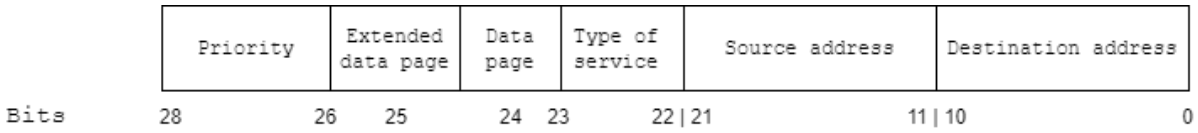
ISO15762-3 [16] beskriver hur UDS ska implementeras över CAN. Protokollet går djupt i detaljnivå men allt anses inte vara relevant för detta arbete. Ett kapitel anses dock vara högst relevant och behandlar hur CAN meddelandets ID ska användas för fordonsdiagnostik på nätverkslagret i OSI modellen. Protokollet skiljer på hur CAN meddelanden i 2.0A formatet (se figur 2) och CAN meddelanden i 2.0B formatet (se figur 3) ska användas tillsammans med UDS.

3.3.7.1 CAN 2.0A

CAN meddelanden med 2.0A formatet innefattar ett 11 bitars CAN meddelande ID och kan användas av någon av de tre övriga sessionerna som beskrivs i kapitel 3.3.1. ISO15765-3 [16] anser dock att SID 0x3E är den enda UDS-tjänsten som ska använda ett CAN meddelande av typ 2.0A.

3.3.7.2 CAN 2.0B

CAN meddelanden med 2.0B formatet innefattar ett 29 bitars CAN meddelande ID där ID:t är uppdelat i två delar, ID A med 11 bitar och ID B med 18 bitar. De 29 bitarna är fördelade enligt figur 10.



Figur 10. De 29 bitarna i CAN meddelandets ID fördelas enligt följande: De första 11 bitarna är destinationens adress, de nästkommande 11 bitarna är källans adress, sedan är bit 22 och 23 dedikerade för vilken typ av meddelande det är. Bit 24 och 25 avgör vilket format CAN ID ska ha medan bit 26–28 avgör vilken prioritet meddelandet ska ha.

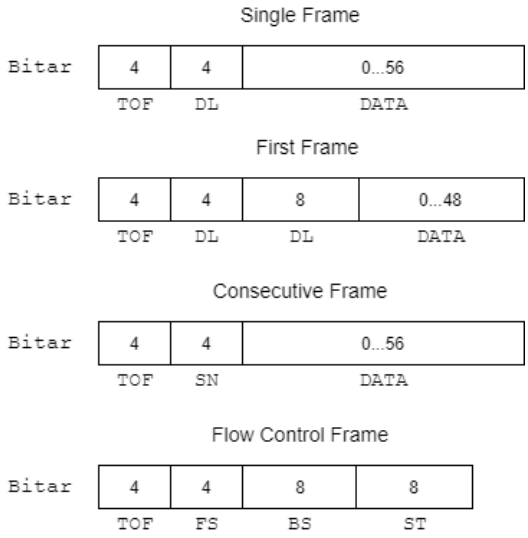
Destinationens adress är mottagarnodens adress medan källans adress är den avsändarnodens adress. Vid diagnostiska meddelanden ska prioritetsfältet ha värdet 6 (110_2 i basen 2). Övriga fält är antingen OEM specifika, reserverade av ISO eller definierade av olika protokoll. För att följa ISO15765-3, implementation av UDS på CAN, ska meddelandet formateras enligt följande, $0x6^1F^2(0\text{-}7FF)^3(0\text{-}7FF)^4$ [16]. Förklarning på meddelandets format enligt ISO15765-3 följer nedan:

1) Prioritetsfältet ska ha värdet 110_2 som är 0x6.
2) Bitarna 22–25 ska ha värdet 1111_2 som är 0xF. Bit 24 och 25 anger att det är ISO 15765 formatet som ska följas och bitarna 22 och 23 anger att meddelandetypen kommer att följa ISO 15765-3.
3) Källans adress ska vara 11 bitar vilket innebär värden i intervallet 0x0-0x7FF
4) Destinationens adress ska vara 11 bitar vilket innebär värden i intervallet 0x0-0x7FF

När ett meddelande ska skickas över fordonsnätverket finns det två olika sätt att sända meddelandet. Antingen kan meddelandet skickas till alla noder på fordonsnätverket vilket görs genom att sätta destinationens adress till 0x7FF eller så kan meddelandet skickas till noder i ett specifikt subnätverk. Varje nod på ett subnätverk kommer sedan jämföra sin unika adress med meddelandets destionationsadress och om de matchar varandra kommer informationen på meddelandet behandlas på ett högre lager i OSI modellen [16].

3.3.8 UDS på transportlagret

Datafältet i ett CAN meddelande är 64 bitar det vill säga 8 byte stort. Om ett meddelande är större än 7 byte behöver meddelandet segmenteras för att data inte ska försvinna. Utan segmenteringen kommer endast de 7 första byten skickas och övriga bytes förloras. ISO15765-2 (ISO-TP) är implementerat på transportlagret i OSI modellen och definierar hur meddelandeflödet ska utföras över CAN-bussen. För att få information om meddelandet använder transportlagret en byte av CAN meddelandets datafält för *Protocol Control Information* (PCI), som används för att kontrollera meddelandeflödet och läggs till i början av CAN meddelandets datafält som mest signifikanta byte. Det fyra olika PCI typerna, *Single Frame* (SF), *First Frame* (FF), *Consecutive Frame* (CF), och *Flow Control Frame* (FCF), illustreras i figur 11.



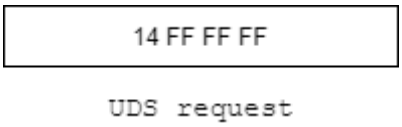
Figur 11. Single Frame har Type of Frame (TOF) värdet 0x0 som de 4 mest signifikanta bitarna och därefter 4 bitar som anger data length (DL) i bytes. Resterande 7 bytes används som datafält. First Frame har TOF värdet 0x1. Det finns 12 bitar (4+8) DL som anger hur många byte meddelandet är. Sex byte data kan skicks i ett FF. Consecutive Frame har TOF värdet 0x2 samt 4 bitar som anger sequence number (SN). SN har 4 bitar vilket innebär ett maximalt värde på 15. Om SN når 15 börjar det om från 0. För varje skickat meddelande ökar SN med 1. 7 byte data kan skickas i ett CF. Flow Control Frame har TOF värdet 0x3. Flow Status (FS) anger vilket stadie mottagarnoden befinner sig i, antingen kan noden befinna sig i stadiet redo att sända, vänta eller kö. Nästkommande byte Block Size (BS) anger hur många CS som ska skickas i samma block. Nästkommande byte Separation Time (ST) anger det minsta tidsintervallet som måste ha passerats innan nästa CF kan skickas. Ett ISO-TP meddelande är alltid 8 byte stort och icke använda bytes fylls på med utfyllnadsbytes med värdet 0xAA, 0x55 eller 0x00.

Om hela datafältets innehåll får plats i ett CAN meddelande används SF. Detta innebär att datafältets innehåll kan vara maximalt 7 byte stort då PCI tar upp den åttonde byten. Om datafältets innehåll inte får plats i ett CAN meddelande delas datafältets innehåll upp i flera CAN meddelanden. Det första CAN meddelandet som skickas är FF som innehåller de sex första byten av datafältets innehåll samt storleken för de resterande CAN meddelandena. Mottagaren av FF skickar FCF som innehåller överföringsparameterar för CF såsom leveranshastighet och blockstorlek. CF är resterande CAN meddelanden som behövs för att kunna återskapa datafältets innehåll. Antalet CF som skickas i ett block bestäms av FCF och när alla CF i blocket har skickats kommer en ny FCF skickas med nya förhållningsregler. Denna flödeskontroll har stöd för upp till 4095 byte stora dataöverföringar. Vid funktionell adressering kommer dock FF, CF och FCF alltid ignoreras av noderna [14, 21, 23, 26].

3.4 UDS över CAN

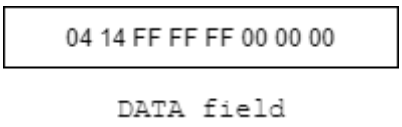
För att beskriva hur UDS är implementerat över CAN-bussen anges ett exempel som visar vilken information som läggs till på de lagren i OSI modellen som UDS protokollen anger. I exemplet kommer UDS förfrågan med UDS-tjänsten *ClearDTCInformation* 0x14 anges och datat kommer vara 0xFFFFFF som betyder att alla UDS-servrar ska nollställa sina eventuella DTC som finns lagrade i ECU:ernas minnen. Avsändarnodens adress är 0x456 och mottagarnodens adress är 0x123 i exemplet.

På applikationslagret sker UDS förfrågan. Datat som skickas är SID 0x14 och dataposten 0xFFFFFF. Datat kommer att hamna i CAN meddelandets datafält. Figur 12.1 visar datat i en UDS förfrågan på applikationslagret.



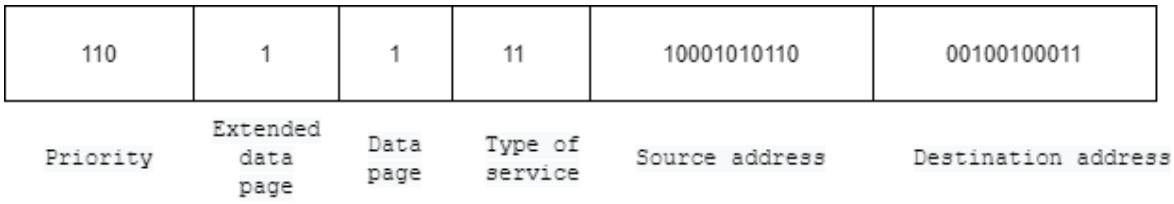
Figur 12.1. UDS förfrågan innehåller SID värdet 0x14. Varken subfunktioner eller DID värden behövs i denna UDS-tjänst. Det är datat i dataposten som anger vilka UDS-servrar som ska nollställa sina eventuella DTC:er.

På transportlagret läggs PCI till som mest signifikanta bit i CAN-meddelandets datafält. Denna information baseras på datat som kommer från applikationslagret. Om datat är större än 7 bytes kommer transportlagret sköta segmenteringen av meddelandet och skicka rätt sorts meddelandetyp vilket förklarades i kapitel 3.3.8. Figur 12.2 visar var PCI läggs till i meddelandet från applikationslagret.



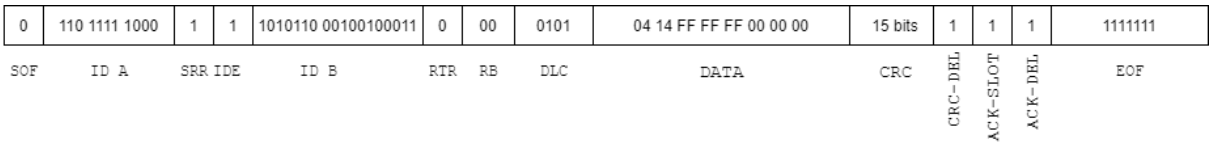
Figur 12.2. CAN meddelandets datafält är färdigt. PCI läggs till som mest signifikanta byte. I detta scenario är värdet 0x04 där 0x0 anger att det är ett SF och 0x4 anger att det är 4 byte med data. Utfyllnadsbytes med värdet 0x00 läggs till i detta scenario.

Från transportlagret skickas meddelandet till nätverkslagret där CAN ID:t ska formuleras. CAN ID:t kommer i CAN 2.0B formatet då UDS-tjänsten inte är 0x3E. CAN 2.0B formatet är dock att föredra för alla typer av UDS-tjänster, även 0x3E, men UDS protokollet gör ett undantag för 0x3E UDS-tjänsten. Figur 12.3 visar informationen som läggs till på nätverkslagret.



Figur 12.3. Mottagarnodens adress är 0x123 och avsändarnodens adress är 0x456. Prioritetsfältet ska vara 0x6 då datafältet innehåller ett UDS meddelande. Extended data page och Data page anger att meddelandet följer ISO 15765 och type of service anger att det är ISO15765-3 som ska följas.

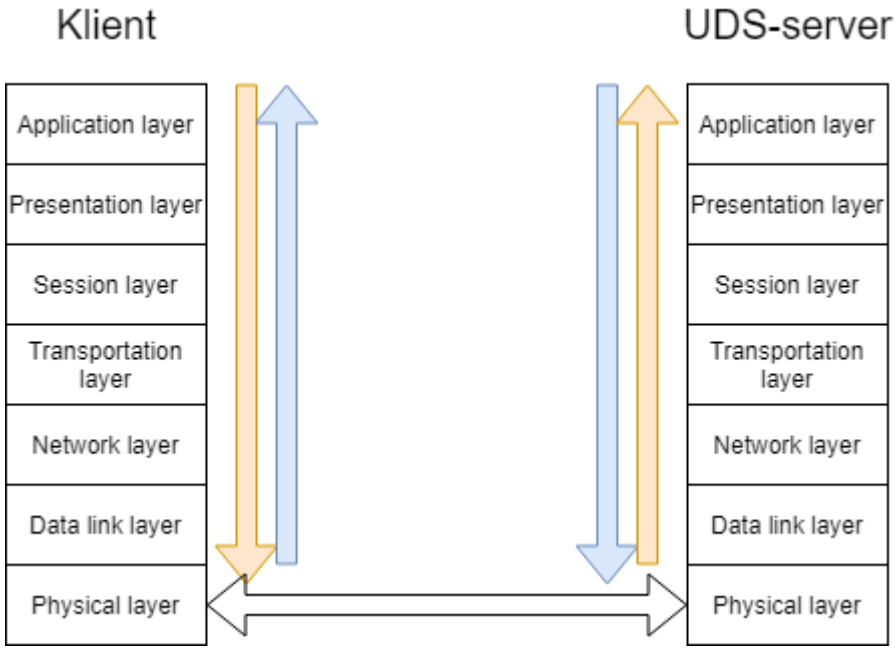
Efter nätverkslagret tar CAN protokollet ISO11898 över och bygger ihop CAN ID:t och datafältet till ett CAN meddelande. Hur det kan se ut visas i figur 12.4. CAN meddelandet skickas sedan ut på CAN-bussen.



Figur 12.4. Meddelandet som skickas från avsändarnoden via CAN-bussen. Det 29 bitars CAN ID:t från figur 12.3 delas upp i ID A som innehåller de 11 mest signifikanta bitarna och ID B som innehåller övriga bitar. Datafältet visas i hexadecimal form då det är tydligare än 64 bitar i basen 2. Värdena i fältet utöver CAN ID och datafältet

Alla noder på CAN-bussen kommer att kunna se och läsa meddelandet, men det är endast noden med den unika adressen 0x123 som behandlar CAN meddelandet. Mottagarnoden kommer ta emot meddelandet och försöka utföra UDS förfrågan på applikationsnivån. Ett svarsmeddelande formuleras där destinationens adress och avsändarens adress byter plats, ny avsändare är 0x123 och ny mottagare

är 0x456. Om UDS-servern kan ge ett positivt svarsmeddelande kommer svarsmeddelandet inneha värdet 0x54, inga andra värden skickas tillbaka till klienten. Om svaret var negativt kommer svarsmeddelandet innehålla värdena 0x7F, 0x14 och 0xXX där XX representerar möjligt NRC värde. Svarsmeddelandet kommer hanteras i ett likadant flöde som UDS förfrågan som visades i figur 12.1–12.4. Figur 13 illustrerar meddelandeflödet mellan klienten och UDS-servern.

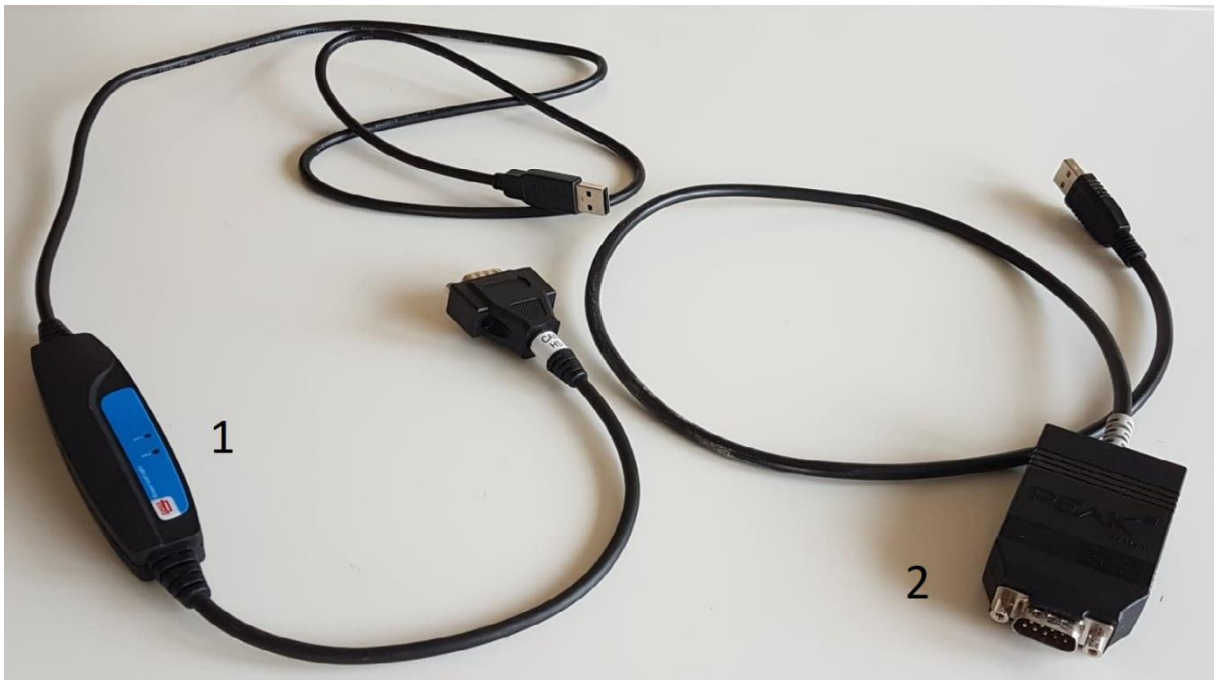


Figur 13. Flödet för klientens UDS förfrågan illustreras av de orangea pilarna. UDS förfrågan skickas från klientens applikationslager och de olika lagrena lägger till data tills CAN meddelandet innan det når det fysiska lagret, CAN-bussen. UDS-servern som har UDS-förfrågans destinationsadress kommer att ta emot meddelandet från CAN-bussen då den identifierar UDS förfrågans angivna mottagaradress som sin egen unika adress. De olika lagren kommer att ta del av den informationen från CAN meddelandet de behöver innan UDS förfrågan når UDS-serverns applikationslager. De blåa pilarna representerar flödet för UDS-serverns svar på klientens UDS förfrågan.

3.5 Interface

Kommunikation till en ECU i fordonets nätverk kräver en kommunikationsväg. För att kunna koppla upp sin dator mot fordonet behövs ett interface som är en hårdvarumodul. Det finns olika tillverkare av interfaces som Vector, Kvaser, PEAK och National Instruments med olika avancerade interfaces med priser från några tusen kronor till över 20 000 kronor. Dessa interfaces kan kommunicera med en dator via USB, Ethernet och även integreras genom kretskort och PCI-Express.

Två interface användes i detta arbete och visas i figur 14. Kvaser Leaf Light HS var det primära interfacet som den internt utvecklade mjukvaran baserades på medan PCAN-USB FD främst användes för att verifiera datat som skickads över CAN-bussen genom PCAN-VIEW som är en gratis mjukvara från PEAK. Det kan vara svårt att köpa Kvaser Leaf Light HS idag då det blivit ersatt av Kvaser Leaf Light HS v2. Kvaser säger dock att program som är skrivna för ett interface kommer kunna köras på alla andra typer av interfaces från Kvaser utan att programmet behöver ändras [24].



Figur 14. De två interfacen som användes vid utvecklingen av mjukvaran. 1) är Kvaser Leaf Light HS och 2) är PCAN-USB FD från PEAK.

3.6 Python bibliotek för fordonsdiagnostik

Kommunikation mellan mjukvaran och CAN-bussen kan utföras genom socketprogrammering på python. Det finns även bibliotek som är dedikerade för kommunikation med CAN-bussen. Python-can är ett bibliotek som stöder kommunikation med CAN-bussen. Med python-can kan de olika interface tillverkarnas egna bibliotek integreras, exempelvis har Kvaser biblioteket CANLIB och PEAK har biblioteket PCANBasic. Det finns även ett interface-oberoende bibliotek, SocketCan, som är integrerat i Linux kärna och delar resurser med internet protokollet (IP). Fördelen med detta bibliotek är att det är generellt och fungerar till de flesta interface men nackdelen är att det delar resurser med IP vilket innebär att hög internettrafik fördröjer trafiken på CAN-bussen [25]. Det finns även dedikerade python bibliotek för UDS och ISO-TP som heter udsoncan, Python-uds och can-isotp, och alla dessa bibliotek kan integreras med python-can.

4 Metod

I detta kapitel beskrivs metoderna som användes för att svara på frågeställningarna. Arbetet hade två huvudsakliga delar som använde olika metoder. Den första delen var en undersökning av protokoll och artiklar som behandlade hur UDS är implementerat över CAN och användes för att kunna svara på den första frågan i frågeställningen. Den andra delen var skapandet av en programvara som baserades på resultatet från nämnda undersökning samt testning.

4.1 Implementation

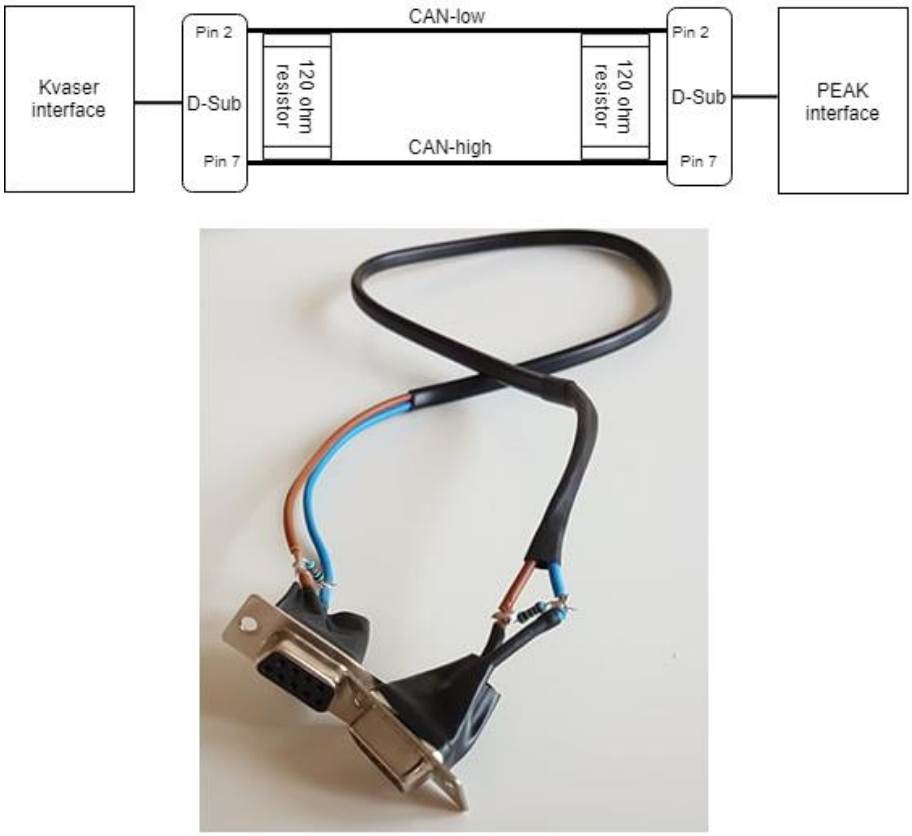
Konstruktionen av programvaran baserades på en undersökning där ISO14229 [15], ISO15765-3 [16] och Assawinjaipetch et al. [14] analyserades samt nyttjande av befintliga python bibliotek som behandlar CAN och UDS.

4.1.1 Undersökning av CAN och UDS

För att få en förståelse om hur UDS är implementerat över CAN behövde först CAN protokollet undersökas. Målet var att få en grundläggande förståelse om hur fordonsnätverket var uppbyggt. Kunskap om CAN kom främst från Voss [2] och Pazul [7], men även nätresurser som Cia [20] var till nytta. När grundläggande kunskap om CAN erhållits undersöktes ISO14229 [15] där kapitel 6, 7, 8 och 10 var av stor nytta för att förstå hur UDS ska implementeras på applikationslagret. ISO15765-3 [16] undersöktes för att få en förståelse över hur UDS är implementerat över nätverkslagret och Assawinjaipetch et al. [14] beskrev hur UDS är implementerat på transportlagret. Informationen från dessa källor sammanfogades till en helhetsbild som beskrevs i kapitel 3.4.

4.1.2 Hårdvarukonfiguration

Den första fasen av programvaruutvecklingen gjordes via två interface inkopplade mellan två datorer. För att kunna kontrollera att data kunde skickas och tas emot korrekt behövde en CAN-buss konstrueras där Kvaser interfacet var en nod och PEAK interfacet var den andra noden på CAN-bussen. Målet var att kunna skicka data mellan noderna på CAN-bussen. För att kunna utföra detta behövde en CAN-buss lödas ihop med två 9 pinnars D-Sub kontakter och två 120 ohms motstånd i kablarnas ändar. Figur 14 visar schemat och slutprodukten.

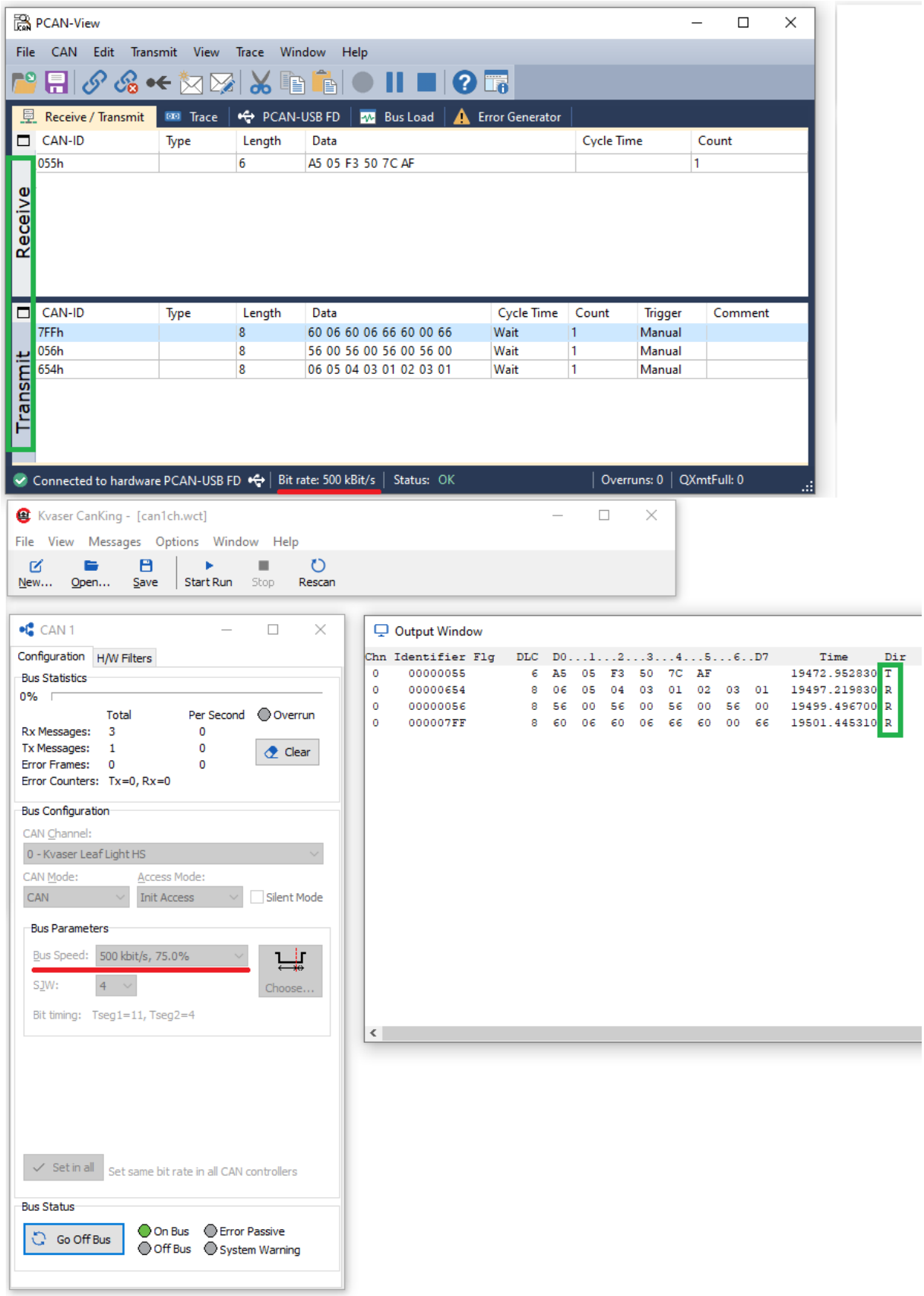


Figur 14. Överst på bilden syns schemat som användes för att kunna skapa en enkel CAN-buss. På D-Sub kontakten är pinne 2 CAN-low signalen och pinne 7 är CAN-high signalen, övriga pinnar behövde inte någon signal. Den nedre bilden visar kontakten som användes som CAN-buss. Fler noder kan kopplas upp på bussen mellan de båda 120 ohms motstånden, det viktiga är att 120 ohms motstånden finns i kabelns ände. Interfacen kopplades upp mellan var sin dator och var sin D-sub kontakt.

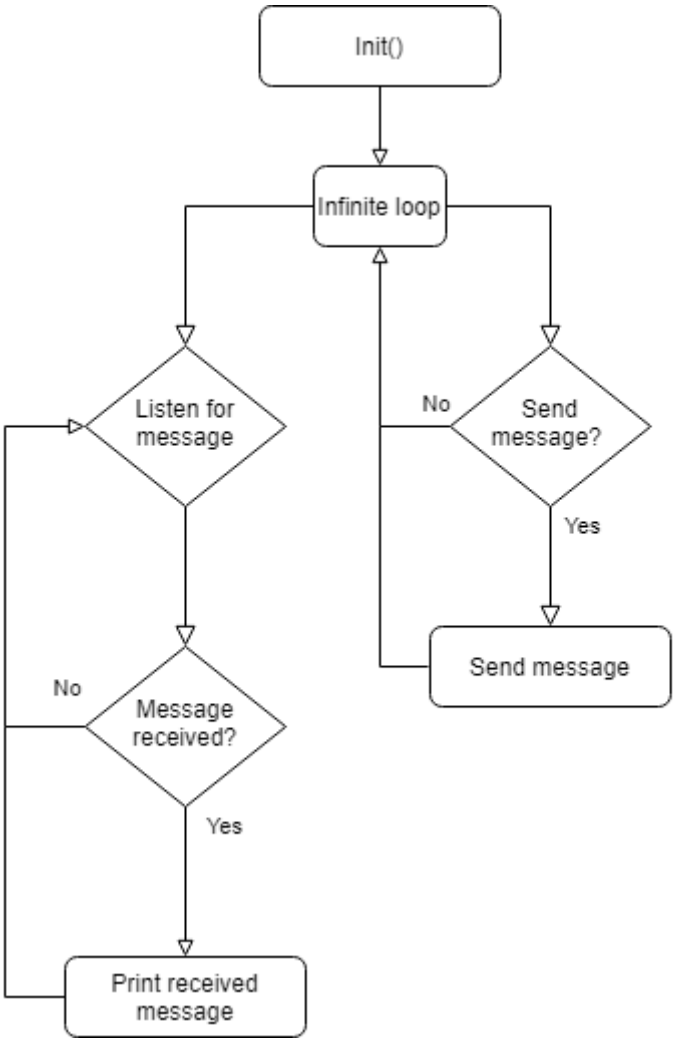
4.1.3 Programvarukonstruktion

Innan skapandet av programvaran påbörjades testades kommunikationen mellan interfacen genom programvara från tillverkarna av de båda interfacen. Kvaser har CANKING och PEAK har PCAN-VIEW och båda programmen hanterar vanliga CAN meddelanden. CAN meddelanden skickades mellan interfacen för att kontrollera att hårdvaran kunde kommunicera med varandra. Det var viktigt att busshastigheten var likadan på båda interfacen för att de skulle kunna kommunicera med varandra. Figur 15 visar de två programmen från Kvaser och PEAK.

När kommunikationen mellan interfacen kunde bekräftas skapades ett simpelt testprogram, baserat på python-can, för Kvaser interfacet som skulle ersätta CANKING. Detta för att verifiera att en korrekt busskonfiguration för Kvasers interface kunde skapas. Ett förenklat flödesschema för testprogrammet visas i figur 16.



Figur 15. PCAN-VIEW är det övre programmet och CANKING är det nedre. De två röda markeringarna visar att bussen var tvungen att ha samma baud rate för att kunna kommunicera med varandra. De gröna markeringarna belyser riktningen som meddelanden skickas på. PCAN-VIEW har separata rutor för vilken riktning meddelandena har medan CANKING samlar alla meddelanden i samma ruta och anger direktion genom T = transmit och R = receive.

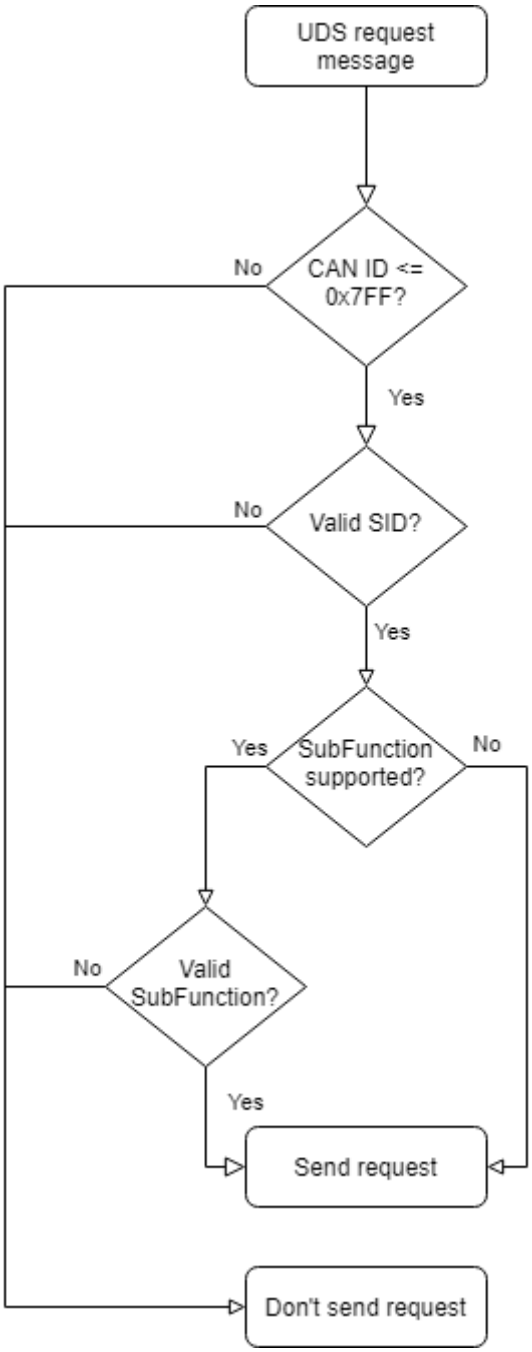


Figur 16. Följande funktionalitet implementerades tillsammans med ett användargränssnitt för att testa hur bussen skulle konfigureras samt undersöka hur CAN meddelanden skickas och tas emot. Listen-For-Message är ett blockerande funktionsanrop och kördes därmed parallellt i en bakgrundstråd. Huvudtråden var eventbaserad och kontrollerade om eventet ”Send message” aktiverats genom knapptryck. Om eventet var aktiverat så skickades ett meddelande med data inhämtat från användargränssnittet.

När testprogrammet kunde ersätta CANKING som kommunikationsväg till PCAN-VIEW kasserades all kod utöver användargränssnittet och busskonfigurationen från testprogrammet. Anledningen till att övrig kod kasserades var att biblioteket udsoncan hade egna funktioner för att konstruera CAN meddelanden och tidigare kod ansågs vara överflödig och onödigt komplicerad för att skicka UDS förfrågningar jämfört med udsoncans funktioner. Innan ny kod började skrivas undersöktes udsoncan biblioteket noggrant genom att läsa dokumentationen[3] om olika funktioner samt hur udsoncan biblioteket skulle integreras med can-isotp biblioteket.

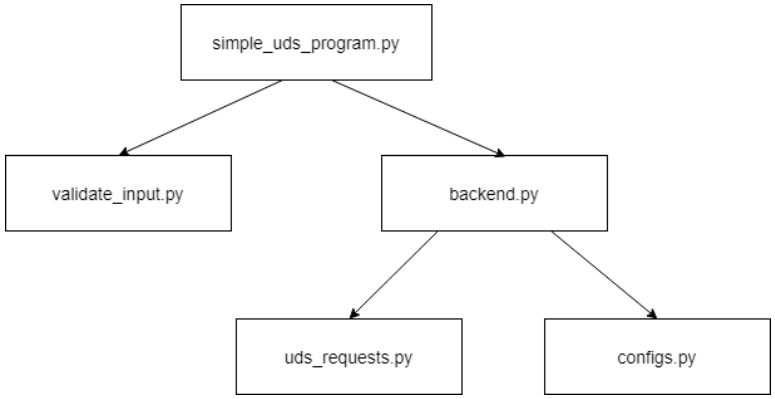
När dokumentationen från udsoncan undersökts påbörjades skapandet av programvaran. Undersökningen av CAN visade att det kan vara mycket trafik på CAN-bussen, så det är viktigt att undvika onödiga meddelanden över CAN-bussen som ändå inte kommer kunna genomföras. Figur 17 visar en enkel kontroll som undviker onödig trafik över CAN-bussen.

[3] https://udsoncan.readthedocs.io/en/latest/index.html



Figur 17. När UDS förfrågan är färdigformulerad kan en enkel kontroll göras innan förfrågan skickas. CAN ID kan vara högst 11 bitar vilket ger ett maximalt värde på 0x7FF. CAN ID högre än detta får inte finnas enligt UDS protokollet och om användaren anger ett CAN ID högre än detta ska UDS förfrågan förkastas. UDS protokollet har en lista med giltiga SID värden, om angivet SID värde inte finns i listan kommer förfrågan förkastas. Om SID var giltig enligt UDS protokollet och implementerad i programvaran kommer även den eventuella subfunktionen kontrolleras. Om UDS-tjänsten inte stöder subfunktioner är kontrollen klar och UDS förfrågan kommer att skickas. Om UDS-tjänsten stöder subfunktioner kontrolleras värdet och om det är giltigt skickas meddelandet, om inte förkastas meddelandet. Implementationen av UDS-förfrågan i udsoncan biblioteket kontrollerar själv om det angivna DID värdet finns med i den angivna konfigurationsfilen, om DID värdet saknas skickas inte UDS-förfrågan. På liknande sätt kontrollerar udsoncan även datat i dataposten. Vissa tjänster har ett maxvärde enligt UDS protokollet för dataposten, exempelvis kan tjänsten 0x14 ha ett datapostvärde i intervallet 0x000000–0xFFFFFF, om en användare anger värdet 0x01000000 i dataposten kommer udsoncan själv vägra skicka UDS förfrågan då datat i dataposten är ogiltigt.

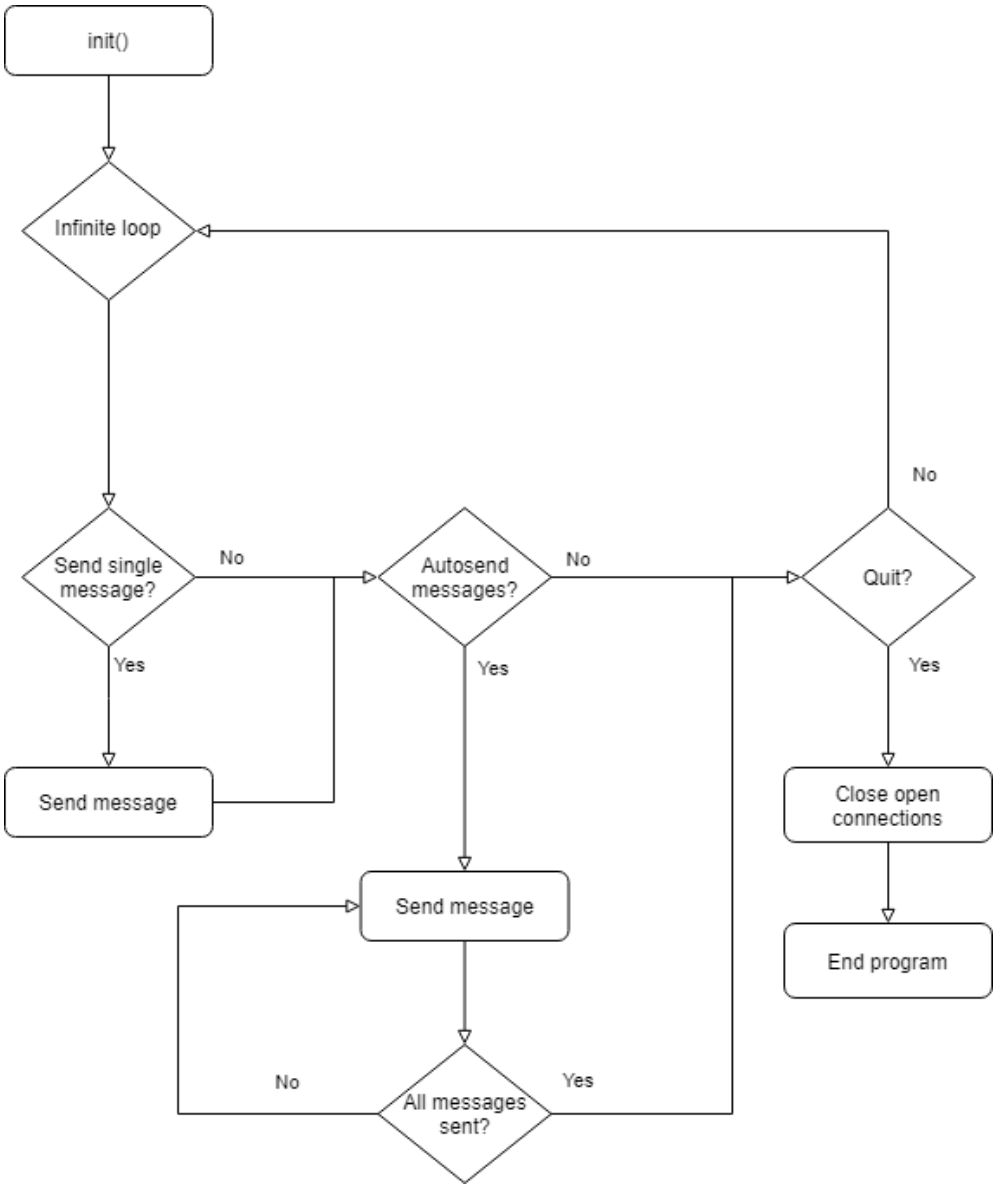
Fem filer skapades för att bygga programmet. Simple_uds_program.py var huvudprogrammet som skapade användargränssnittet och sedan körde en oändlig loop som interagerade med användaren. För att verifiera att indatat var korrekt kallade huvudprogrammet på validate_input.py som kontrollerade att indatat var hexadecimalt och inom tillåtna gränsvärden. Backend.py utförde den huvudsakliga logiken, funktionaliteten samt initieringen av de olika lagren i OSI-modellen. Backend.py kallade i sin tur på två filer, configs.py som innehöll fordonskonfigurationen som behövdes för att kunna skicka och tolka DID och datapostens data medan uds_requests.py innehöll funktionsanrop till UDS-tjänsterna. Figur 18 illustrerar hierarkin för programmet. Som grafiskt användargränssnitt användes PySimpleGUI då dokumentationen var utförlig och inlärningsperioden tordes vara kort.



Figur 18. Ett träd över programmets hierarki. Funktionsanrop går från toppen av trädet och ner ett steg medan returnerade värden går upp ett steg i trädet. Exempelvis kan inte simple_uds_program.py direkt kalla på funktioner i uds_requests.py utan simple_uds_program.py måste kalla på backend.py som i sin tur kallar på funktioner i uds_requests.py. backend.py kommer sedan modifiera returvärdet från uds_requests.py innan den returnerar det efterfrågade värdet till simple_uds_program.py.

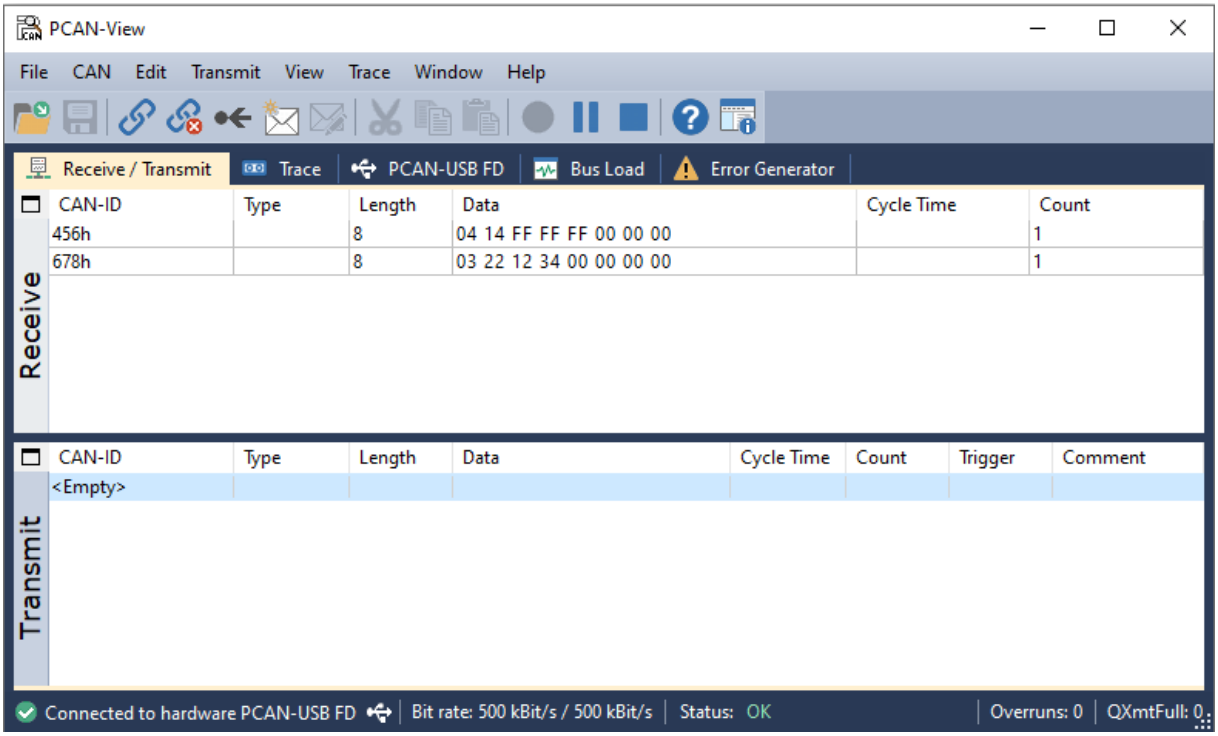
Figur 19 visar logiken för huvudprogrammets eventbaserade oändliga loop. Då UDS protokollet är baserat på förfrågan–svar dynamiken som visades i figur 5 så fanns det ingen vinning i att ha en bakgrundstråd som lyssnade efter meddelanden på CAN-bussen. UDS förfrågan genom udsoncan biblioteket kunde inte heller användas med en bakgrundstråd som lyssnar på meddelanden då förfrågningarna gjordes genom ett funktionsanrop som returnerade ett svar, och om inget svar kommit inom en i förväg definierad tidsperiod slutade funktionsanropet att lyssna efter ett svar på CAN-bussen och ett timeout undantag returnerades. Därmed kontrollerade den oändliga loopen endast tre event:

1) *Send single message* hämtade in datat som användaren angivit i användargränssnittets olika datafält och skickade en UDS förfrågan enligt det inhämtade datat.
2) *Autosend messages* läste in mottagarnodens adress och läste sedan in testfall från en fil och körde alla testfallen från filen.
3) *Quit* stängde ner öppna anslutningar och stängde sedan av programmet.



Figur 19. Huvudprogrammets oändliga eventbaserade loop. I varje iteration kontrollerar loopen om något nytt event skett (genom knapptryckning i användargränssnittet) och utför den eventuellt efterfrågade funktionaliteten. Vid eventet "Send single message" och "Autosend messages" följdes kontrollen som visades i figur 17

Programvaruutvecklingen utfördes med Kvaser och PEAK interfacet inkopplade i var sin dator. PEAK interfacet användes som mottagarnod och körde programmet PCAN-VIEW för att lyssna på trafiken över CAN-bussen medan den egenskapade programvaran använde Kvaser interfacet. Med denna utvecklingsmetod kunde bara UDS förfrågan analyseras. PCAN-VIEW är inte en ECU med UDS protokollet implementerat och kunde därmed inte returnera några svar, däremot kunde det användas för att kontrollera att UDS förfrågan såg korrekt ut teoretiskt. Resultatet för två UDS förfrågningar uppfångade av PCAN-VIEW visas i figur 20.



Figur 20. Den första UDS förfrågan hade mottagaradressen 0x456, UDS-tjänsten 0x14 och dataposten 0xFFFFFF. Den andra UDS förfrågan hade mottagaradressen 0x678, UDS-tjänsten 0x22 och DID 0x1234. Båda UDS förfrågningarna skickades från klienten, genom transportlagret där PCI lades till som mest signifikanta byte och sedan vidare ut på CAN-bussen. PCAN-VIEW visade allt data i CAN meddelandets datafält och hade inte processat datat genom ett transportlager med ISO-TP implementerat därmed visades även PCI. PCI för det första meddelandet var 0x04 där 0x0 angav att det är ett SF och 0x4 angav att UDS förfrågans storlek var 4 byte. PCI för den andra UDS förfrågan var 0x03 där 0x0 angav att det är ett SF samt 0x3 som angav UDS förfrågans storlek var 3 byte. Teoretiskt sett såg allt korrekt ut.

När PCAN-VIEW kunde verifiera att korrekt data togs emot kunde programmet testas mot en bil. Uppkopplingen till bil skedde genom en dator kopplad till ett interface som var kopplad till en ECU genom specialdesignat kablage för att komma åt CAN-bussen. Det specialdesignade kablaget ger ingen funktionell skillnad jämtemot att koppla upp interfacet till CAN-bussen genom OBD II uttaget i bilen. Under denna del av programvaruutvecklingen kunde även responsen från UDS förfrågan utvärderas. För att kontrollera vilket data och vilka eventuella fel som uppkom användes funktionen setup_logging() från udsoncan som skrev ut anslutningens status, vilket data som skickades och vilken respons som mottogs i form av bytes. Med dessa bytes från responsen kunde konfigurationsfilen configs.py färdigställas. Konfigurationsfilen behövde veta hur den mottagna responsen skulle avkodas till ett python objekt. Detta är något som udsoncan biblioteket inte gör utan resultaten från UDS förfrågan måste avkodas själv genom olika avkodningsklasser som var beroende på hur stor dataposten förväntades bli och hur de returnerade byten skulle tolkas. Avkodningsklasserna behövde tre funktioner, *encode*, *decode* och *length* där *length* inte kunde sättas dynamiskt. Den förväntade storleken på responsen från de olika DID värdena fanns dock med i fordonstillverkarens konfigurationsfil. Konfigurationsfilen innehöll även information om huruvida de returnerade byten skulle tolkas som en ascii-sträng eller som hexadecimala värden.

4.2 Utvärdering

För att kunna svara på den andra frågan i frågeställningen behövde den egenskapade programvaran jämföras mot referenstiderna från CarDiagnosticsProgram123. Referenstiderna togs tillsammans med Vector VN1610 interfacet som används tillsammans med CarDiagnosticsProgram123. För att ta fram

referenstiderna utfördes samma testsekvens som den egenskapade programvaran skulle utföra, men de snabbaste tiderna som uppkom i de olika delmomenten användes som referenstid. De två hårdvaru- och mjukvarukombinationerna (HV/MV kombination) som testades var den egenskapade programvaran med PCAN-USB FD interfacet och den egenskapade programvaran med Kvaser Leaf Light HS interfacet. Syftet med att utföra testningen med båda kombinationerna var att undersöka hur stor tidsskillnad det blir med olika interfaces då den egenskapade mjukvaran inte har stöd för Vectors interface och därmed inte kan ha samma hårdvara. De olika interfacen kan påverka resultatet och det är viktigt att fastslå hur stor påverkan det kan ha.

Utvärderingen bestod av 10 testomgångar med varje HV/MV kombination där varje testomgång utfördes på följande sätt:

1) Ett tidtagarur startas när programmet startas.
2) En enskild UDS förfrågan skickas. Det ska vara samma förfrågan för alla tre HW/MV kombinationer i samma testomgång. Värdena kan dock skilja sig åt mellan varje testomgång.
3) När 2) fått sitt svar, kör igenom den fördefinierade listan med x antal UDS förfrågningar.
4) När 3) fått sina svar, skicka en enskilt UDS förfrågan som i 2).
5) När 4) fått sitt svar, stoppa tidtagaruret.

För del 2), 3), och 4) kommer deltider tas för att få en mer detaljerad bild över var den eventuella tidsvinsten kan finnas. Tabell 1 visar bedömningsmallen som användes vid varje testomgång. Tidtagaruret som användes var Catiga CG-503.

HV/MV kombination: _____		Omgång: _____
Event:	Tid vid event slut:	UDS förfrågans värde
Första enskilda UDS förfrågan	_____s	
Kör testfallen från filen automatiskt	_____s	UDS förfrågan på den fördefinierade textfilen.
Andra enskilda UDS förfrågan	_____s	

Tabell 1: Bedömningsmallen som användes vid varje testomgång. UDS förfrågans värde fälten ska vara detsamma för alla tre HV/MV kombinationerna

Alla testen utfördes på samma bil. Bilen som testades var hemligstämplad då den vid testtillfället inte hade släppts på marknaden. Testen utfördes när bilen stod still och alla HV/MV kombinationer kopplades upp på samma accesspunkt i bilen vid sina testsessioner. Figur 21 visar hur interfacet kopplades mellan en dator och direkt till ECU:n som skulle testas. Kopplingen till ECU:n genomfördes genom det specialdesignade kablaget för att kunna komma åt CAN-bussen. I kapitel 4.1 nämndes det att det specialdesignade kablaget inte hade någon funktionell skillnad jämtemot att koppla upp interfacet på CAN-bussen genom OBD II uttaget i bilen. Testningen utfördes av en av Syntronics testare som hade gedigen erfarenhet av CarDiagnosticsProgram123. Testaren fick en kort genomgång av hur den egenskapade programvaran fungerade innan testerna genomfördes.

Figur 21. Interfacets USB kabel kopplades till datorn och 9 pinnars DB9 kontakten kopplades till det specialdesignade kablaget.

Efter att alla testomgångar var färdiga sammanställdes referenstiden genom att ta den snabbaste tiden för delmomenten från de olika testomgångarna. Sedan besvarades en kort utvärdering där testaren jämförde programvarorna och angav vad som var positivt och negativt med de båda. Frågorna som besvarades var:

1) Vilken funktionalitet från CarDiagnosticsProgram123 saknade du och skulle göra den egenskapade programvaran bättre?
2) Vad var bättre med den egenskapade programvaran jämtemot CarDiagnosticsProgram123?

5 Resultat

I detta kapitel presenteras den resulterande programvaran och dess prestanda i form av tid jämtemot företagets nuvarande lösning med CarDiagnosticsProgram123.

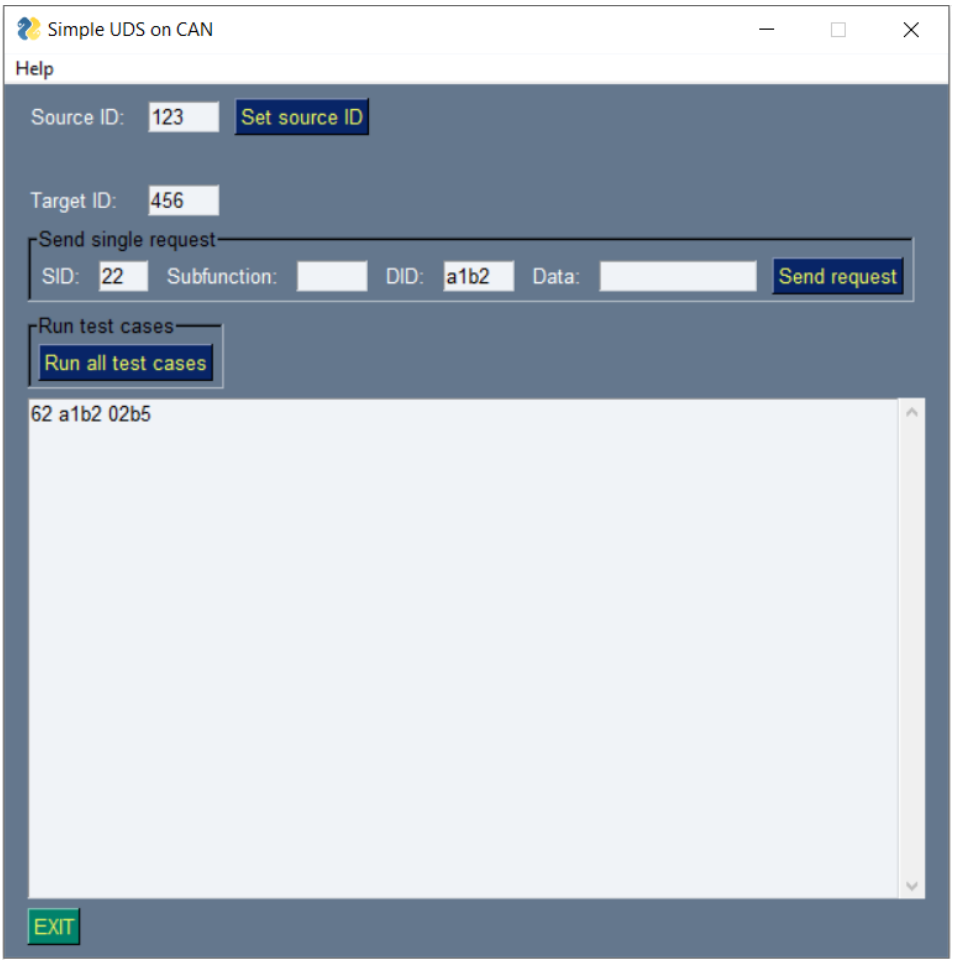
5.1 Implementation

Programvaruimplementationen baserades på tre python bibliotek, python-can, can-isotp och udsoncan. Alternativ till python-can och udsoncan fanns men dessa valdes av följande anledningar:

- python-can är ett generellt CAN bibliotek. Alternativet hade varit att använda interfacetillverkarnas egna bibliotek, men då hade programvaran inte kunnat bli så pass generell att interface från olika tillverkare kunde användas. De olika interfacetillverkarnas bibliotek behövdes dock för att kunna skapa bussanslutningen, men det var det ända de biblioteken användes till.
- För att implementera UDS fanns det två färdiga bibliotek, Python-uds och udsoncan. Valet föll på udsoncan då det biblioteket var mer välutvecklat och hade fler UDS-tjänster implementerat. Detta ansågs vara positivt då det möjliggjorde en större vidareutvecklingspotential för programvaran.

Figur 22 visar programvarans grafiska användargränssnitt. Funktionaliteten som implementerades var följande:

- Avsändarnodens adress (”Source ID”) sattes med ett knapptryck för att visa att värdet bara behövde sättas en gång då avsändarnodens adress bara behöver sättas en gång under programmets aktiva session. Om interfacet ska kopplas till en ny accesspunkt bör programmet stängas av så att alla öppna anslutningar stängs av korrekt och sedan startas igen. Det går dock att byta avsändarnodens adress under programmets körtid.
- Mottagarnodens adress (”Target ID”) hämtades in vid varje enskild UDS förfrågan och en gång vid varje tillfälle som alla testfall skulle köras.
- För att illustrera de två olika möjligheterna och vilket data som behövde anges för att skicka UDS förfrågningar sattes de in i olika ramar. ”Send single request” läste in värdena ”SID”, ”Subfunction”, ”DID” och ”Data” utöver ”Target ID”) medan ”Run all test cases” bara läste in ”Target ID”. ”Target ID” som användes av båda möjligheterna angavs i ett fält utanför båda ramarna.
- Svaret från UDS förfrågan skrevs ut på den vita utskriftsrutan.
- UDS förfrågans svar skrevs till en textfil direkt efter att svaret skrevs ut på utskriftsrutan.
- Under utskriftsrutan fanns ”Exit” knappen som stängde ner alla öppna anslutningar och sedan avslutade programmet.
- Hjälpmenyn (”Help”) öppnade ett fönster med information om programmet.



Figur 22. DID värdena och det returnerade datat är hemligstämplat, så detta är en UDS förfrågan och ett svar med fabricerade värden. En UDS förfrågan skickades med avsändaradressen 0x123 och mottagarnodens adress var 0x456. En enskild UDS förfrågan skickades där vald UDS-tjänst var 0x22 med DID värdet 0xA1B2. Då UDS-tjänsten inte stöder subfunktioner eller dataposter lämnades fälten tomma. Svaret från UDS förfrågan hamnade i den vita utskriftsrutan samt på en textfil. Svaret var positivt då värdet 0x62 returnerades. Nästkommande två bytes var en kopia på UDS förfrågans DID värde. Resterande bytes var det returnerade datat från ECU:n. I denna UDS förfrågan returnerades 0x02B5.

5.2 Utvärdering

Referenstiden som skulle slås visas i tabell 2 och var den snabbaste tiden som CarDiagnosticsProgram123 presterade. Även de snabbaste delmomentstiderna visas i tabell 2 och identifierar vart eventuell tidsvinst kan finnas. De resulterande tiderna för den HV/MV kombinationerna presenteras både som genomsnittlig tid och som en mediantid för de tre olika HV/MV kombinationerna i tabell 3 och 4

CarDiagnosticsProgram123	Snabbast tid (s)	Från omgång (nr)	Tidsskillnad från föregående tidsstämpel. (s)
Första enskilda UDS förfrågan	27.5	9	-
Kör testfallen från filen automatiskt	40	8, 9 och 10	12.5
Andra enskilda UDS förfrågan	47	8	7
Referenstid	47	8	-

Tabell 2: referenstiderna från CarDiagnosticsProgram123. Detta är de snabbaste tiderna från testomgångarna.

PCAN-USB FD och egen programvara kombinationen	Genomsnittlig tid (s)	Tidsskillnad från föregående tidsstämpel. Genomsnittlig tid (s)	Mediantid (s)	Tidsskillnad från föregående tidsstämpel. Mediantid (s)
Första enskilda UDS förfrågan	16.2	-	15	-
Kör testfallen från filen automatiskt	29	12.8	28	13
Andra enskilda UDS förfrågan	35.7	6.7	34.5	6.5

Tabell 3: Genomsnittlig tid och median tiden för PCAN-USB FD och den egenskapade programvarukombinationen samt tider för delmomenten.

Kvaser Leaf Light HS och egen programvara kombinationen	Genomsnittlig tid (s)	Tidsskillnad från föregående tidsstämpel. Genomsnittlig tid (s)	Mediantid (s)	Tidsskillnad från föregående tidsstämpel. Mediantid (s)
Första enskilda UDS förfrågan	15.8	-	15.5	-
Kör testfallen från filen automatiskt	26.3	10.5	26	10.5
Andra enskilda UDS förfrågan	32.8	6.5	34	8

Tabell 3: Genomsnittlig tid och median tiden för PCAN-USB FD och den egenskapade programvarukombinationen samt tider för delmomenten.

Den egenskapade programvaran var snabbare än referenstiden oavsett om PEAKs eller Kvasers interface användes. Då 10 testomgångar inte är många är mediantiden mest representativ som jämförelsetid, och följande observation kan fås från tabellerna 2 – 4.

- PCAN-USB FD med den egenskapade programvaran var 12.5 sekunder snabbare än referenstiden.
- Kvaser Leaf Light HS var 13 sekunder snabbare än referenstiden.
- CarDiagnosticsProgram123 en halv sekund snabbare mellan delmomenten ”Första enskilda UDS förfrågan” och ”Kör testfallen från filen automatiskt” jämfört med PCAN-USB FD
- CarDiagnosticsProgram123 var en sekund snabbare mellan delmomenten ”Kör testfallen från filen automatiskt” och ”Andra enskilda UDS förfrågan” jämfört med Kvaser Leaf Light HS
- PCAN-USB FD var en halv sekund snabbare mellan delmomenten Kör testfallen från filen automatiskt” och ”Andra enskilda UDS förfrågan” jämfört med CarDiagnosticsProgram123.
- Kvaser Leaf Light HS var två sekunder snabbare mellan delmomenten ”Första enskilda UDS förfrågan” och ”Kör testfallen från filen automatiskt” jämfört med CarDiagnosticsProgram123.

Vid utvärderingen från testaren angående programvarorna framkom följande svar på frågorna:

1) Vilken funktionalitet från CarDiagnosticsProgram123 saknade du i den egenskapade programvaran?
   Testaren ville ha möjlighet att själv namnge loggfilen när programmet var aktivt. Testaren ville även ha möjlighet att välja vilken fil med testfall som skulle köras. I CarDiagnosticsProgram123 behövde varken avsändarnodens eller mottagarnodens adress anges och testaren saknade detta i den egenskapade programvaran.

2) Vad var bättre med den egenskapade programvaran jämtemot CarDiagnosticsProgram123?
   Både uppstart och nedstängning av den egenskapade programvaran var betydligt snabbare än CarDiagnosticsProgram123 vilket ansågs vara positivt. Testaren uppskattade även att funktionaliteten för att sända en enskild UDS förfrågan och att köra listan med testfallen låg nära varandra på det grafiska användargränssnittet. Det var även bra att loggfilen skapades automatiskt och sparade ner datat, men att kunna namnge filen själv istället för default namnet som skapades hade gjort det bättre.

6 Diskussion

I detta kapitel kommer resultatet, metoden och arbetet i ett vidare sammanhang diskuteras.

6.1 Resultat

Att den egenskapade programvaran var snabbare än CarDiagnosticsProgram123 var föga överraskande. Detta då CarDiagnosticsProgram123 hade många mer funktioner som kunde konfigureras och det tog i genomsnitt nära 30 sekunder innan alla nödvändiga filer hade laddats in och programmet kunde användas. CarDiagnosticsProgram123 är en mer komplett programvara med många mer funktioner, men om man bara ska utföra lättare fordonsdiagnostik är programvaran för avancerad vilket medför viss overhead tid. Genom att konfigurera den egenskapade programvaran vid uppstart utan användarinteraktion utöver att sätta avsändarnodens adress så kunde ungefär 15 sekunder sparas in. Jag hade däremot inte räknat med att den egenskapade programvaran även skulle vara snabbare i olika delmoment. CarDiagnosticsProgram123 utförde UDS förfrågningarna och fick svaren snabbare men det krävdes mer interaktion då användaren behövde gå igenom fler menyer för att kunna utföra olika funktioner. Det var här tidsvinsten fanns. Exempelvis tog det 9–14 sekunder att köra listan med x antal UDS förfrågningar och få svar med den egenskapade programvaran medan CarDiagnosticsProgram123 kunde utföra de på 2–3 sekunder, trots detta blev den totala tiden snabbare eller lika snabb. Även detta berodde på att CarDiagnosticsProgram123 krävde att användaren gick mellan menyer för att kunna köra listan med de x antal UDS förfrågningarna medan den egenskapade programvaran kunde utföra det med ett knapptryck. I detta scenario var dock tidsjämförelsen möjligtvis orättvis. Detta då CarDiagnosticsProgram123 tillät användaren att välja vilken fil med testfall som skulle köras medan den egenskapade programvaran bara kunde köra den testfilen. Det finns dock bara en testfil, men om flera olika testfiler skapas framöver måste den egenskapade programvaran också låta användaren välja vilken testfil som ska köras, och då är CarDiagnosticsProgram123 snabbare då den kan köra testfallen på en femtedel av tiden det tar för den egenskapade programvaran.

Utvärderingen av testaren gav mer funktionsspecifika svar gällande hur välfungerande den egenskapade programvaran presterade jämfört med CarDiagnosticsProgram123. Ett av de största problemen med den egenskapade programvaran var att avsändarnodens adress och mottagarnodens adress behövde anges manuellt. Detta är något som CarDiagnosticsProgram123 inte krävde, utan dessa värden var redan satta. Testaren visste inte vart avsändarnodens och mottagarnodens adress kunde hittas utan att använda CarDiagnosticsProgram123 tillsammans med CANalyzer[4] som är ett program från Vector som hämtar in all data, inklusive adresserna för de olika noderna, från CAN-bussen. Det är möjligt att dessa adresser finns med i konfigurationsfilerna och det är även möjligt att dessa adresser är konstanta på alla bilar inom fordonstillverkarkoncernen. Den sistnämnda tesen har inte kunnat testas då programmet endast testades i en bil. Ett förslag från testaren var att det egenskapade programmet skulle spara avsändarnodens och mottagarnodens adress när programmet stängdes av och sedan laddas in automatiskt när programmet startades. Detta är något som kan implementeras relativt enkelt, men det löser inte problemet med att hitta rätt adresser första gången programmet ska köras i en ny bil.

En annan funktionalitet med förbättringspotential med den egenskapade programvaran var att låta användaren ange filnamnet själv. Den egenskapade programvaran skapade filnamnet automatiskt genom modifiering av ett DateTime object i python som ger det nuvarande datumet och tiden. Detta ansågs ge unika filnamn då tiden gavs med en precision i mikrosekunder, och chansen att två användare skulle lyckas skapa en fil exakt samtidigt ansågs vara låg. Dessa filnamn sorterades även kronologiskt i mappen som sparade loggfilerna vilket gjorde det enkelt att hitta den senaste loggfilen. Men testaren föredrog

[4] https://www.vector.com/int/en/products/products-a-z/software/canalyzer/#c652

att få möjligheten att namnge filen själv och att nuvarande filnamn med det modifierade DateTime objektet kunde få vara default filnamnet om inget eget filnamn angivits.

Fördelen med den egenskapade programvaran var att den var snabbare och enklare att arbeta med. En stor del i detta låg i att funktionaliteten för att skicka en enskild UDS förfrågan samt att köra listan med de fördefinierade UDS förfrågningarna låg bredvid varandra och man behövde inte navigera mellan menyer för att byta mellan dem.

En aspekt som inte togs med i utvärderingen men som uppskattades av testaren var tiden det tog att stänga ner programmet. Med CarDiagnosticsProgram123 behövde användaren aktivt spara filen om denne ville behålla datat och sedan stänga ner programmet. Även nedstängningen hade viss interaktion utöver knapptrycket för att stänga ner programmet vilket innebar ytterligare några sekunders körtid. Den egenskapade programvaran sparade istället ner resultatet på en textfil under programmets körtid och vid avslut stängdes programmet av omgående. Anledningen till att denna aspekt inte inkluderades i testningen var att denna tid var en så pass liten del av den totala körtiden för programmet att det ansågs vara försumbart.

6.2 Metod

Arbetet började med två veckors undersökning av hur CAN och UDS fungerar. Information om CAN fanns i flera artiklar och böcker. Variationen av pålitliga källor som beskrev CAN gjorde det enkelt att hitta texter som beskrev CAN på ett pedagogiskt sätt och jag kunde snabbt få en grundläggande förståelse för CAN. UDS är däremot ett mer nischat område och en överblicksartikel eller bok om UDS hittades inte. Artiklarna jag hittade som behandlade UDS beskrev det kortfattad med stora kunskapsluckor för en nybörjare gällande UDS. Valet föll på att börja läsa UDS protokollet ISO14229 [15] vilket, med facit i hand, inte är att rekommendera om man inte har en grundläggande kunskap om UDS. Icke vetenskapliga källor som PiEmbSysTech [27], en blogg med fokus på inbyggda system, beskrev UDS mer pedagogiskt, och ett tidigt fokus på en sådan källa hade gett en bättre överblick över UDS och därmed underlättat undersökningen av ISO14429 [15]. Med kunskapen från bloggen blev även protokollen och de vetenskapliga artiklarna som behandlade UDS mycket lättare att förstå och en djupare förståelse kunde hämtas från de.

Protokollen som arbetet baserades på var utdaterade. Det finns nyare protokoll som jag däremot inte hade tillgång till. ISO15765-3 [16] är från 2004 och blev ersatt av ISO14229-3 år 2012. Informationen i ISO15765-3 [16] är inte felaktig men nyare information fanns i ISO14229-3. Detsamma gällde ISO14229 [15] som är från 2006 och blev ersatt av ISO14229-1:2013 som i sin tur blivit ersatt av ISO14229-1:2020. Även i detta fall är informationen i ISO14429 [15] inte felaktig men nyare och mer aktuell information finns.

Coronakrisen medförde restriktioner för personalen på Syntronic. Utvecklarna skulle arbeta hemifrån och detta inkluderade även mig. Detta medförde viss problematik då mina potentiella lösningar inte kunde testas i en bil kontinuerligt utan majoriteten av utvecklingen skedde hemifrån mellan två interface som inte hade UDS implementerat. När en potentiell lösning fanns bokades en tid in med en utvecklare på Syntronic för att få möjlighet att testa lösningen mot en ECU i en bil. Programvaran testades mot en bil tre gånger i 1–2 timmars sessioner innan UDS-tjänsten 0x22 fungerade korrekt. Denna arbetsmetod var inte optimal och en bättre slutprodukt med fler implementerade UDS-tjänster hade varit möjlig med mer utvecklingstid i en bil. Med ett tidigare fokus på programvaruimplementationen hade det kunnat vara möjligt med åtminstone en till testsession i en bil och det är möjligt att ytterligarare en UDS-tjänst hade kunnat implementerats. Detta hade dock tagit tid från den teoretiska genomgången och det är inte säkert att slutprodukten hade blivit bättre.

6.2.1 Implementation

Den viktigaste UDS-tjänsten för Syntronic var *ReadDataByIdentifier (0x22)* och endast den blev implementerad. Andra UDS-tjänster var en bonus och ett försök att implementera två andra UDS-tjänster, ClearDTC (0x14) och ReadDTCInformation (0x19), gjordes men misslyckades. Med mer utveckling vid bil hade de dock kunnat implementerats.

Nackdelen med att använda udsoncan var att varje UDS förfrågan var ett blockerande funktionsanrop. Man kunde ange hur länge blockeringen skulle fortgå innan ett timeout undantag gavs, men det hade varit att föredra om UDS förfrågan skickades och sedan väntade på svaret i en ny bakgrundstråd. UDS förfrågningsmeddelanden har låg prioritet på CAN-bussen och det kan dröja någon sekund innan ett svar fås. Detta var inte något större bekymmer när en enskild UDS förfrågan skickades, men när listan med testfall som innehöll x antal UDS förfrågningar skulle skickas blev det märkbart då ingen ny UDS förfrågan kunde skickas innan den tidigare UDS förfrågan hade fått sitt svar. Att gå igenom listan med testfallen tog mellan 9 och 14 sekunder och programfönstret var fryst under denna tidsperiod. Detta försökte lösas genom att implementera en trådpool och en ”arbetarfunktion” som utförde ett antal UDS förfrågningar samtidigt (*concurrency* ej att förväxla med parallellt). Funktionaliteten testades hemma mot PCAN-View och alla UDS förfrågningar skickades korrekt på en femtedel av tiden det tog innan. Om denna implementation hade fungerat skulle det ta ungefär två sekunder att utföra alla UDS förfrågningar. Tyvärr fick jag det inte att fungera. När detta testades i bil fick 5 UDS förfrågningar av x svar, övriga UDS förfrågningar fick diverse felmeddelanden. Detta tros bero på att udsoncan släppte den tidigare förfrågans anslutning när en ny gjordes vilket resulterade i att den tidigare UDS förfrågans svar inte kunde tas emot av klienten då anslutningen redan hade avbrutits när den nya UDS förfrågan skickats.

Fördelen med att använda udsoncan var att det möjliggjorde att programmet kunde bli klart i tid. Det hade varit möjligt att skapa en egen implementation från python-can biblioteket och implementera ISOTP och UDS protokollen genom ett vanligt CAN meddelande, och implementera en bakgrundstråd som lyssnar efter svaren från UDS förfrågningarna, men det hade inte varit möjligt tidsmässigt under ett kandidatarbetes omfång.

6.2.2 Utvärdering:

Precisionen på tidtagningen var inte bra. En inbyggd timer kunde startas och stoppas automatiskt i den egenproducerade mjukvaran, men CarDiagnosticsProgram123 behövde konfigureras manuellt och en automatiskt styrd timer kunde inte implementeras. Det mest rättvisa var därmed att använda samma tidtagningsmetod för båda mjukvarorna, vilket innebar manuell tidtagning med ett tidtagarur. Tidtagarur kräver dock mänsklig reaktionstid och är inte exakt. En annan problematik med tidtagning som utvärderingsmetod är att trafiken på CAN-bussen påverkar hur snabbt testfallen en UDS förfrågan kan skickas och sedan få sitt svar. UDS meddelanden är i CAN 2.0B formatet och har låg prioritet på CAN-bussen vilket innebär att om många högre prioriterade CAN meddelanden skickas kommer UDS meddelandena att behöva vänta för att få tillgång till CAN-bussen och denna trafik kan inte användaren påverka.

Att utföra 10 testomgångar med varje HW/MV kombination kan anses vara i det minsta laget men då tidsskillnaden var stor mellan den snabbaste tiden för CarDiagnosticsProgram123 och mediantiderna från de två övriga HW/MV kombinationerna ansågs det vara tillräckligt för att kunna svara på den andra frågan i frågeställningen. Testet kunde även avgöra om det fanns någon skillnad beroende på vilket interface som användes. Skillnaderna var inte stora, men Kvaser Leaf Light HS kunde utföra de x antal UDS förfrågningarna från listan ungefär 2.5 sekunder snabbare än PCAN-USB FD. Däremot påverkades inte den totala tiden för testomgångarna nämnvärt. Detta var dock inte en av frågeställningarna utan uppkom i kapitel 4.3 när utvärderingsmiljön bestämdes.

6.3 Arbetet i vidare sammanhang

Bilar påverkar både miljö och människor. Människor dör av både direkt och indirekt påverkan från bilar som kollisioner och föroreningar. Om programvaran som skapades inte är ordentligt testad innan användning kan både miljö och människors hälsa påverkas. Om programvaran ger felaktig respons, antingen genom en felaktigt utförd UDS förfrågan eller ett svar som tolkas felaktigt kan det få ödesdigra konsekvenser för miljön, människan eller bådadera. Med en annan konfigurationsfil från en biltillverkare kan exempelvis bilens utsläpp läsas av och felaktig respons kan ge falska svar, både negativa och positiva. Detsamma gäller om programmet exempelvis används för att läsa responsen från bromsarnas funktion. Felaktiga svar kan i bästa fall orsaka ekonomiska skador och varumärkesskador för biltillverkaren om bilar behöver återkallas och i värsta fall dödsfall om bromsarna inte fungerar som de skall. I ett sådant scenario påverkas även företaget som utförde testningen negativt om det kan påvisas att testföretaget brast i sin testning.

7. Slutsats

Båda frågorna från problemformuleringen kunde besvaras lyckosamt. Svaret på den första frågan i problemformuleringen ” 1) Hur är UDS protokollet implementerat över CAN-bussen?” kunde besvaras som ett fiktivt scenario i kapitel 3.4. Den andra frågan i problemformuleringen ” 2) Hur kan en internt utvecklad mjukvara som ska utföra fordonsdiagnostik enligt UDS protokollet, genom automatiserade tester och automatisk datainsamling konstrueras så att total körtid, inklusive uppstart av mjukvaran, är snabbare än referenstiderna som sattes av CarDiagnosisProgram123?” kunde också besvaras och utförandet och slutresultatet besvarades i kapitel 4.1.3 och kapitel 5.1. I denna fråga finns det dock inget entydigt svar och den resulterande lösningen kan förbättras. Även de önskvärda kraven ”3) kunna kommunicera på CAN bussen med flera interface.”, ”4) kunna live-visualisera data som kommer från UDS kommandona.” och ”5) visualisera autosparad data.” kunde implementeras i den egenskapade programvaran.

Nyttan för Syntronic var att de fick en programvara som kunde utföra fordonsdiagnostik med en snabbare total körtid än CarDiagnosticsProgram123. Detta gällde dock bara för den konfigurationsfil från fordonstillverkaren som användes. Om en annan konfigurationsfil från en annan fordonstillverkare ska användas kan den inte bara laddas in, utan DID värdena och hur de ska avkodas måste läggas in manuellt i filen configs.py. Detta är dock inget omfattande arbete utan tar ungefär 20–30 minuter och behöver endast utföras en gång. En konfigurationsfil från en fordonstillverkare kan vara aktuell i flera år så denna overhead-tid anses vara försumbar. Om konfigurationsfilerna från fordonstillverkaren kom oftare hade detta behövt lösas med en automatiserad lösning.

Resultaten anses hålla även om utvärderingen endast utfördes på en bil. Detta då den största tidsvinsten låg under initieringen av programmet innan den första UDS förfrågan hade skickats, och denna tid är fordonsoberoende.

Litteratur

[1] W Dubitzky, T Karacay, “CAN – From its early days to CAN FD”, Can Newsletter Online, http://www.can-newsletter.org/uploads/media/raw/6b2563046de889524638725c61627661.pdf

[2] V. Voss, “A Comprahensible Guide to Controller Area Network”, Copperhill Technologies, 2005, ISBN 0-9765116-0-6

[3] S. Dekanic , R. Grbic, T. Maruna, I. Kolak, ”Integration of CAN Bus Drivers and UDS on Aurix Platform”, Published in: 2018 Zooming Innovation in Consumer Technologies Conference (ZINC), 2018

[4] M. Salcianu, C. Fosalau, “A new CAN diagnostic fault simulator based on UDS protocol”, Published in: 2012 International Conference and Exposition on Electrical and Power Engineering, 2012

[5] Kvaser, “Using termination to ensure recessive bit transformation”, https://www.kvaser.com/using-termination-ensure-recessive-bit-transmission/

[6] Y. Jinghua, L. Feng, “An Automated Testing Method for UDS Protocol Stack of Vehicles”, in Proceedings 2016 5th International Conference on Measurement, Instrumentation and Automation (IMCMIA 2016), 2016

[7] Pazul, “Controller Area Network (CAN) Basics”, Microship Technology Inc, 1999

[8] S. Godavarty, S. Broyles, M. Parten, “Interfacing to the On-Board Diagnostics System”, Published in: Vehicular Technology Conference Fall 2000 IEEE VTS Fall VTC2000. 52nd Vehicular Technology Conference (Cat. No.00CH37152), 2000

[9] S. Kelkar, R. Kamal, “Adaptive Fault Diagnosis Algorithm for Controller Area Network”, Published in: IEEE Transaction on Industrial Electronics (Volume 61, Isssue: 10, Oct. 2014), 2014

[10] Kvaser, ”CAN Bus Error Handling”, 2020, https://www.kvaser.com/about-can/the-can-protocol/can-error-handling/

[11] SE. Marx, JD. Luck, SK. Pitla, RM. Hoy, ” Comparing different hardware/software solutions and conversion methods for Controller Area Network (CAN) bus data collection”, Computers and Electronics in Agriculture (Volume 128, October 2016, Pages 141-148), 2016

[12] P. Kharche, M. Murali, G. Khot, “UDS Implementation for ECU I/O Testing”, Published in: 2018 3rd IEEE International Conference on Intelligent Transportation Engineering (ICITE), 2018

[13] M. Rings, P. Phillips, “Adding Unified Diagnostic Services over CAN to an HIL Test System”, SAE Technical Paper 2011-01-0454, 2011

[14] P. Assawinjaipetch, M. Heeg, D. Gross, S. Kowalewski, “Unified Diagnostic Services Protocol Implementation in an Engine Control Unit”, SemanticScholar, 2013

[15] ISO 14229. Road Vehicles – Unified diagnostic services (UDS) – Specifications and requirements. ISO, Geneva Switzerland 2006

[16] ISO 15765-3. Road Vehicles – Diagnostics on Controller Area Networks (CAN) – Part 3: Implementation of unified diagnostic services (UDS on CAN). ISO, Geneva Switzerland 2004

[17] Embitel, “4 UDS Protocol Software Services that Every Automotive Product Development Team Should Know”, 2018, https://www.embitel.com/blog/embedded-blog/4-uds-protocol-services-every-automotive-geek-should-know

[18] EmbededCLogic, “UDS-Remote Activation Of Routine”, https://embedclogic.com/uds-protocol/uds-remote-activation-of-routine/

[19] D. Hu, D. Hou, K. Guo, C. Sun, “Design and implementation of Diagnostic system for Integrated body Controller based on CAN bus”, Published in: 2019 Chinese Automation Congress (CAC), 2019

[20] CAN in Automation (Cia), “CAN lower- and higher-layer protocols”, https://www.can-cia.org/can-knowledge/

[21] C. Miller, C. Valasek, “Adventures in Automotive Networks and Control Units”, IOActive, Technical white paper, 2014

[22] C. Smith, “The Car Hacker’s Handbook – A Guide for the Penetration Tester”, No Starch Press US, 2016, ISBN: 9781593277031

[23] STI Innsbruck, “Vehicle Networks – CAN-based Higher Layer Protocols”, https://www.yumpu.com/en/document/read/4303401/vehicle-networks-can-based-higher-layer-protocols-sti-innsbruck

[24] Kvaser, “Kvaser Leaf Light HS v2”, https://www.kvaser.com/product/kvaser-leaf-light-hs-v2/

[25] M. Kleine-Budde, ”SocketCa – The official CAN API of the Linux kernel”. Published in: Proceedings of the 13th International CAN Conference (iCC 2012), 2012

[26] PiEmbSysTech, “CAN-TP Protocol”, https://piembsystech.com/can-tp-protocol/

[27] PeEmbSysTech, ”UDS Protocol”, https://piembsystech.com/uds-protocol/