Linköpings universitet | Institutionen för Datavetenskap
Examensarbete på grundnivå, 16hp | Datateknik
Vårtermin 2020 | LIU-IDA/LITH-EX-G--20/026-SE

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

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page:
http://www.ep.liu.se/.

© Michael Abraham

Sammanfattning

Bilar blir allt mer tekniskt avancerade och fler ECU:er har utvecklats som har medfört ökad säkerhet och komfort samt minskad miljöpåverkan. Det 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 en mycket större funktionalitet än de behöver för att utföra fordonsdiagnostik och all denna onödiga funktionalitet i programvaran har medfört onödigt långa körtider. Genom att studera CAN och UDS och genom att analysera hur de samverkar kunde jag skapa en programvara genom att systematiskt utveckla programvaran med två gränssnitt 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 anpassad för företagets behov och den mer funktionalitetsanpassade programvaran kunde utföra samma diagnostik snabbare än företagets nuvarande programvara. Den UDS-tjänst företaget använde mest kunde implementeras och den skapade programvaran konstruerades så att fler UDS-tjänster kunde läggas till utan modifikation av huvudprogrammet eller dess funktioner.

Abstract

Cars are getting more technically 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 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 unnecessarily long runtimes for the program. By studying CAN and UDS and analyzing how they interact, I was able to create a software 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 and the created software enabled more UDS services to be added without modifications of the main program or its features.

Förord

Jag skulle vilja tacka min examinator Jonas Wallgren och min handledare John Tinnerholm på Linköpings universitet för hjälp och guidning med detta kanditatarbete. Från Syntronic skulle jag vilja tacka Marcus Hall och Jonah Ekelund som varit mina handledare, Mattias Jons och Elvis Jakobsson som hjälpt till med testning vid bil och Niklas Larsson som hjälpte till med teorin till den egentillverkade CAN-bussen. Jag skulle även vilja rikta ett stort tack till min fru Therese som hjälpt till med korrekturläsning och som har varit ett fantastiskt stöd utanför studierna.

Förkortningar och begrepp

CAN – *Controller Area Network.*

CarDiagnosticsProgram123 – fiktivt namn för den konfidentiella programvaran som Syntronic AB använder för att kunna utföra fordonsdiagnostik enligt UDS.

CanDoRequests – namnet på den egenskapade programvaran.

can-isotp – Pythonbibliotek som kan utföra funktionalitet som bestäms i ISO15765-2.

CF – *Consecutive Frame.*

DID – *Diagnostic Identifier.*

DTC – *Diagnostic Trouble Code.*

ECU – elektronisk styrenhet.

FCF – *Flow Control Frame.*

FF – *First Frame.*

Φ – ett i förväg definierat antal UDS-förfrågningar i intervallet 1–200. I rapporten har Φ alltid samma värde.

Gränssnitt – hårdvarumodul som kan kommunicera över CAN-bussen.

HV/MV-kombination – hårdvaru- och mjukvarukombination.

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.*

python-can – Pythonbibliotek som hanterar CAN.

REC – *Receive Error Counter.*

RTR – *Remote Transmit Request.*

SAE – *SAE International.*

SF – *Single Frame.*

SID – *Service Identifier.*

Subfunktion – Vissa UDS-tjänster har möjligheten att kalla på en funktion som är direkt kopplad till den UDS-tjänsten.

TEC – *Transmit Error Counter.*

UDS – *Unified Diagnostic Service.*

UDS-protokollen – ISO14229, ISO15765-2 och ISO15765-3

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

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

udsoncan – Pythonbibliotek som användes för att kunna implementera UDS.

Innehållsförteckning

Upphovsrätt ... i
Copyright ... i
Sammanfattning ... ii
Abstract ... ii
Förord ... iv
Förkortningar och begrepp ... v
1 Introduktion ... 1
1.1 Bakgrund ... 1
1.2 Motivering ... 1
1.3 Mål ... 2
1.4 Problemformulering ... 2
1.4.1 Önskvärda krav ... 2
1.5 Avgränsningar ... 2
2 Relaterade arbeten ... 3
3 Teori ... 4
3.1 ECU ... 4
3.2 CAN ... 4
3.2.1 Meddelanden ... 5
3.2.2 CAN-bussen ... 6
3.2.3 Fysisk och funktionell adressering ... 7
3.2.4 Diagnostik på CAN-bussen ... 7
3.3 UDS ... 8
3.3.1 Diagnostiska sessioner ... 9
3.3.2 Säkerhet på UDS ... 9
3.3.3 UDS-förfrågningsmeddelande ... 9
3.3.4 Positiva svarsmeddelanden ... 10
3.3.5 Negativa svarsmeddelanden ... 11
3.3.6 Exempel på ett meddelandeflöde ... 11
3.3.7 UDS på nätverkslagret ... 12
3.3.8 UDS på transportlagret ... 13
3.4 UDS över CAN ... 14
3.5 Gränssnitt ... 16
3.6 Pythonbibliotek för fordonsdiagnostik ... 17
4 Metod ... 18

4.1 Implementation 18
4.1.1 Hårdvarukonfiguration 18
4.1.2 Programvarukonstruktion 19
4.2 Utvärdering 26
5 Resultat 28
5.1 Implementation 28
5.2 Utvärdering 30
6 Diskussion 32
6.1 Resultat 32
6.2 Metod 34
6.2.1 Implementation 35
6.2.2 Utvärdering: 36
6.3 Arbetet i större sammanhang 36
7. Slutsats 37
Litteratur 38
Bilaga 1. Utvärderingssvar 40
Bilaga 2. Sluttiderna för alla testomgångar 41

1 Introduktion

Detta kandidatarbete utfördes för Syntronic AB i Linköping. De arbetar med utveckling av nya bil-modeller 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 drev utvecklingen ännu mer. Innan denna standard infördes kommunicerade olika sensorer och elektroniska styrenheter (ECU) (sensorer och ECU:er kommer att inkluderas i noder framöver) direkt med varandra. Med tiden implementerades flera elektriska komponenter till bilen för att öka säkerheten och komforten och för att minska miljöpåverkan. Mellan 1990 och 2000 ökade antalet noder i en bil från 10 till 40 och idag kan det finnas mer än 70 noder i en bil. Det tidigare tillvägagångssättet var inte hållbart då det blev väldigt dyrt. Dessutom var inte kommunikationen mellan noderna tillförlitlig. Lösningen på problemet var CAN-bussen, en databuss som onödiggjorde kablage mellan alla noder. Utan CAN-bussen är det inte säkert att fordonsutvecklingen hade kunnat fortgå som den har gjort sedan 1990-talet [1]. Det medförde dock nya problem då testningen och felsökningen bland alla noder var ett komplext och tidskrävande arbete.

Unified Diagnostic Services (UDS) är ett diagnostiskt kommunikationsprotokoll. Med detta protokoll kan ett diagnostiskt verktyg kommunicera med alla ECU:er på CAN-bussen som har stöd för UDS vilket är en majoritet av ECU:erna i en modern bil. UDS-protokollen har även definierat vilka diagnostiska tjänster som finns och hur de ska användas. Innan UDS-protokollet standardiserades kunde bil-tillverkarna välja att implementera olika diagnostiska protokoll för att kunna utföra diagnostik på ECU:erna i en bil. Fordonsutvecklare och mekaniker kunde inte använda ett diagnostiskt verktyg för att kommunicera genom CAN-bussen på bilar från olika biltillverkare och denna problematik löstes med UDS.

1.2 Motivering

Syntronic AB:s Linköpingskontor arbetar med testning av delsystem i en bil. Testningen sker genom att läsa värdena från en ECU när bilen är stillastående. För att få ut data från en ECU måste livesignaler (meddelanden på CAN-bussen) skickas och läsas vilket utförs med ett gränssnitt som kommunicerar över CAN-bussen. 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 använder Syntronic ett gränssnitt från Vector tillsammans med en konfidentiell programvara (CarDiagnosticsProgram123) som utför diagnostiken enligt UDS-protokollen. CarDiagnosticsProgram123 är en avancerad programvara som har många funktioner, men den används enbart för att skicka UDS-förfrågningar och för att ta emot det returnerade svaret. För denna funktionalitet är CarDiagnosticsProgram123 för avancerad och tiden för uppstart och nedstängning av programmet påverkas negativt av detta. Den totala körtiden för att skicka ett UDS-förfrågningsmeddelande påverkas negativt då det tar ungefär 30 sekunder från att CarDiagnosticsProgram123 startats innan det första UDS-förfrågningsmeddelandet kan skickas. Övrig funktionalitet har vissa tidsförluster då användaren måste navigera genom olika menyer för att kunna utföra de olika testerna. Om testerna kan utföras snabbare kan antingen fler tester utföras under samma testsession vilket innebär en säkrare bekräftelse på att allt fungerar som det skall, alternativt kan bilen bli färdigtestad tidigare vilket innebär att fler modeller kan testas.

1.3 Mål

Målet med detta kandidatarbete var att skapa en programvara (CanDoRequests) som skulle vara snabbare än CarDiagnosticsProgram123. Detta testades genom att jämföra CanDoRequests tider mot en referenstid som sattes av CarDiagnosticsProgram123. CanDoRequests skulle vara konstruerad så att fler UDS-tjänster kunde läggas till i efterhand utan större modifikation. Programmet skulle kunna utföra fördefinierade UDS-förfrågningar från en fil, tillåta användaren att skapa enskilda UDS-förfrågningsmeddelanden, kunna spara ned svarsmeddelandena från UDS-förfrågningsmeddelandena till en textfil automatiskt och vara snabbare än referenstiderna som togs med CarDiagnosticsProgram123. Det tordes kunna göras genom en mjukvara som var specialiserad utefter Syntronics behov genom att endast implementera den funktionalitet som användes och genom att undvika menyer för att växla mellan att skicka enskilda UDS-förfrågningsmeddelanden och att köra filen som innehöll Φ UDS-förfrågningar. För att kunna skapa programvaran undersöktes både UDS-protokollen och hur UDS-meddelanden kunde skickas och tas emot över CAN-bussen.

1.4 Problemformulering

Följande problemformulering togs fram och baserades på målbilden som angavs i kapitel 1.3.

1) Hur kan en mjukvara som ska utföra fordonsdiagnostik enligt UDS-protokollen konstrueras så att total körtid, inklusive uppstart av mjukvaran, är snabbare än referenstiden som sattes av CarDiagnosticsProgram123?

1.4.1 Önskvärda krav

Följande önskvärda krav var inte en del av målbilden som angavs i kapitel 1.3 utan dessa önskvärda krav från Syntronic AB skulle tillgodoses om tid fanns.

2) Mjukvaran ska kunna kommunicera på CAN-bussen med flera olika gränssnitt.
3) Mjukvaran ska kunna live-visualisera svarsmeddelanden som kommer från UDS-förfrågningsmeddelandena.
4) Mjukvaran ska kunna visualisera de automatiskt sparade svarsmeddelandena.

1.5 Avgränsningar

Detta kandidatarbete hanterade UDS över CAN. UDS kan implementeras på andra nätverk som LIN och Ethernet, men det ligger utanför detta arbetes omfång och inget fokus lades på det.

Det finns två versioner av CAN, klassiskt CAN och CAN FD som är en nyare version av CAN. Det här kandidatarbetet tog ingen hänsyn till CAN FD utan all information om CAN baserades på klassiskt CAN.

Alla UDS-tjänster implementerades inte då det är ett komplext arbete och det var inte heller av intresse för Syntronic. UDS-tjänster som kräver särskild behörighet implementerades inte då ECU:ernas säkerhetsalgoritm inte innehades av författaren.

Fordonstillverkaren, vilket/vilka delsystem som testades, Syntronics nuvarande programvara och värdena från konfigurationsfilen som Syntronic AB fick av fordonstillverkaren var konfidentiella och beskrevs inte i rapporten. Detta gäller även antalet UDS-förfrågningar som fanns på filen med testfall.

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

Jinghua och Feng [3] skapade en automatiserad testmetod som baserades på UDS-protokollen. För att kunna göra detta använde de AutoCAN[1] från ihr. De importerade en ODX-fil för att skapa en script-generator som skapade test baserat på en script-mall, en importerad databasfil och användarens konfigurationer. Script-generatorn skapade nya testfall som sedan utfördes. När testet var klart skapades en loggfil automatiskt som innehöll resultatet från testfallen.

Assawinjaipetch et al. [4] implementerade UDS på en ECU medan detta kandidatarbete implementerade UDS på ett diagnostiskt verktyg (klient). De beskrev hur de ville att deras ECU skulle agera med UDS implementerat på nätverkslagret och transportlagret. Detta baserades på ISO15765-2 som detta kandidatarbete inte hade tillgång till vilket innebär 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

Detta kapitel presenterar teorin som det här kandidatarbetet baserades på. Voss [5] 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 det här kandidatarbetets fokus låg på personbilar baserades teorin på användningen i personbilar.

3.1 ECU

En ECU är ett inbyggt system som kontrollerar ett antal sensorer och ställdon. Varje ECU har olika funktioner i bilen och behöver kunna kommunicera med andra ECU:er över CAN-bussen [6]. ECU:erna har både hårdvara och mjukvara för att kunna utföra sina funktioner där hårdvaran innehåller minst ett sorts minne. Mjukvaran innehåller även metadata för ECU:n för att kunna identifiera och utföra olika UDS-tjänster. ISO15765-3 [7] 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 av flera ECU:er är en automatisk växellåda, airbagsystemet och ABS.

3.2 CAN

Kommunikationen mellan olika noder i en bil sker genom CAN-bussen. Alla noder kan kommunicera med varandra och när ett meddelande har skickats från en nod har det ingen fördefinierad rutt utan meddelandet skickas ut på CAN-bussen och alla noder kan läsa meddelandet. Dekanic et al. [8] hävdar att fördelen med CAN är att den har en hög immunitet mot elektriska störningar och att detta är en av anledningarna till att CAN fungerar bra i personbilar. Pazul [9] 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. Assawinjaipetch et al. [4], Voss [5], Pazul [9] och Cia [10] säger att CAN är uppbyggt efter ISO/OSI 7-lagersmodellen som visas i figur 1. Voss [5] och Cia [10] påstår att CAN använder lager 1, 2 och 7 medan Assawinjaipetch et al. [4] och Pazul [9] 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 som är ISO11898, men Voss [5] och Cia [10] menar att det finns vissa standardiserade CAN-protokoll på applikationslagret såsom CANopen[2]. Voss [5] hävdar att applikationslagret interagerar med applikationen eller operativsystemet på noden medan datalänkslagret och det fysiska lagret är integrerat på nodens chip.

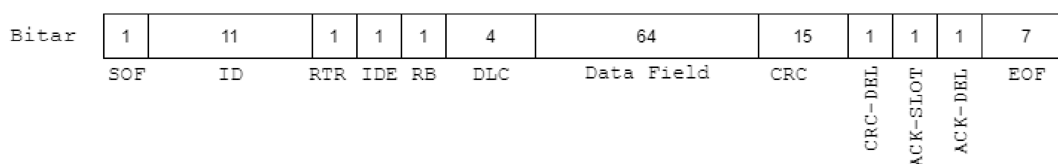
[2] https://www.can-cia.org/canopen/

7	Application layer
6	Presentation layer
5	Session layer
4	Transport layer
3	Network layer
2	Data link layer
1	Physical layer

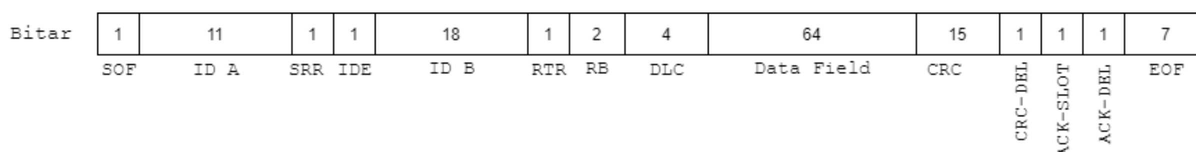
Figur 1. OSI:s 7-lagersmodell över fordonsnätverket. CAN är implementerat på det fysiska lagret och på datalänkslagret enligt ISO11898-1.

3.2.1 Meddelanden

Det finns två format på meddelanden som kan 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 och det är meddelandets ID-fält som bestämmer prioriteten. I CAN-protokollet anses en logisk 0:a vara dominant över en logisk 1:a vilket innebär att ju lägre värde ID-fältet har desto högre prioritet har meddelandet [9]. Meddelandets datafält är 64 bitar långt och det är i detta fält som meddelandets data finns. Meddelandet har även en kontrollsumma i CRC-fältet som kontrollerar att meddelandet är oförvanskat [8, 9].



Figur 2. Ett meddelande i standardform. I CAN-bussen har ett meddelande i standardform högre prioritet än ett utvidgat meddelande. IDE-biten är alltid satt till 0 i ett meddelande i standardform.



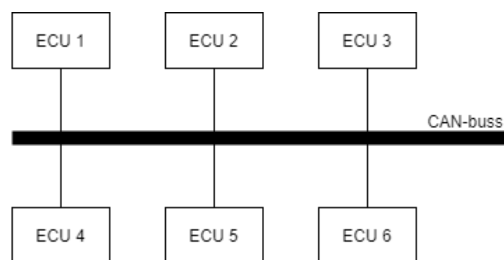
Figur 3. Ett utvidgat meddelande. SRR-biten måste vara satt till 1 i och med att meddelanden på standardform har högre prioritet. IDE-biten är alltid satt till 1 i utvidgade meddelanden vilket anger att meddelandet är i utvidgad form. ID-fältet är uppdelat i två olika fällt där ID A är den första delen 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. För datameddelanden 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 skickas 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 har upptäckts på databussen. Exempel på varför ett felmeddelande skickas är att kontrollsumman i CRC-fältet är felaktigt när mottagarnoden jämför sitt beräknade värde med den skickande nodens kontrollsumma. 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 CAN-bussen och meddelar att noden som skickade nämnda kö-meddelande inte är redo att ta emot nya meddelanden under en viss tidsperiod [9, 11].

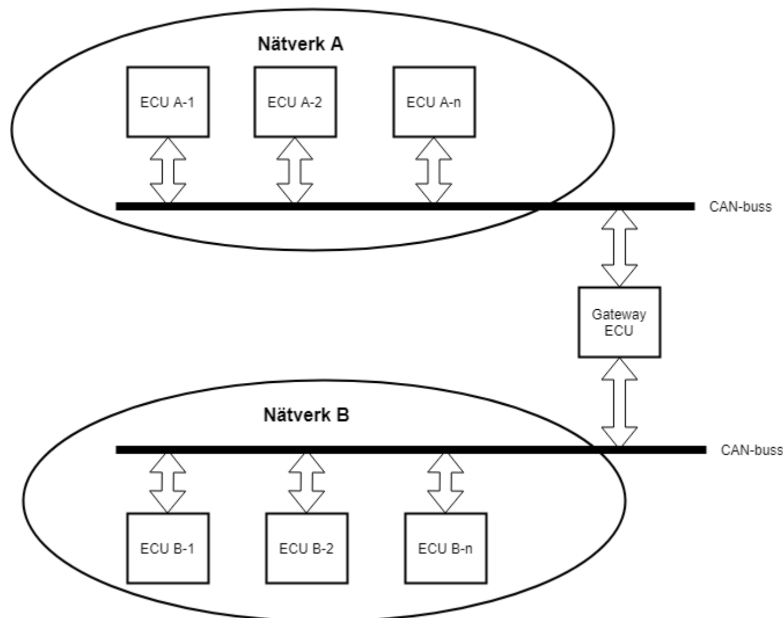
3.2.2 CAN-bussen

När CAN-bussen introducerades försvann behovet av att ha alla noder direkt ihopkopplade med varandra. Noderna i bilen kopplades istället upp på en seriell databuss och de kunde kommunicera med varandra via databussen. Dekanic et al. [8] säger att den maximala signalhastigheten är 1 Mbps och att maximalt 30 noder kan vara uppkopplade på en CAN-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 CAN-bussen och inte direkt till ECU 3. Alla noder på CAN-bussen kan läsa meddelandet, men det kommer bara att accepteras av noder som är programmerade till att acceptera meddelanden med det specifika identifikationsnumret som meddelandet har. När meddelandet har accepterats av ECU 3 kommer ACK-Slot-biten att sättas till 0 om meddelandet var oförvanskat. Exempelvis kommer ECU 5 att 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 en bra signalstyrka behöver varje ände av CAN-bussen ha ett 120 ohms motstånd mellan kablarna [12]. Ett fordon kan ha över 70 noder men maximalt 30 noder kan vara uppkopplade på samma CAN-buss. För att kunna koppla ihop olika CAN-bussar används en *gateway* ECU som dirigerar trafik mellan olika subnätverk så att meddelanden kan 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 [7, 12].



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 [9] 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 under den tidsperioden innan de kan skicka ett meddelande. När denna aktivitetsfria tid är avklarad har alla noder möjlighet att skicka meddelanden. Två eller fler noder kan försöka att 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 att få tillgång till databussen medan meddelandet med den lägre prioriteten kommer att stoppas. När meddelandet med den högre prioriteten har skickats 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 [8, 9].

3.2.3 Fysisk och funktionell adressering

Det finns två metoder för att skicka ett meddelande över CAN-bussen. Det sker antingen genom fysisk adressering eller genom funktionell adressering. Det är avsändaren av meddelandet som bestämmer vilken sorts adressering som sker genom 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 [7] anger att mottagaradressen ska vara 0x7FF för att sända ut meddelandet över CAN-bussen med funktionell adressering [7, 13]. 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 har upptäckts. Förutom att tända lampan sparar systemet även viktig information om det upptäckta felet som en diagnostisk felkod (DTC) [14]. OBD II-uttaget ska vara placerat på en plats i bilen

man kan nå från förarsätet och när man ansluter en klient till OBD II-uttaget kommer klienten att få tillgång till CAN-bussen. 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. Mer avancerade verktyg har en större tillgång till meddelandena på CAN-bussen och har även möjligheten att låsa upp fler funktioner i ECU:erna. Fordonsutvecklare behöver ofta köpa in dyra verktyg från *Orignal Equipment Manufacturer* (OEM), som i detta fall är fordonstillverkarna, för att kunna testa olika delsystem i bilarna. En klient behöver inte alltid kopplas upp via OBD II-uttaget utan de kan även kopplas upp direkt på noden som ska testas genom specialutrustning.

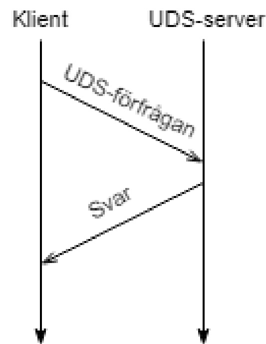
Kelkar och Kamal [15] säger att CAN hanterar fel på nodnivå och att felhanteringen beror på nodens beteende i något av de tre tillstånden som en nod kan befinna sig i, aktiv, passiv eller *bus-off*. En nod som visar ett felaktigt beteende kommer att stängas av för att säkerställa att CAN-bussen kan fortsätta att fungera korrekt. Varje nod har två felräknare, *Transmit Error Counter* (TEC) och *Receive Error Counter* (REC) och det finns flera regler som bestämmer hur dessa felräknare 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 kommer inte längre att kunna skicka några meddelanden på databussen [11, 15].

3.3 UDS

UDS är en standard som är 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 på applikationslagret i OSI:s 7-lagersmodell, men har även speciella förhållningsregler på nätverkslagret och på transportlagret. Alla noder i ett fordonsnätverk har inte stöd för UDS men det blir vanligare och idag har nästan alla noder i ett fordonsnätverk UDS implementerat. Det finns flera fördelar med UDS, bland annat är det lättare, både för utvecklare och för mekaniker, att felsöka noderna i fordonsnätverket [16]. Embitel [16] sammanfattar fyra kategorier av UDS-tjänster i ISO14229 [13] som behandlar UDS. Dessa är:

1) dataöverföringsmöjligheter som 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 kommer en DTC att sparas i ECU:ns minne. Denna DTC kan sedan läsas av en klient vilket bland annat underlättar felsökningen för en mekaniker.
3) upp- och nedladdningsmöjligheter som innebär att mjukvaran i en UDS-server bland annat kan uppdateras.
4) *Remote routine activation* som kan användas om ett test behöver köras över en viss tidsperiod, exempelvis vill kanske en mekaniker testa bilens motorfläkt och spara ned data och då kan denna kategori användas.

Embitel [16] nämnde inte den femte kategorin [17]. 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 ett fordonsnätverk genom CAN-bussen. Det fungerar genom att en klient skickar ett UDS-förfrågningsmeddelande till en UDS-server som returnerar ett svarsmeddelande vilket illustreras i figur 5. Både UDS-förfrågningsmeddelandet och svarsmeddelandet följer standarden för meddelanden över CAN-bussen. Både Dekanic et al. [8] och Salcianu och Fosalau [2] bekräftar ISO14229 [13] som anger att standarden för diagnostiska tjänster har ett gemensamt meddelandeformat som innefattar UDS-förfrågningsmeddelanden, positiva svarsmeddelanden och negativa svarsmeddelanden. Varje UDS-tjänst i UDS-protokollen 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 för utvecklare, vissa värden är reserverade för framtida bruk medan andra är reserverade av ISO14229 [13].



Figur 5. Kommunikationen initieras alltid av klienten enligt UDS-protokollen medan UDS-servern svarar på klientens UDS-förfrågningsmeddelande.

3.3.1 Diagnostiska sessioner

UDS-protokollen definierar fyra olika diagnostiska sessioner som en ECU kan befinna sig i. Endast en diagnostisk session måste vara implementerad och det är standardsessionen. När en ECU har startats ska den alltid befinna sig i standardsessionen. UDS-tjänsterna som kan utföras i standardsessionen är begränsade och säkerhetskritiska UDS-tjänster kan kräva särskilda behörigheter. 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 exempelvis airbagens inställningar [13].

UDS-tjänsten *TesterPresent* (0x3E) används för att hålla en session aktiv. I standardsessionen behövs inte denna UDS-tjänst för att hålla sessionen aktiv, men vid aktivitet i övriga sessioner behöver klienten skicka ett UDS-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 om UDS-tjänsten 0x3E inte har efterfrågats kommer ECU:n att återgå till standardsessionen efter en viss tidsperiod [13, 18].

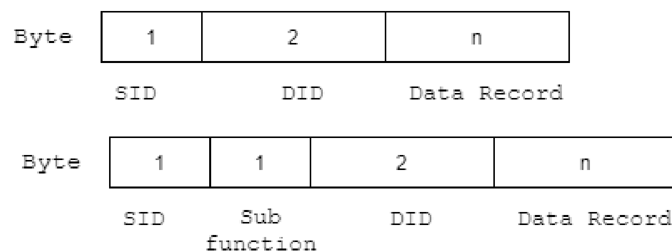
3.3.2 Säkerhet på UDS

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 som 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 ett UDS-förfrågningsmeddelande med nyckeln som hör till den mottagna initieringen. Om nyckeln var giltig kommer klienten att få tillgång till den efterfrågade UDS-tjänsten [13].

3.3.3 UDS-förfrågningsmeddelande

Ett UDS-förfrågningsmeddelande kan bara skickas i en riktning, från en klient till en UDS-server. Figur 6 visar ett exempel på två olika UDS-förfrågningsmeddelanden. Endast ett fält är obligatoriskt, SID-fältet som innehåller UDS-tjänstens värde och är en byte stort. SID-värdets betydelse är identiskt i kli-

enten och på UDS-servern vilket garanterar att den efterfrågade UDS-tjänsten har samma betydelse för både klienten och för UDS-servern. 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 UDS-förfrågningsmeddelandet 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-värdet och subfunktionernas värden är standardiserade i ISO14229 och har samma betydelse i alla ECU:er som har UDS implementerat. *Diagnostic ID* (DID)-fältet ska vara två byte stort enligt ISO14229 och dessa värden är inte standardiserade i UDS. DID-värdet refererar till olika lokala dataposter i en ECU, ett 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. Alla UDS-tjänster kräver inte ett DID-värde. *Data Record* (dataposten) innehåller metadata och är länkat till DID-värdet om UDS-tjänsten krävde ett DID-värde. Dataposten är obligatoriskt för vissa UDS-tjänster och valbart för andra UDS-tjänster. Både DID-värdet 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, och de kan även skilja sig mellan olika årgångar av samma bilmodell [13, 19, 20].



Figur 6. Ett exempel på två olika UDS-förfrågningsmeddelanden. Det övre meddelandet saknar en subfunktion medan det nedre meddelandet har en subfunktion. Ett UDS-förfrågningsmeddelande har inte en bestämd storlek. Endast SID-värdet är obligatoriskt då det beskriver vilken UDS-tjänst som ska utföras. Det innebär att övriga fält är obligatoriska/valbara beroende på vilken UDS-tjänst som har efterfrågats.

Vissa UDS-tjänster har stöd för subfunktionen *suppressPosRspMsgIndicationBit*. Denna subfunktion meddelar UDS-servern att ett positivt svarsmeddelande inte behöver skickas. Denna subfunktion är lämplig att använda när det positiva svaret inte innehåller någon datapost utan mest agerar som ett kvitto på att UDS-förfrågningen kunde utföras. Om UDS-förfrågningen inte kunde utföras kommer ett negativt svarsmeddelande att skickas från UDS-servern 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. 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. Svarsmeddelandet från denna UDS-förfrågning innehåller ingen datapost utan svarsmeddelandet innehåller endast UDS-förfrågningsmeddelandets SID-värde + 0x40 och en kopia på värdet i subfunktionen och tar således 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).

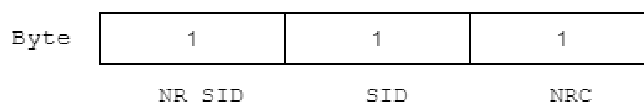
3.3.4 Positiva svarsmeddelanden

Om UDS-förfrågningsmeddelandet var korrekt formaterat och UDS-förfrågningen kunde utföras kommer UDS-serven att utföra UDS-förfrågningen och sedan 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 vilken UDS-tjänst som efterfrågades. Alla positiva svarsmeddelanden skickar tillbaka UDS-förfrågningsmeddelandets SID-värde men

adderar 0x40 till det positiva svarsmeddelandets SID-värde för att påvisa att det är ett positivt svarsmeddelande. Ett undantag finns vilket är om svarsmeddelandet är av typ två från UDS-tjänsten *ReadDataByPeriodicIdentifer* som är specificerad i ISO14229. Om UDS-förfrågningsmeddelandet hade SID-värdet 0x14 kommer det positiva svarsmeddelandet att ha SID-värdet 0x54. Om UDS-förfrågningsmeddelandet hade en subfunktion och/eller ett DID-värde kommer dessa värden att vara identiska i svarsmeddelandet. Om den efterfrågade UDS-tjänstens svar innehåller annat data kommer det att finnas i dataposten [13].

3.3.5 Negativa svarsmeddelanden

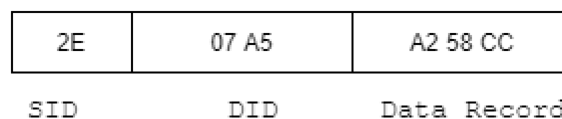
Om UDS-förfrågningsmeddelandet inte var korrekt formaterat eller om UDS-tjänsten inte kunde utföras kommer UDS-servern att skicka tillbaka ett negativt svarsmeddelande som visas i figur 7. Det första fältet, *Negative Response Service ID* (NR_SID), är alltid 0x7F enligt UDS-protokollen. Det andra fältet innehåller en kopia av UDS-förfrågningsmeddelandets SID-värde. Det tredje fältet innehåller *Negative Response Code* (NRC) och är en byte stort. NRC-fältet innehåller ett värde som anger varför svaret från UDS-förfrågningen blev negativt. Det finns ett flertal anledningar till att ett svar blir negativt, exempelvis kan det bero på att UDS-förfrågningsmeddelandet var felaktigt formaterat, att DID-värdet inte stöds av UDS-servern eller att behörighet att utföra UDS-tjänsten saknas [13].



Figur 7. Ett negativt svarsmeddelande är 3 byte stort och innehåller ett negativt SID-värde, en kopia på UDS-förfrågningsmeddelandets SID-värde och en negativ responskod.

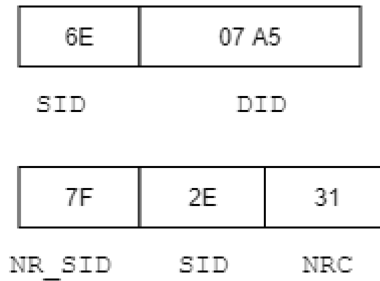
3.3.6 Exempel på ett meddelandeflöde

Det här kapitlet ger 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-fältet och för dataposten. Figur 8 visar hur ett UDS-förfrågningsmeddelande från en klient till en UDS-server kan se ut när UDS-tjänsten 0x2E efterfrågas.



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

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



Figur 9. Det övre meddelandet är ett positivt svarsmeddelande för UDS-tjänsten 0x2E. Svarsmeddelandet innehåller värdet 0x2E + 0x40 som SID-värde och en kopia på DID-värdet från UDS-förfrågningsmeddelandet. Det nedre meddelandet är ett negativt svarsmeddelande. Detta syns tydligt då den mest signifikanta byten har värdet 0x7F som är reserverad för negativa svarsmeddelanden oavsett vilken UDS-tjänst som har efterfrågats. Den andra byten är en kopia på SID-värdet från UDS-förfrågningsmeddelandet. Den tredje byten är den negativa responskoden som är fördefinierad i ISO14229 och anger varför det blev ett negativt svarsmeddelande.

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

3.3.7 UDS på nätverkslagret

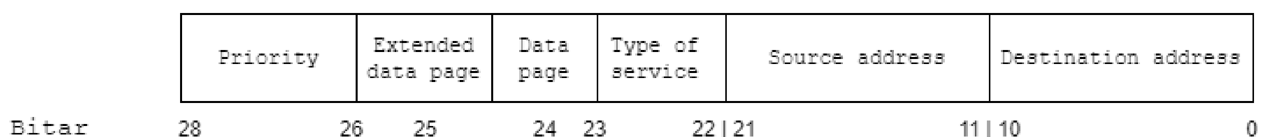
ISO15765-3 [7] beskriver hur CAN-meddelandets ID-fält ska användas för fordonsdiagnostik på nätverkslagret i OSI:s 7-lagersmodell. Protokollet skiljer på hur CAN-meddelanden i 2.0A-formatet (se figur 2) och CAN-meddelanden i 2.0B-formatet (se figur 3) ska användas tillsammans med UDS.

3.3.7.1 CAN 2.0A

CAN-meddelanden i 2.0A-formatet har ett 11 bitars ID-fält och kan användas av någon av de tre övriga sessionerna som beskrevs i kapitel 3.3.1. ISO15765-3 [7] anser däremot att det endast är UDS-tjänsten 0x3E som bör använda ett CAN-meddelande i 2.0A-formatet.

3.3.7.2 CAN 2.0B

CAN-meddelanden i 2.0B-formatet har ett 29 bitars ID-fält som är uppdelat i två delar, ID A med 11 bitar och ID B med 18 bitar. Detta format är att föredra när ett UDS-förfrågningsmeddelande ska skickas då både mottagarnodens adress och avsändarnodens adress anges vilket medför att UDS-servern kan skicka ett svar till klienten. De 29 bitarna i ID-fältet är fördelade enligt figur 10.



Figur 10. De 29 bitarna i CAN-meddelandets ID-fält fördelas enligt följande: De 11 minst signifikanta bitarna är mottagarnodens adress, de nästkommande 11 bitarna är avsändarnodens adress, sedan är bit 22 och 23 dedikerade för vilken typ av meddelande det är. Bit 24 och 25 anger vilket format som följs medan bit 26–28 anger vilken prioritet meddelandet ska ha.

Vid diagnostiska meddelanden ska prioritetsfältet ha värdet 0x6 [7]. För att följa ISO15765-3, implementation av UDS på CAN, ska meddelandet formateras enligt följande, 0x6[1]F[2](0-7FF)[3](0-7FF)[4] [7]. 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 ISO15765 formatet som ska följas och bitarna 22 och 23 anger att meddelandetypen kommer att följa ISO15765-3.
3) avsändarnodens adress ska vara 11 bitar vilket innebär värden i intervallet 0x0-0x7FF.
4) mottagarnodens adress ska vara 11 bitar vilket innebär värden i intervallet 0x0-0x7FF.

3.3.8 UDS på transportlagret

Datafältet i ett CAN-meddelande är 64 bitar vilket är 8 byte. Om ett UDS-meddelande är större än 7 byte behöver meddelandet segmenteras för att data inte ska försvinna. Utan segmenteringen kommer endast de 7 första byten att skickas och övriga byte kommer att förloras. ISO15765-2 (ISO-TP) är implementerat på transportlagret i OSI:s 7-lagersmodell och definierar hur meddelandeflödet ska utföras över CAN-bussen. För att få information om UDS-meddelandet använder transportlagret en byte av CAN-meddelandets datafält för *Protocol Control Information* (PCI), som används för att kontrollera meddelandeflödet. Det läggs till i början av CAN-meddelandets datafält som den mest signifikanta byten. De 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 storleken på innehållet i datafältet (DL) i byte. 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 har. 6 byte data kan skicks i ett FF. Consecutive Frame har TOF värdet 0x2 och 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 sitt värde 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 CF 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 CAN-meddelande ska enligt ISO-TP alltid vara 8 byte stort och icke använda byte fylls på med utfyllnadsbytes med värdet 0xAA, 0x55 eller 0x00.

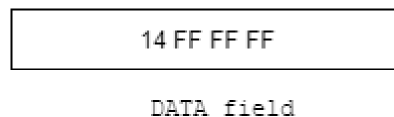
Om hela datafältets innehåll får plats i ett CAN-meddelande används ett SF. Det innebär att datafältets innehåll inte kan vara mer än 7 byte stort då PCI tar upp den åttonde byten. Om datafältets innehåll inte får plats i ett CAN-meddelande delas datafältets innehåll upp i flera olika CAN-meddelanden. Det första CAN-meddelandet som skickas är ett FF som innehåller de 6 första byten av

datafältets innehåll samt den totala storleken för de resterande CAN-meddelandena. Mottagaren av FF skickar ett 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 kan skickas i ett block bestäms av FCF och när alla CF i blocket har skickats kommer ett nytt FCF att 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 att ignoreras av noderna [4, 6, 21, 22].

3.4 UDS över CAN

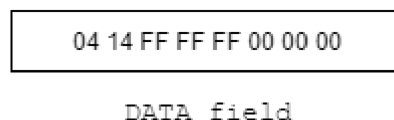
För att beskriva hur UDS är implementerat på en klient som kommunicerar över CAN-bussen anges ett exempel som visar vilken information som läggs till på applikationslagret, transportlagret och nätverks-lagret. I exemplet kommer en UDS-förfrågning med UDS-tjänsten *ClearDTCInformation* (0x14) att efterfrågas och dataposten kommer att innehålla värdet 0xFFFFFF som betyder att alla UDS-servrar ska nollställa sina eventuella DTC:er som finns lagrade i ECU:ns minne. Avsändarnodens adress är 0x456 och mottagarnodens adress är 0x123 i exemplet.

UDS-förfrågningen skickas från applikationslagret och kommer innehålla SID-värdet 0x14 och dataposten 0xFFFFFF. Figur 12.1 visar innehållet i UDS-förfrågningen.



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

På transportlagret läggs PCI till som mest signifikanta byte i CAN-meddelandets datafält. PCI baseras på informationen som kommer från applikationslagret. Om den mottagna informationen är större än 7 byte kommer transportlagret att sköta segmenteringen av meddelandet och sedan skicka rätt sorts med-delandetyper vilket förklarades i kapitel 3.3.8. Figur 12.2 visar var PCI läggs till i UDS-förfrågningen från applikationslagret.



Figur 12.2. UDS-förfrågningsmeddelandet är färdigt. PCI läggs till som mest signifikanta byte. I detta scenario har PCI 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 UDS-förfrågningsmeddelandet till nätverkslagret där CAN-meddelandets ID-fält formuleras. CAN-meddelandet är i CAN 2.0B-formatet då UDS-tjänsten inte är 0x3E. Figur 12.3 visar informationen som läggs till på nätverkslagret där klienten är avsändarnod och UDS-servern är mottagarnod.

110	1	1	11	10001010110	00100100011
Priority	Extended data page	Data page	Type of service	Source address	Destination address

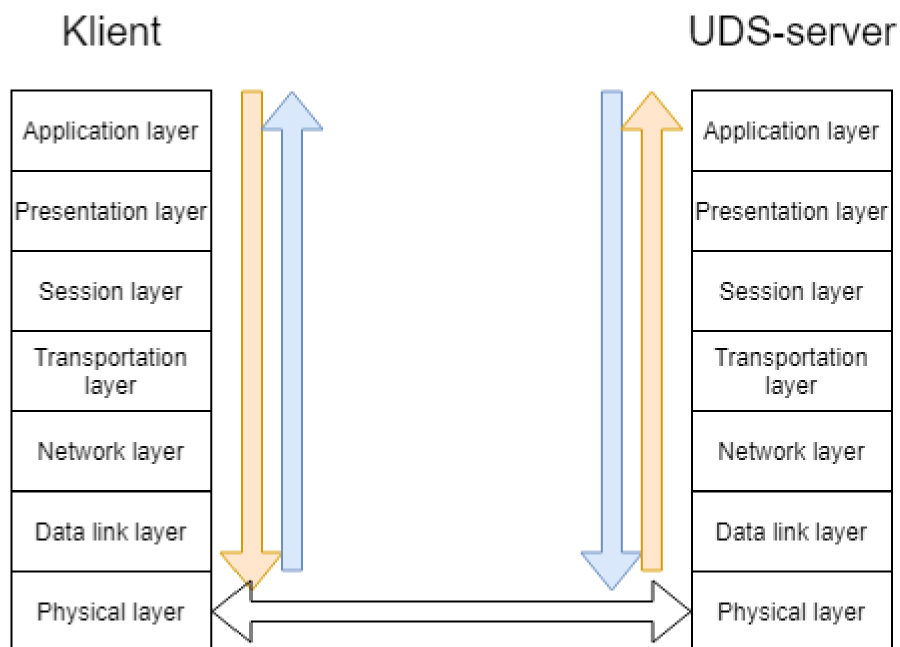
Figur 12.3. Mottagarnodens adress är 0x123 och avsändarnodens adress är 0x456. Prioritetsfältet ska vara 0x6 då datafältet innehåller ett UDS-meddelande. Extended data page och Data page anger att meddelandet följer ISO15765 och Type of service anger att det är ISO15765-3 som ska följas.

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

0	110 1111 1000	1	1	1010110 00100100011	0	00	0101	04 14 FF FF FF 00 00 00	15 bits	1	1	1	1111111
SOF	ID A	SRR	IDE	ID B	RTR	RB	DLC	DATA	CRC	CRC-DEL	ACK-SLOT	ACK-DEL	EOF

Figur 12.4. CAN-meddelandet som skickas från avsändarnoden via CAN-bussen. Det 29 bitars ID-fältet 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. DLC-fältet innehåller värdet 0x5 då datafältet innehåller 5 byte med data. Detta fält bestäms tillsammans med övriga fält i ISO11898.

Alla noder på CAN-bussen kommer att kunna läsa meddelandet, men det är endast noden med den unika adressen 0x123 som kommer att ta emot CAN-meddelandet. Mottagarnoden kommer att ta emot meddelandet och försöka att utföra UDS-förfrågningen på applikationsnivån. Ett svarsmeddelande formuleras där mottagarnodens adress och avsändarnodens adress byter plats, ny avsändarnod är 0x123 och ny mottagarnod är 0x456. Om UDS-servern kan ge ett positivt svarsmeddelande kommer svarsmeddelandet att ha värdet 0x54. Om svaret var negativt kommer svarsmeddelandet att innehålla värdena 0x7F, 0x14 och 0xXX där XX representerar möjligt NRC-värde. Svarsmeddelandet kommer att hanteras i ett likadant flöde som UDS-förfrågningsmeddelandet som visades i figur 12.1–12.4. Figur 13 illustrerar meddelandeflödet mellan klienten och UDS-servern.



Figur 13. Den vita pilen mellan klientens fysiska lager och UDS-serverns fysiska lager representerar CAN-bussen. Flödet för klientens UDS-förfrågningsmeddelande illustreras av de två yttre pilarna som är orangea. UDS-förfrågningsmeddelandet skickas från klientens applikationslager och de olika lagren tillsätter data till CAN-meddelandet innan det når det fysiska lagret. UDS-servern kommer att ta emot CAN-meddelandet från CAN-bussen då den identifierar UDS-förfrågningsmeddelandets angivna mottagaradress som sin egna unika adress. De olika lagren kommer att ta del av den informationen från CAN-meddelandet de behöver innan UDS-förfrågningsmeddelandet når UDS-serverns applikationslager. De inre pilarna som är blåa representerar flödet för UDS-serverns svar på klientens UDS-förfrågningsmeddelande.

3.5 Gränssnitt

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

Två gränssnitt användes i detta arbete och visas i figur 14. Kvaser Leaf Light HS var det primära gränssnittet som CanDoRequests utvecklades på medan PCAN-USB FD främst användes för att verifiera data som skickades 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 den blivit ersatt av Kvaser Leaf Light HS v2. Kvaser säger dock att program som är skrivna för ett gränssnitt kommer att kunna köras på alla andra typer av gränssnitt från Kvaser utan att programmet behöver ändras [23].

Figur 14. De två gränssnitten som användes vid utvecklingen av mjukvaran. 1) är Kvaser Leaf Light HS och 2) är PCAN-USB FD från PEAK. Båda ansluts till en dator genom USB-kontakten och till CAN-bussen genom var sin 9 pinnars D-sub-kontakt.

3.6 Pythonbibliotek för fordonsdiagnostik

Kommunikation mellan mjukvaran och CAN-bussen kan utföras genom socketprogrammering i Python. Det finns även bibliotek som är dedikerade för kommunikation med CAN-bussen. Python-can är ett bibliotek som stöder kommunikation med CAN-bussen. Gränssnittstillverkarnas egna bibliotek kan integreras med python-can, exempelvis har Kvaser biblioteket CANLIB och PEAK har biblioteket PCAN-Basic och båda dessa bibliotek kan integreras med python-can. Det finns även ett gränssnittsoberoende bibliotek, SocketCan, som är integrerat i Linux kärna och som delar resurser med internetprotokollet (IP). Fördelen med detta bibliotek är att det är generellt och fungerar till de flesta gränssnitten men nackdelen är att det delar resurser med IP vilket innebär att en hög internettrafik fördröjer trafiken på CAN-bussen. En annan nackdel är att SocketCan bara fungerar med Linux som operativsystem [24]. Det finns även dedikerade Pythonbibliotek för UDS och ISO-TP som heter udsoncan, Python-uds och can-isotp, och alla dessa bibliotek kan integreras med python-can.

4 Metod

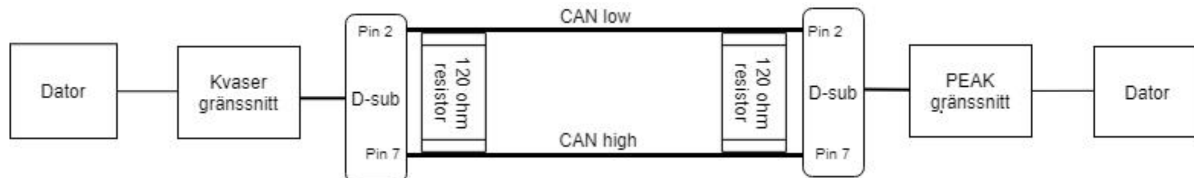
Följande kapitel beskriver metoden som användes för att besvara problemformuleringen som angavs i kapitel 1.4. För att kunna besvara problemformuleringen delades arbetet upp i två delar. Först behövde CanDoRequests skapas, sedan behövde programvaran utvärderas för att kunna fastslå om CanDoRequests var snabbare än CarDiagnosticsProgram123. Skapandet av CanDoRequests beskrivs genom en implementationsprocess i kapitel 4.1 medan utvärderingen utfördes genom att låta CanDoRequests och CarDiagnosticsProgram123 utföra samma experiment. Utvärderingen beskrivs i kapitel 4.2

4.1 Implementation

Konstruktionen av CanDoRequests baserades på teorin som angavs i kapitel 3 och genom nyttjande av befintliga Pythonbibliotek som behandlade CAN och UDS.

4.1.1 Hårdvarukonfiguration

Den första fasen av programvaruutvecklingen utfördes med två gränssnitt inkopplade mellan två datorer. För att kunna kontrollera att data kunde skickas och tas emot korrekt behövde en CAN-buss konstrueras där en dator som var kopplad till Kvaser-gränssnittet var en nod och en annan dator som var kopplad till PEAK-gränssnittet var den andra noden på CAN-bussen. Målet var att kunna skicka data mellan noderna på CAN-bussen. För att kunna utföra detta skapades en CAN-buss genom att löda ihop ett kabelpar med två 9 pinnars D-sub-kontakter och två 120 ohms motstånd. Figur 15 visar schemat för CAN-bussen och den färdiga CAN-bussen.

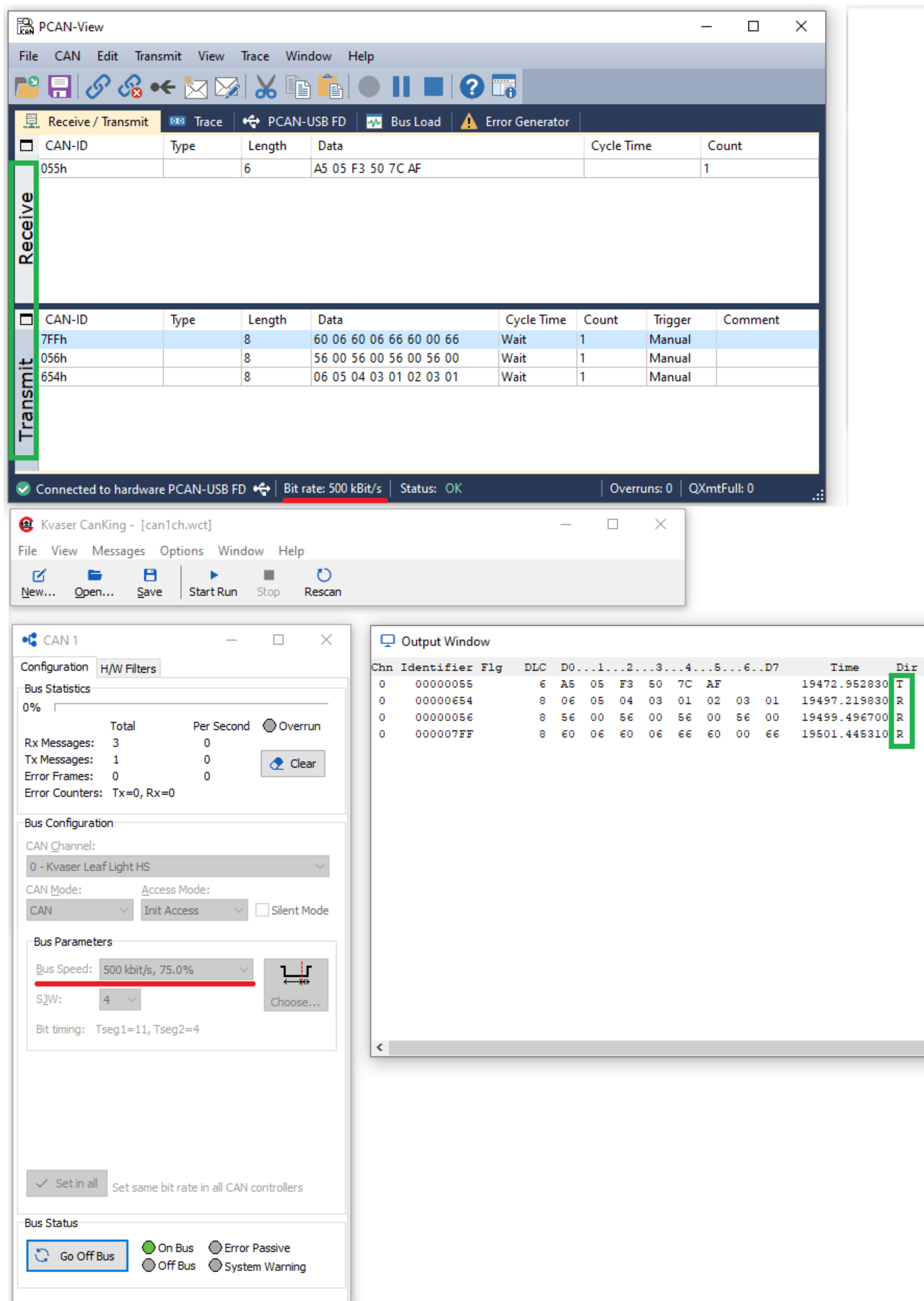


Figur 15. Överst på bilden visas schemat som användes för att kunna skapa en enkel CAN-buss. På D-sub-kontakten är pinne 2 CAN low-signalen och pinne 7 är CAN high-signalen, övriga pinnar behövde inte någon signal. Den nedre bilden visar den resulterande CAN-bussen. Fler noder kan kopplas upp på CAN-bussen mellan de båda 120 ohms motstånden. Det var viktigt att 120 ohms motstånden fanns i CAN-bussens båda ändar innan D-sub-kontakterna. Gränssnitten kopplades upp mellan var sin dator och på var sin 9 pinnars D-sub-kontakt.

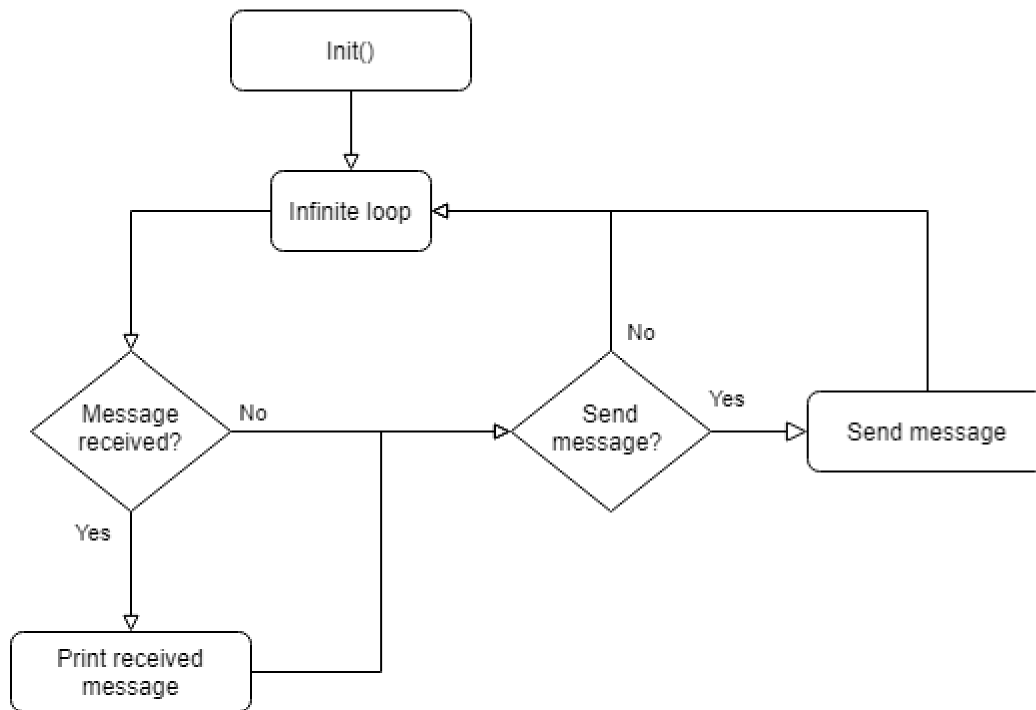
4.1.2 Programvarukonstruktion

Innan skapandet av CanDoRequests påbörjades testades kommunikationen mellan gränssnitten genom programvara från tillverkarna av båda gränssnitten. Kvaser erbjuder CANKING som gratis programvara för kommunikation över CAN-bussen och PEAK erbjuder PCAN-VIEW som gratis programvara för kommunikation över CAN-bussen. Båda dessa programvaror hanterar vanliga CAN-meddelanden. CAN-meddelanden skickades mellan gränssnitten för att kontrollera att de kunde kommunicera med varandra. Det var viktigt att det var samma busshastighet på båda gränssnitten för att de skulle kunna kommunicera med varandra. Figur 16 visar programmen från Kvaser och från PEAK.

När kommunikationen mellan gränssnitten kunde bekräftas skapades ett simpelt testprogram baserat på python-can för Kvaser Leaf Light HS som ersatte CANKING. Testprogrammet skapades för att kunna verifiera att en korrekt busskonfiguration kunde skapas. Ett förenklat flödesschema för testprogrammet visas i figur 17.



Figur 16. PCAN-VIEW är det övre programmet och CANKING är det nedre programmet. De två röda markeringarna visar att gränssnitten hade samma busshastighet. Om gränssnitten hade olika hastigheter för CAN-bussen fungerade inte kommunikationen. De gröna markeringarna visar riktningen för CAN-meddelandet, antingen har CAN-meddelandet skickats eller tagits emot. PCAN-VIEW har separata fönster för vilken riktning CAN-meddelandena hade medan CANKING samlade alla CAN-meddelanden i samma fönster och angav direktionen genom T = transmit och R = receive.

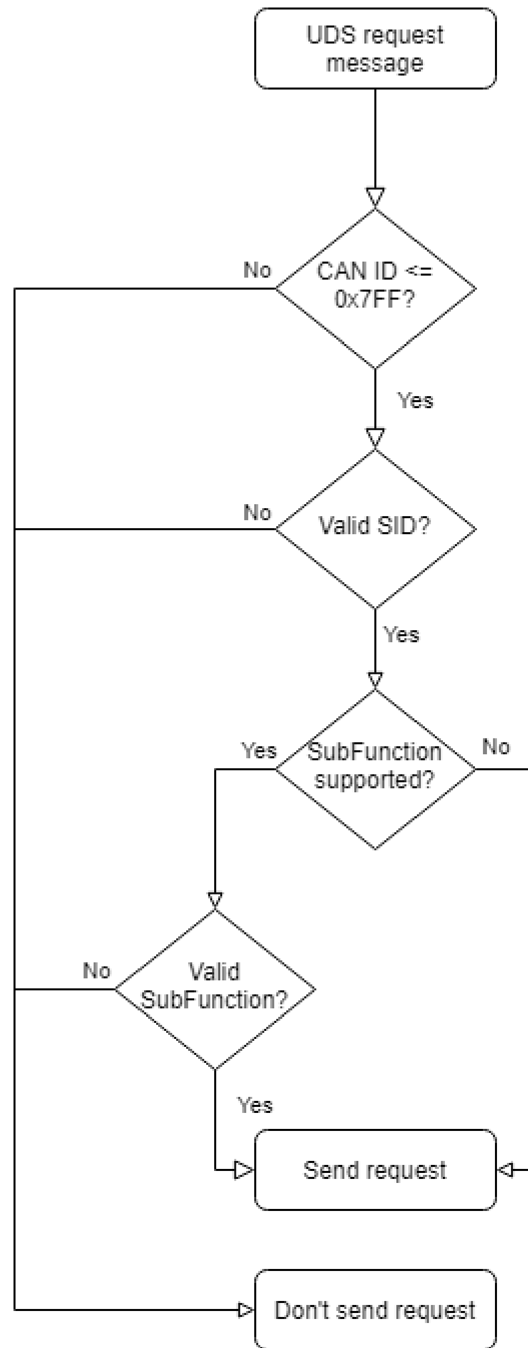


Figur 17. Följande funktionalitet implementerades tillsammans med ett grafiskt användargränssnitt för att kunna testa hur bussen skulle konfigureras och för att undersöka hur CAN-meddelanden kunde skickas och tas emot. Att lyssna efter CAN-meddelanden på CAN-bussen var ett blockerande funktionsanrop och kunde lösas genom att implementera funktionsanropet som en bakgrundstråd. När ett CAN-meddelande hade ankommit placerades datafältets innehåll och ID-fältets värde i ett kö-objekt. Den oändliga loopen var eventbaserad och kontrollerade först om eventen "Message received" hade inträffat. Det eventet kontrollerade innehållet i kön och så fort ett objekt fanns i kön togs det ut ur kön och skrevs sedan ut. Sedan kontrollerades det om eventet "Send message" hade aktiverats genom ett knapptryck. Om eventet hade aktiverats skickades ett CAN-meddelande med data inhämtat från användargränssnittet. Därefter skedde en ny iteration.

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

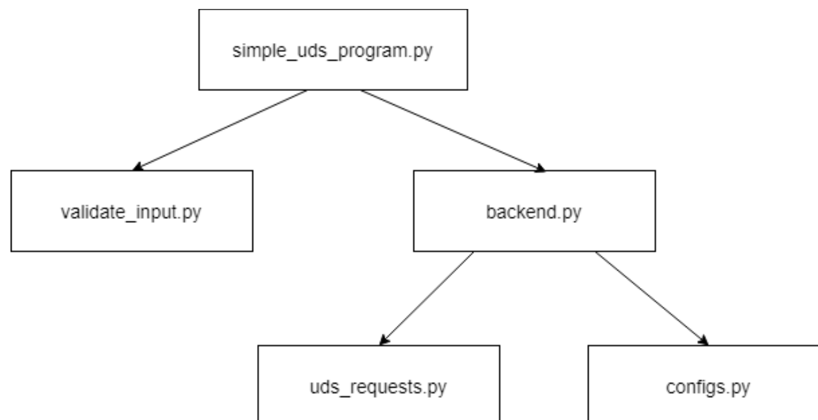
När dokumentationen från udsoncan hade undersökts påbörjades skapandet av CanDoRequests. Undersökningen av CAN visade att det kan vara mycket trafik över CAN-bussen, så det är viktigt att undvika onödiga CAN-meddelanden över CAN-bussen som ändå inte kommer att kunna genomföras. Figur 18 visar en enkel kontroll som undviker felaktigt formulerade UDS-förfrågningar.

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



Figur 18. En enkel kontroll kan utföras innan UDS-förfrågningsmeddelandet ska skickas. En nods adress kan vara högst 11 bitar vilket ger ett maximalt värde på 0x7FF. Noder med adresser högre än det värdet får inte finnas enligt UDS-protokollen och om användaren anger en adress som är högre än detta värde ska UDS-förfrågningsmeddelandet förkastas. Om SID-värdet var giltigt enligt UDS-protokollen och om UDS-tjänsten är implementerad i programvaran kommer även den eventuella subfunktionen att kontrolleras. Om UDS-tjänsten inte stöder subfunktioner är kontrollen klar och UDS-förfrågningsmeddelandet kommer att skickas. Om UDS-tjänsten stöder subfunktioner kommer värdet att kontrolleras och om det är giltigt kommer UDS-förfrågningsmeddelandet att skickas. Om det inte var giltigt kommer UDS-förfrågningsmeddelandet att förkastas. Implementationen av UDS-förfrågningsmeddelanden i udsoncan kontrollerar själv om det angivna DID-värdet finns med i den angivna konfigurationsfilen, om DID-värdet saknas kommer inte UDS-förfrågningsmeddelandet att skickas. På liknande sätt kontrollerar udsoncan även värdet i dataposten. Några UDS-tjänster har enligt UDS-protokollen ett maximalt värde för dataposten, exempelvis kan UDS-tjänsten 0x14 ha ett datapostvärde i intervallet 0x000000–0xFFFFFF, om en användare anger värdet 0x01000000 i dataposten kommer udsoncan själv att vägra skicka UDS-förfrågningsmeddelandet då värdet i dataposten är ogiltigt.

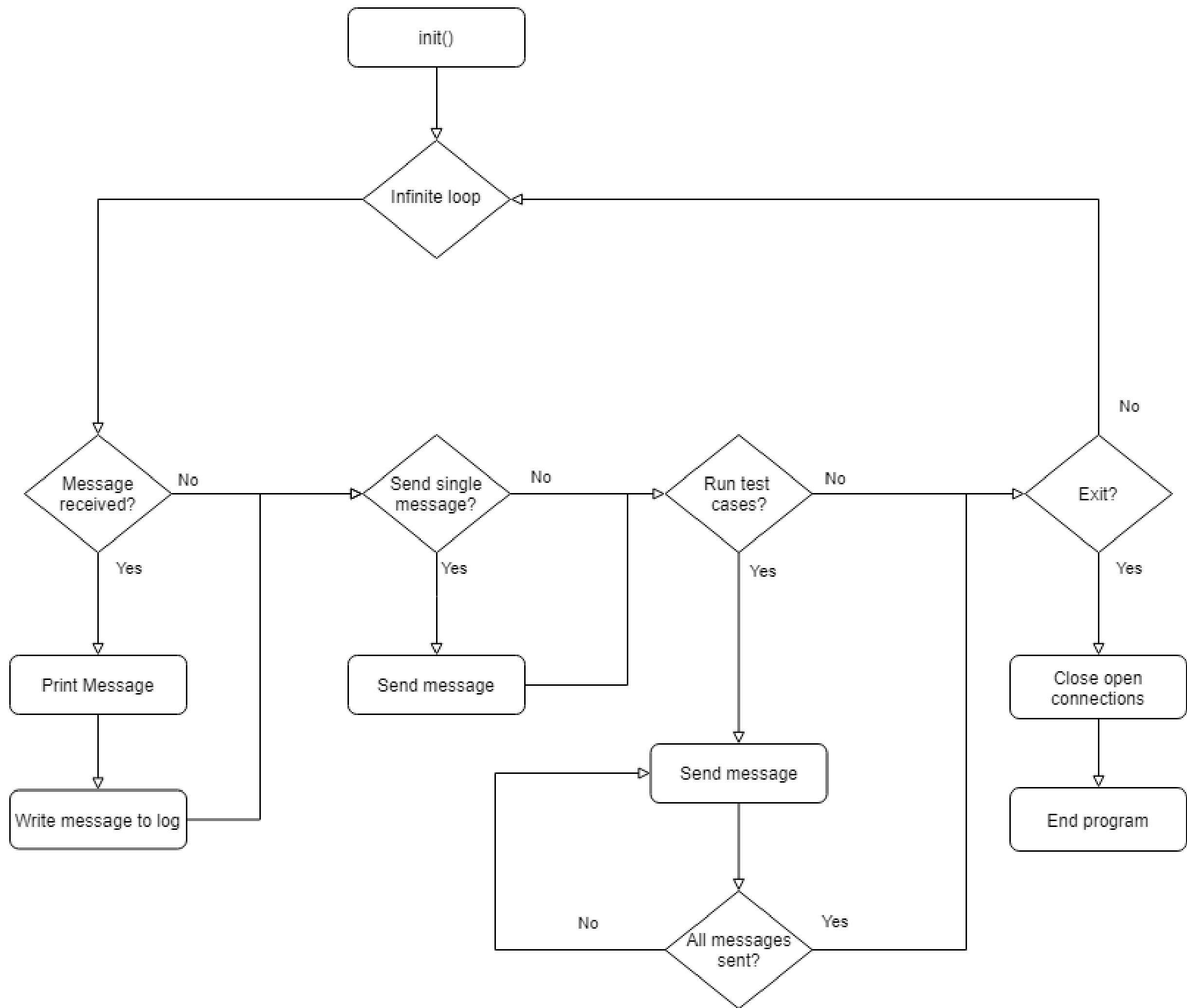
Fem filer skapades för att bygga programmet. Simple_uds_program.py var huvudprogrammet som skapade användargränssnittet och som sedan körde en oändlig loop som interagerade med användaren. Simple_uds_program.py anropade validate_input.py för att kontrollera att indatan från användargränssnittet var hexadecimalt och inom tillåtna gränsvärden. Backend.py utförde den huvudsakliga logiken, funktionaliteten och initieringen av de olika lagren i OSI:s 7-lagersmodell. Backend.py anropade i sin tur två filer, configs.py som innehöll fordonskonfigurationen som användes för att kunna skicka och tolka DID-värden och datapostens data medan uds_requests.py innehöll funktionsanropen till UDS-tjänsterna. Figur 19 illustrerar hierarkin för programmet. Som grafiskt användargränssnitt användes PySimpleGUI då dokumentationen var utförlig och inlärningsperioden tordes vara kort.



Figur 19. Ett träd över programmets hierarki. Ett funktionsanrop går från toppen av trädet och ned ett steg medan det returnerade värdet går upp ett steg i trädet. Exempelvis kan inte simple_uds_program.py anropa funktioner i uds_requests.py utan simple_uds_program.py måste anropa en funktion i backend.py som i sin tur anropar funktioner i uds_requests.py. Funktionen i backend.py kommer sedan att modifiera returvärdet från uds_requests.py innan den returnerar det efterfrågade värdet till simple_uds_program.py.

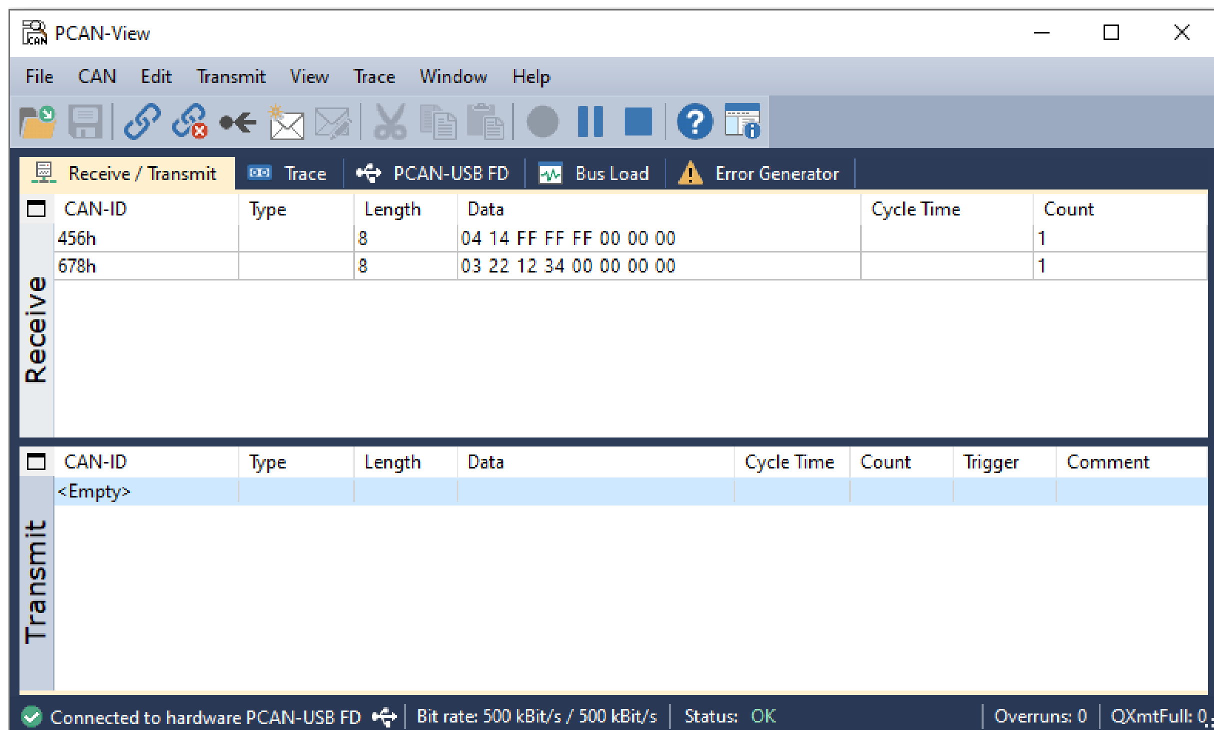
Figur 20 visar logiken för huvudprogrammets eventbaserade oändliga loop. UDS-förfrågningsmeddelanden genom udsoncan kunde inte användas med en bakgrundstråd som lyssnade på svarsmeddelanden då UDS-förfrågningarna gjordes genom ett funktionsanrop som returnerade ett svarsmeddelande. Om inget svarsmeddelande hade ankommit inom en i förväg definierad tidsperiod slutade funktionsanropet att lyssna efter ett svarsmeddelande på CAN-bussen och ett timeout undantag returnerades. Om ett svarsmeddelande hade ankommit placerades det i ett kö-objekt. Därmed kontrollerade den oändliga loopen fyra event:

1) Om kö-objektet inte var tomt hade ett svarsmeddelande ankommit och kunde därmed skrivas ut. Endast ett svarsmeddelande kunde skrivas ut under varje iteration då endast ett svarsmeddelande togs ut från kön. Direkt efter att svarsmeddelandet hade skrivits ut på utskriftsfönstret i användargränssnittet skrevs svarsmeddelandet även till en loggfil.
2) *”Send single message”* hämtade först in mottagarnodens adress och sedan hämtades indatan som användaren hade angivit i användargränssnittets olika datafält. Sedan skickades ett UDS-förfrågningsmeddelande enligt den inhämtade informationen.
3) *”Run test cases”* hämtade först in mottagarnodens adress och sedan hämtades alla UDS-förfrågningarna från en fördefinierad fil. Sedan skickades ett UDS-förfrågningsmeddelande för varje UDS-förfrågning som hämtades från filen.
4) *”Exit”* stängde ned alla öppna anslutningar och stängde sedan av programmet.



Figur 20. Huvudprogrammets oändliga eventbaserade loop. I varje iteration kontrollerar loopen om något nytt event har skett och utför den eventuellt efterfrågade funktionaliteten. Eventet "Send single message" och "Run test cases" följde kontrollen som visades i figur 18.

Programvaruutvecklingen utfördes med Kvaser Leaf Light HS och PCAN-USB FD inkopplade i var sin dator. Datorn som var ihopkopplad med PCAN-USB FD användes som mottagarnod och körde programmet PCAN-VIEW för att lyssna på trafiken över CAN-bussen medan datorn som körde CanDoRequests använde Kvaser Leaf Light HS. Med denna utvecklingsmetod kunde bara UDS-förfrågningsmeddelandet analyseras. PCAN-VIEW är inte en ECU med UDS-protokollen implementerat och kunde därmed inte returnera några svarsmeddelanden. PCAN-VIEW kunde däremot användas för att kontrollera att UDS-förfrågningsmeddelanden kunde skickas från CanDoRequests och att det såg korrekt ut teoretiskt. Resultatet från två UDS-förfrågningsmeddelanden uppfångade av PCAN-VIEW visas i figur 21.



Figur 21. Det första UDS-förfrågningsmeddelandet hade mottagaradressen 0x456, UDS-tjänsten 0x14 och dataposten 0xFFFFFF. Det andra UDS-förfrågningsmeddelandet hade mottagaradressen 0x678, UDS-tjänsten 0x22 och DID-värdet 0x1234. Båda UDS-förfrågningarna skickades från CanDoRequests, 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 då den inte hade processat allt data genom ett transportlager med ISO-TP implementerat, därmed visades även PCI. PCI för det första UDS-förfrågningsmeddelandet var 0x04 där 0x0 angav att det är ett SF och 0x4 angav att UDS-förfrågningens storlek var 4 byte. PCI för det andra UDS-förfrågningsmeddelandet var 0x03 där 0x0 angav att det är ett SF och 0x3 som angav UDS-förfrågningens storlek var 3 byte. Teoretiskt sett såg allting korrekt ut.

När PCAN-VIEW kunde verifiera att korrekt data togs emot kunde CanDoRequests testas mot en bil. Uppkopplingen till en bil skedde genom att ansluta en dator till ett gränssnitt som var kopplad till CAN-bussen genom specialdesignat kablage. Det specialdesignade kablaget ska inte ge någon funktionell skillnad jämtemot att koppla upp gränssnittet till CAN-bussen genom OBD II-uttaget i en bil. Under denna del av programvaruutvecklingen kunde responsen från UDS-förfrågningsmeddelandet utvärderas. För att kunna 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 togs emot i form av byte. Med dessa byte 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 inte gör utan de returnerade byten från ett svarsmeddelande behövde avkodas genom olika avkodningsklasser som var beroende av hur stor dataposten förväntades att 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 avgöra om CanDoRequests var snabbare än CarDiagnosticsProgram123 utförde båda programvarorna samma experiment. Den snabbaste tiden som CarDiagnosticsProgram123 lyckades prestera användes som referenstid. Referenstiden togs med Vector VN1610 gränssnittet som användes tillsammans med CarDiagnosticsProgram123 och var den hårdvaru- och mjukvarukombination (HV/MV-kombination) som Syntronic använde vid testning. Två andra HV/MV-kombinationer testades, dessa var CanDoRequests med PCAN-USB FD och CanDoRequests med Kvaser Leaf Light HS. Syftet med att utföra testningen med båda kombinationerna var att undersöka hur stor tidsskillnad det kunde vara med olika gränssnitt då CanDoRequests inte hade stöd för Vectors gränssnitt och kunde därmed inte ha samma hårdvara som CarDiagnosticsProgram123. De olika gränssnitten kunde ha påverkat resultatet och det var viktigt att fastslå om valet av gränssnitt hade någon påverkan.

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

1) Ett tidtagarur startades när programmet startades.
2) Ett enskilt UDS-förfrågningsmeddelande skickades. Det fanns ett villkor och det var att det skulle vara samma UDS-förfrågning för alla tre HV/MV-kombinationer i samma testomgång. Värdena kunde dock skilja sig åt mellan varje testomgång.
3) När 2) fått sitt svar kördes filen med Φ UDS-förfrågningar.
4) När 3) fått sina svar skickades ett enskilt UDS-förfrågningsmeddelande med samma villkor gällande UDS-förfrågningens värde som angavs i 2).
5) När 4) fått sitt svar stoppades tidtagaruret.

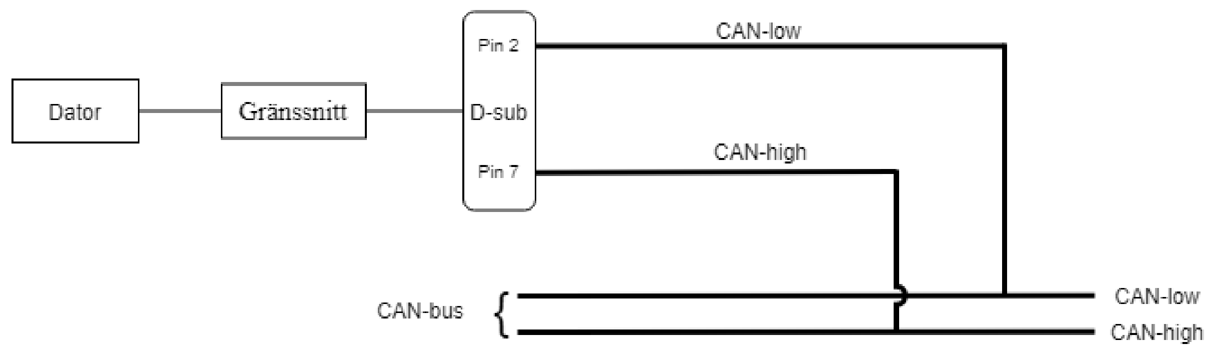
För delmoment 2), 3), och 4) togs deltider för att få en mer detaljerad bild över var den eventuella tidsvinsten kunde finnas. Tabell 1 visar bedömningsmallen som användes vid varje testomgång. Tidtagaruret som användes var Catiga CG-503.

HV/MV-kombination: ____________________		Omgång: _____
Event:	Tid vid event slut:	UDS-förfrågningens värde
Första enskilda UDS-förfrågningsmeddelandet	_________s	0x___________
Kör testfallen från filen automatiskt	_________s	UDS-förfrågning från den fördefinierade textfilen.
Andra enskilda UDS-förfrågningsmeddelande	_________s	0x___________

Tabell 1. Bedömningsmallen som användes vid varje testomgång. UDS-förfrågningen för delmomenten ska ha samma värde för alla tre HV/MV-kombinationerna i varje testomgång.

Alla testomgångar för de tre HV/MV-kombinationerna utfördes i samma bil. Bilen som testades var hemligstämplad då den vid testtillfället inte hade släppts på marknaden. Testen utfördes när bilen var avstängd med tändningen påslagen och alla HV/MV-kombinationer kopplades upp på samma anslutningspunkt i bilen. Figur 22 visar hur gränssnittet kopplades mellan en dator och direkt till CAN-bussen

i bilen genom det specialdesignade kablaget. Testningen utfördes av en av Syntronics testare som hade en bred erfarenhet av CarDiagnosticsProgram123. Testaren fick en genomgång om hur CanDoRequests fungerade innan testerna genomfördes.



Figur 22. Gränssnittets USB-kontakt kopplades till en dator och D-sub-kontakten kopplades till det specialdesignade kablaget som var direkt kopplat till bilens CAN-buss.

5 Resultat

Följande kapitel presenterar den resulterande programvaran och dess prestanda i form av tid jämtemot företagets nuvarande lösning med CarDiagnosticsProgram123.

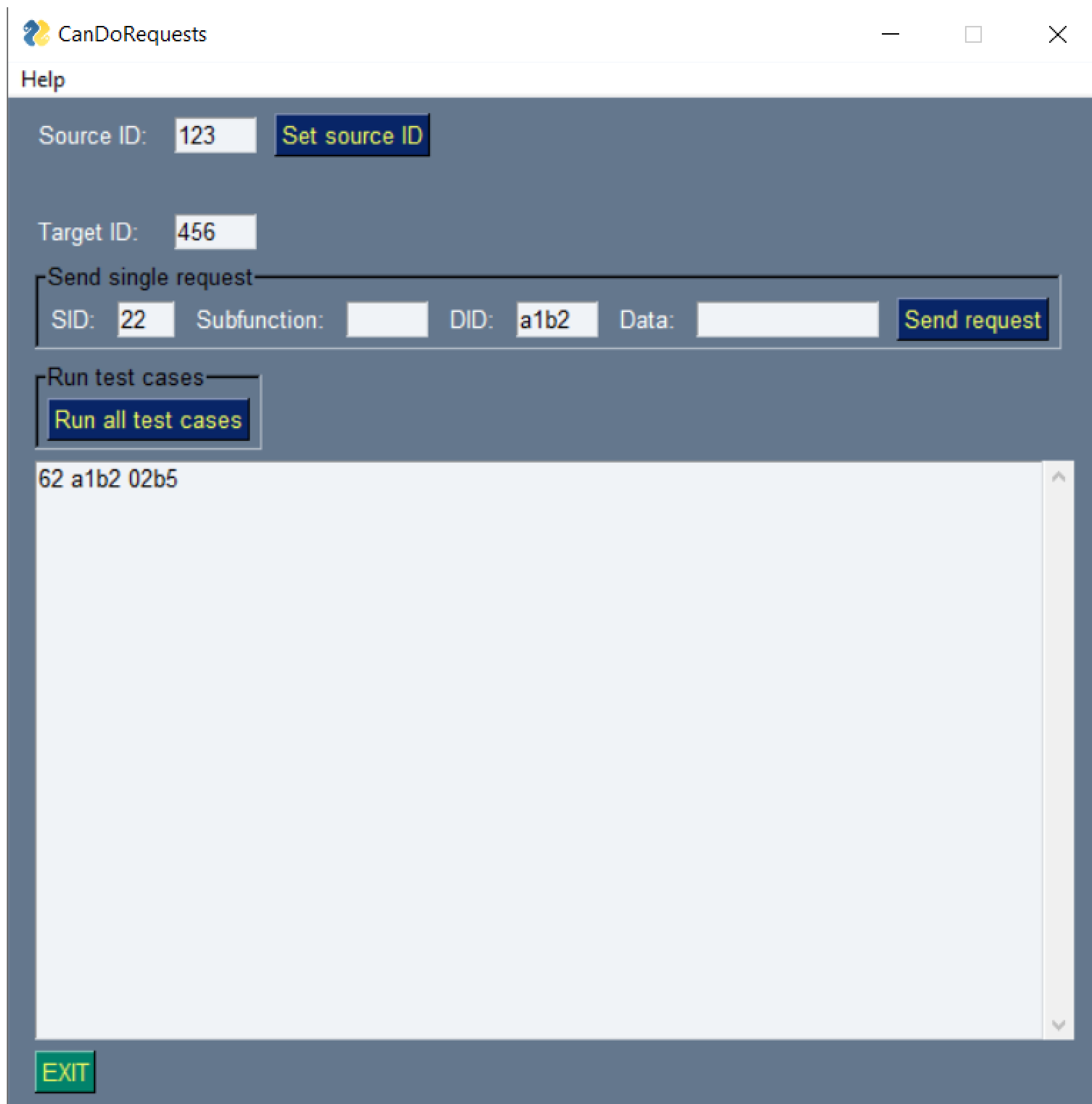
5.1 Implementation

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

- python-can är ett generellt CAN-bibliotek. Alternativet hade varit att använda gränssnittstillverkarnas egna bibliotek, men då hade programvaran inte kunnat bli tillräckligt generellt så att gränssnitt från olika tillverkare hade kunnat användas. Gränssnittstillverkarnas bibliotek behövdes dock för att kunna skapa bussanslutningen, men det var det enda de biblioteken användes till.
- För att implementera UDS fanns det två befintliga Pythonbibliotek, Python-uds och udsoncan. Valet föll på udsoncan då det var mer välutvecklat och hade även implementerat fler UDS-tjänster. Detta ansågs vara positivt då det möjliggjorde en större vidareutvecklingspotential för CanDoRequests.

Figur 23 visar programvarans grafiska användargränssnitt. Funktionaliteten som implementerades var följande:

- Avsändarnodens adress (”Source ID”) sattes med ett knapptryck för att poängtera att värdet bara behövde sättas en gång under programmets aktiva session. Om gränssnittet behövde kopplas till en ny anslutningspunkt borde programmet stängas av så att alla öppna anslutningar avslutades korrekt. Därefter kunde programmet startas igen när det var anslutet till den nya anslutningspunkten. Det gick dock att byta avsändarnodens adress under programmets körtid utan att programmet kraschade.
- Mottagarnodens adress (”Target ID”) hämtades in vid varje enskild UDS-förfrågning och en gång vid varje tillfälle som alla testfall skulle köras från filen.
- För att illustrera de två olika möjligheterna och vilket data som skulle 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” användes av båda möjligheterna och angavs i ett fält utanför båda ramarna.
- Svarsmeddelandet från UDS-förfrågningen skrevs ut på den vita utskriftsrutan.
- Samma svarsmeddelande skrevs till en textfil direkt efter att det hade skrivits ut på utskriftsrutan.
- Under utskriftsrutan fanns ”Exit”-knappen som stängde ned alla öppna anslutningar och sedan avslutade programmet.
- Hjälpmenyn (”Help”) öppnade ett nytt fönster med information om programmet.



Figur 23. DID-värdena och värdena i dataposten var konfidentiella, så detta är ett exempel på en UDS-förfrågning och ett svar med fabricerade värden. Ett UDS-förfrågningsmeddelande skickades där avsändarnodens adress var 0x123 och där mottagarnodens adress var 0x456. En enskild UDS-förfrågning skickades där vald UDS-tjänst var 0x22 och DID-värdet var 0xa1b2. Då UDS-tjänsten inte stöder subfunktioner eller dataposter lämnades de fälten tomma. Svarsmeddelandets data skrevs ut till den vita utskriftsrutan och till en loggfil. Svaret var positivt då SID-värdet 0x62 returnerades. Nästkommande två bytes var en kopia på UDS-förfrågningens DID-värde. Resterande bytes var den returnerade dataposten från ECU:n. I denna UDS-förfrågning returnerades 0x02b5.

5.2 Utvärdering

Referenstiden och delmomentens tidstämplar från CarDiagnosticsProgram123 presenteras i tabell 2 och var den snabbaste tiden som CarDiagnosticsProgram123 presterade under 10 testomgångar. Referenstiden togs under testomgång 8. De resulterande tiderna för de två andra HV/MV-kombinationerna presenteras både som mediantid och som genomsnittlig tid i tabell 3 och i tabell 4.

Vector VN1610 och CarDiagnosticsProgram123	Snabbast tid (s)	Tidsskillnad från föregående tidsstämpel. (s)
Första enskilda UDS-förfrågningsmeddelandet	29	-
Kör testfallen från filen automatiskt	40	11
Andra enskilda UDS-förfrågningsmeddelandet	47	7
Referenstid	47	-

Tabell 2. Tiderna som CarDiagnosticsProgram123 presterade under testomgång 8. Det är referenstiden på 47 sekunder som CanDoRequests ska vara snabbare än. Delmomentens tidsstämplar angavs för att kunna identifiera var en eventuell tidsvinst kunde finnas.

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

Tabell 3. Mediantiden och den genomsnittliga tiden för kombinationen med PCAN-USB FD och CanDoRequests. Delmomentens tidsstämplar anges för både mediantiden och för den genomsnittliga tiden.

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

Tabell 4. Mediantiden och den genomsnittliga tiden för kombinationen med Kvaser Leaf Light HS och CanDoRequests. Delmomentens tidsstämplar anges för både mediantiden och för den genomsnittliga tiden.

Då 10 testomgångar utfördes kommer mediantiden att användas som jämförelsetid då den anses vara mest representativ. Anledningen till att mediantiden ansågs vara mest representativ var att mänsklig interaktion påverkade resultatet och eventuella ”icke-optimala” testomgångar med felaktigt inmatat data påverkade den genomsnittliga tiden. Sluttiderna för varje HV/MV-kombination visas i figur 25 som finns i bilaga 2. Nedanstående observationer kan hämtas från tabellerna 2 – 4.

Vid jämförelse mellan tabell 2 och tabell 3 observeras följande:

- PCAN-USB FD och CanDoRequests var 12.5 sekunder snabbare än referenstiden.
- CarDiagnosticsProgram123 var två sekunder snabbare mellan delmomenten ”Första enskilda UDS-förfrågningsmeddelandet” och ”Kör testfallen från filen automatiskt” jämfört med PCAN-USB FD och CanDoRequests
- PCAN-USB FD och CanDoRequests var en halv sekund snabbare mellan delmomenten Kör testfallen från filen automatiskt” och ”Andra enskilda UDS-förfrågningsmeddelandet” jämfört med CarDiagnosticsProgram123.

Vid jämförelse mellan tabell 2 och tabell 4 observeras följande:

- Kvaser Leaf Light HS och CanDoRequests var 13 sekunder snabbare än referenstiden.
- Kvaser Leaf Light HS och CanDoRequests var en halv sekund snabbare mellan delmomenten ”Första enskilda UDS-förfrågningsmeddelandet” och ”Kör testfallen från filen automatiskt” jämfört med CarDiagnosticsProgram123.
- CarDiagnosticsProgram123 var en sekund snabbare mellan delmomenten ”Kör testfallen från filen automatiskt” och ”Andra enskilda UDS-förfrågningsmeddelandet” jämfört med Kvaser Leaf Light HS och CanDoRequests.

Vid jämförelse mellan tabell 3 och tabell 4 observeras följande:

- Kvaser Leaf Light HS var en halv sekund snabbare än PCAN-USB FD.
- Kvaser Leaf Light HS var två och en halv sekunder snabbare mellan delmomenten ”Första enskilda UDS-förfrågningsmeddelandet” och ”Kör testfallen från filen automatiskt” jämfört med PCAN-USB FD.
- PCAN-USB FD var en och en halv sekund snabbare mellan delmomenten ”Kör testfallen från filen automatiskt” och ”Andra enskilda UDS-förfrågningsmeddelandet” jämfört med Kvaser Leaf Light HS.

6 Diskussion

Följande kapitel diskuterar resultatet, metoden och arbetet i ett vidare sammanhang.

6.1 Resultat

Att CanDoRequests var snabbare än CarDiagnosticsProgram123 var föga överraskande. CarDiagnosticsProgram123 hade flera funktioner som kunde konfigureras och det tog nära 30 sekunder innan alla nödvändiga filer hade laddats in och programmet kunde användas. CarDiagnosticsProgram123 är en mer komplett programvara men om en användare ska utföra fordonsdiagnostik enligt UDS-protokollen är programvaran för avancerad vilket medför en viss påverkan på programmets totala körtid. Genom att konfigurera CanDoRequests vid uppstart utan användarinteraktion utöver att sätta avsändarnodens adress kunde ungefär 15 sekunder sparas in. Att CanDoRequests även skulle vara snabbare i vissa delmoment var överraskande. CarDiagnosticsProgram123 utförde UDS-förfrågningarna och fick svaren snabbare än CanDoRequests men krävde mer användarinteraktion då användaren behövde navigera mellan olika menyer för att kunna utföra olika funktioner. Exempelvis tog det 9–14 sekunder att köra filen med Φ UDS-förfrågningar och sedan få svar med CanDoRequests medan CarDiagnosticsProgram123 kunde utföra samma funktionalitet på 2–3 sekunder. Trots detta låg den totala tiden för det delmomentet inom samma tidsintervall för alla tre HV/MV-kombinationerna. Det berodde på att CarDiagnosticsProgram123 krävde att användaren navigerade mellan olika menyer för att kunna köra filen med Φ UDS-förfrågningar medan CanDoRequests kunde utföra det med ett knapptryck. I det här scenariot var dock tidsjämförelsen möjligtvis orättvis. Detta då CarDiagnosticsProgram123 tillät användaren att välja vilken fil med testfall som skulle köras medan CanDoRequests bara kunde köra en fördefinierad fil. Det fanns däremot bara en fil med testfall, men om flera olika filer med testfall skapas framöver måste CanDoRequests också låta användaren att välja vilken fil med testfall som ska köras. I det scenariot bör CarDiagnosticsProgram123 vara betydligt snabbare i det delmomentet då den kan köra testfallen på en femtedel av tiden det tar för CanDoRequests.

Bilaga 1 innehåller svaren från en utvärdering som testaren besvarade efter att alla testomgångar med de tre HV/MV-kombinationerna var avklarade. Dessa svar gav mer funktionsspecifika detaljer gällande hur CanDoRequests presterade jämfört med CarDiagnosticsProgram123. Ett av de största problemen med CanDoRequests var att avsändarnodens adress och mottagarnodens adress behövde anges manuellt. Det var något som CarDiagnosticsProgram123 inte krävde, utan dessa värden sattes i bakgrunden utan användarinteraktion. Testaren visste inte var avsändarnodens och mottagarnodens adresser kunde hittas utan att använda CarDiagnosticsProgram123 tillsammans med CANalyzer[4] som är ett program från Vector som lyssnar på CAN-bussen och hämtar in data från CAN-meddelanden, inklusive adresserna för de olika noderna. Det är möjligt att nodernas 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 prövats då CanDoRequests endast testades i en bil. Ett önskemål från testaren var att CanDoRequests skulle spara avsändarnodens adress och mottagarnodens adress när programmet stängdes av och sedan ladda in dessa adresser automatiskt när programmet startades. Det 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 funktion med förbättringspotential var att låta användaren ange filnamnet på loggfilen själv. CanDoRequests skapade filnamnet automatiskt genom att modifiera ett DateTime-objekt i Python som gav det nuvarande datumet och tiden. Detta ansågs kunna 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 liten. Filnamnen sorterades även kronologiskt i mappen som sparade loggfilerna vilket gjorde det lätt att hitta den senaste loggfilen. Testaren föredrog däremot att ha möjligheten att namnge loggfilen

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

själv och att nuvarande filnamn med det modifierade DateTime-objektet kunde få vara standardfilnamnet om inget eget filnamn angivits.

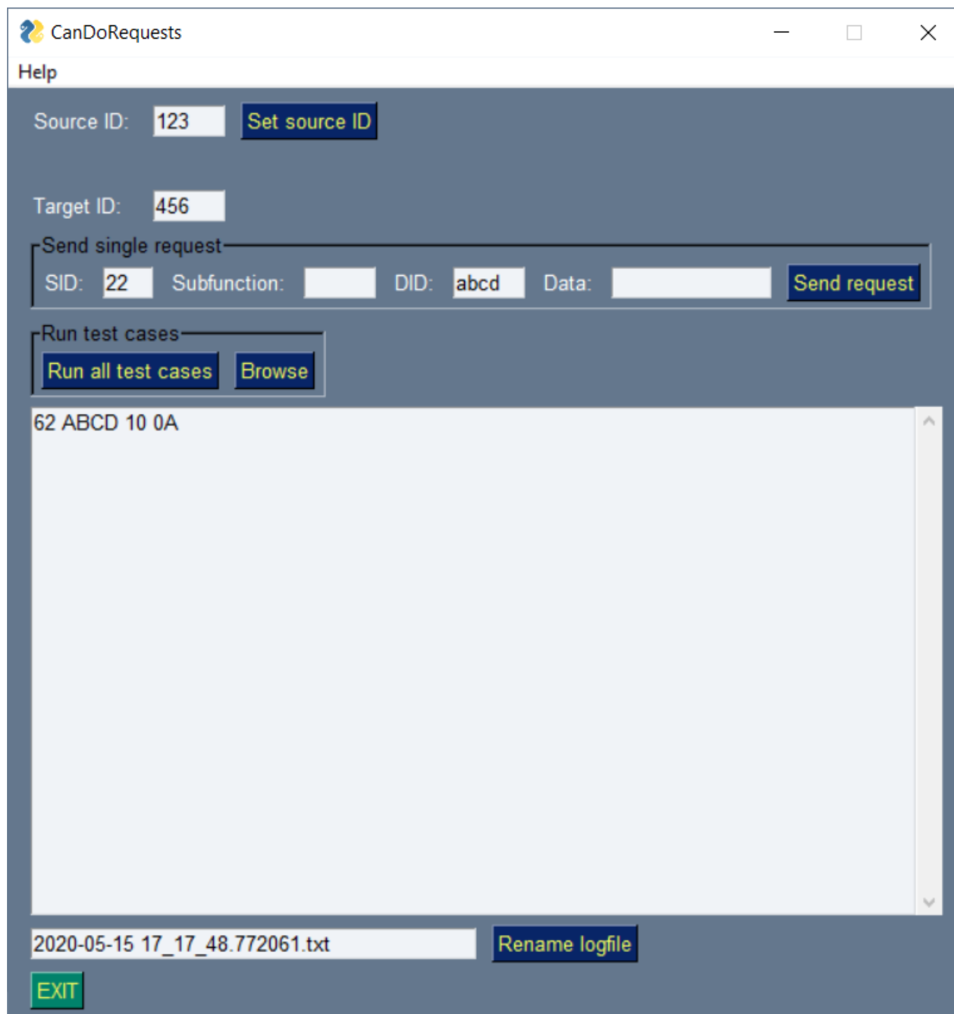
Fördelen med CanDoRequests var att den var både snabbare och enklare att arbeta med. Det berodde på att funktionaliteten för att skicka ett enskilt UDS-förfrågningsmeddelande och att köra filen med de fördefinierade UDS-förfrågningarna låg bredvid varandra i användargränssnittet. Användaren behövde därmed inte navigera mellan olika menyer för att byta mellan olika funktioner.

En aspekt som inte togs med i utvärderingen men som uppskattades av testaren var tiden det tog att stänga ned programmet. CarDiagnosticsProgram123 krävde att användaren aktivt sparade ned data till en loggfil om användaren ville behålla det. Även nedstängningen hade viss interaktion utöver knapptrycket för att stänga ned programmet vilket innebar ytterligare några sekunders körtid. CanDoRequests sparade istället ned samma data som skrevs ut på utskriftsfönstret till en loggfil under programmets körtid och vid avslut stängdes programmet ned omgående. Anledningen till att denna aspekt inte inkluderades i testningen var att det inte var representativt för hur CarDiagnosticsProgram123 användes vid testning.

En förbättrad version av CanDoRequests skapades som baserades på svaren från bilaga 1 och visas i figur 24. Förbättringarna som gjordes var:

- Användaren gavs möjligheten att namnge loggfilen själv. Det skapades alltid en loggfil med ett standardfilnamn som kunde ändras när som helst under programmets körtid.
- Användaren gavs möjligheten att välja vilken fil med fördefinierade UDS-förfrågningar som skulle användas.
- En fil, ”history.txt”, skapades och sparade ned ”Source ID”, ”Target ID” och vilken fil med de fördefinierade UDS-förfrågningarna som hade körts. Värdena från denna fil laddades sedan in vid uppstart av programmet vilket innebar att den senaste sessionens värden sattes som standardvärden vid programmets nästa uppstart.
- Svaren från UDS-förfrågningarna skrevs ut som versaler.

Det var denna version som levererades till Syntronic AB. Då fler värden laddades in vid uppstart av CanDoRequests borde denna version kunna leverera snabbare tider då användaren inte längre behöver ange ”Source ID” och ”Target ID” vid varje uppstart. Dessa värden var alltid oförändrade då en ECU diagnoserades från en anslutningspunkt och dessa värden var därmed konstanta över alla test.



Figur 24. Version 2 av CanDoRequests innehöll förbättringar som baserades på utvärderingen från testaren. ”Source ID”, ”Target ID” och filen med fördefinierade UDS-förfrågningarna var redan satta när programmet startade och behövde inte anges såvida inga ändringar hade skett sedan den senaste sessionen. Bredvid knappen ”Run all test cases” finns det en ”Browse”-knapp som öppnar mappen som programfilerna finns i och tillåter användaren att byta filen med de fördefinierade UDS-förfrågningarna. Allt data i utskriftsfönstret och i loggfilen är numer i versaler. Användaren kan även byta namn på loggfilen, nuvarande namn som visas i fältet ovanför ”Exit”-knappen är standardfilnamnet på filen.

6.2 Metod

Kandidatarbetet började med en teoretisk undersökning för att få en förståelse för hur CAN och UDS fungerade. Variationen av pålitliga källor som beskrev CAN gjorde det enkelt att hitta texter som beskrev CAN på ett pedagogiskt sätt och en grundläggande förståelse för CAN kunde uppnås. UDS är däremot ett mer nischat område och en överblicksartikel eller bok om UDS kunde inte hittas. Artiklarna som hittades och behandlade UDS beskrev det kortfattat och krävde att läsaren hade en grundförståelse om hur UDS fungerar. Valet föll på att börja läsa UDS-protokollet ISO14229 [13] vilket, med facit i hand, inte är att rekommendera om man inte har en grundläggande kunskap om UDS. Icke-vetenskapliga källor som PiEmbSysTech[5], 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 kunde därmed ha underlättat undersökningen av ISO14229 [13]. Med kunskapen från bloggen blev UDS-protokollen och de vetenskapliga artiklarna som behandlade UDS mycket lättare att förstå och en djupare förståelse kunde hämtas från de.

[5] https://piembsystech.com/uds-protocol/

UDS-protokollen som teorin baserades på var utdaterade. Det finns nyare UDS-protokoll som jag däremot inte hade tillgång till. ISO15765-3 [7] är från 2004 och blev ersatt av ISO14229-3 år 2012. Informationen i ISO15765-3 [7] är inte felaktig men nyare information finns i ISO14229-3. Detsamma gällde ISO14229 [13] som är från 2006 och blev ersatt av ISO14229-1:2013 som i sin tur blev ersatt av ISO14229-1:2020. Även i detta fall är informationen i ISO14229 [13] inte felaktig men nyare och mer aktuell information finns.

Covid-19-pandemin medförde restriktioner för personalen på Syntronic. De anställda skulle arbeta hemifrån och detta inkluderade även mig. Detta medförde viss problematik då potentiella lösningar inte kunde testas i en bil kontinuerligt utan majoriteten av utvecklingen skedde hemifrån med två gränssnitt kopplade mellan två datorer. När en potentiell lösning fanns bokades en tid in med en utvecklare från Syntronic för att få möjlighet att testa lösningen mot en ECU i en bil. Programvaran testades mot en bil två gånger i 1–3 timmars sessioner innan UDS-tjänsten 0x22 fungerade korrekt. Denna arbetsmetod var inte optimal och en bättre slutprodukt med fler implementerade UDS-tjänster hade varit möjligt med mer utvecklingstid i en bil. Med ett tidigare fokus på programvaruimplementationen hade det kanske varit 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 UDS-tjänsten blev implementerad. Andra UDS-tjänster var en bonus och ett försök att implementera två andra UDS-tjänster, ClearDTCInformation (0x14) och ReadDTCInformation (0x19), gjordes men misslyckades. Med mer utvecklingstid i en bil hade de kanske ha kunnat implementerats.

Nackdelen med att använda udsoncan var att varje UDS-förfrågningsmeddelande var ett blockerande funktionsanrop. Hur länge blockeringen fortgick innan ett timeout-undantag gavs kunde anges, men det hade varit att föredra om ett UDS-förfrågningsmeddelande hade skickats och sedan kunde svarsmeddelandet ha tagits emot i en ny bakgrundstråd. UDS-förfrågningsmeddelanden har en låg prioritet på CAN-bussen och det kan dröja någon sekund innan ett svar fås. Det här var inte något större bekymmer när ett enskilt UDS-förfrågningsmeddelande skickades, men när filen som innehöll Φ UDS-förfrågningar skulle skickas blev det märkbart då inget nytt UDS-förfrågningsmeddelande kunde skickas innan det tidigare UDS-förfrågningsmeddelandet hade fått sitt svar. Att gå igenom filen med Φ UDS-förfrågningarna tog mellan 9 och 14 sekunder och programfönstret var fryst under denna tidsperiod. Ett försök till att lösa detta gjordes genom att implementera en trådpool och en ”arbetarfunktion” som utförde ett antal UDS-förfrågningar samtidigt (*concurrency* ej att förväxla med parallellt). Funktionaliteten testades mellan två datorer där ena datorn körde PCAN-VIEW. Alla UDS-förfrågningsmeddelanden skickades korrekt på en femtedel av tidsåtgången det tog utan trådpoolen. Om denna implementation hade fungerat med en dator inkopplad på bilens CAN-buss skulle det ha tagit ungefär två sekunder att utföra alla UDS-förfrågningsmeddelanden. Tyvärr fungerade det inte. När implementationen testades i en bil fick 5 av Φ UDS-förfrågningsmeddelanden svar, övriga UDS-förfrågningsmeddelanden fick olika felmeddelanden. Det kan möjligtvis bero på att udsoncan släppte det tidigare UDS-förfrågningsmeddelandets anslutning när ett nytt UDS-förfrågningsmeddelande skickades. Det resulterade i att det tidigare UDS-förfrågningsmeddelandets förväntade svarsmeddelande inte kunde tas emot av klienten då anslutningen redan hade avbrutits när det nya UDS-förfrågningsmeddelandet hade skickats.

Fördelen med att använda udsoncan var att det möjliggjorde att programmet kunde bli klart i tid. Det är möjligt att skapa en egen implementation med python-can där svarsmeddelanden tas emot i en bakgrundstråd. Det var dock inte möjligt tidsmässigt under det här kandidatarbetet då ett stort fokus behövdes läggas på den teoretiska genomgången.

6.2.2 Utvärdering:

Det var ingen maskinprecision på tidtagningen. En inbyggd timer kunde startas och stoppas automatiskt i CanDoRequests, men CarDiagnosticsProgram123 behövde konfigureras manuellt och en automatiskt styrd timer kunde inte implementeras. Det mest rättvisa var därmed att använda samma tidtagningsmetod för båda programvarorna, vilket innebar en manuell tidtagning med ett tidtagarur. Tidtagning med ett 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 ett UDS-förfrågningsmeddelande kan skickas och sedan få sitt svarsmeddelande. UDS-meddelanden är i CAN 2.0B-formatet och har en låg prioritet på CAN-bussen. Det innebär att om många högre prioriterade CAN-meddelanden har skickats kommer UDS-meddelandena att behöva vänta för att få tillgång till CAN-bussen och denna trafik kan användaren inte påverka när bilen är avstängd och när reglage inte har manipulerats.

Att utföra 10 testomgångar med varje HV/MV-kombination ansågs räcka i och med att den långsammaste tiden utan felaktig inmatning från de två övriga HV/MV-kombinationerna var snabbare än referenstiden. Det ansågs därmed vara tillräckligt för att kunna svara på frågan i problemformuleringen. Testet kunde även avgöra om gränssnitten påverkade resultatet. Skillnaderna var inte stora men Kvaser Leaf Light HS kunde utföra Φ UDS-förfrågningar från filen ungefär 2.5 sekunder snabbare än PCAN-USB FD. Däremot påverkades inte den totala mediantiden för testomgångarna nämnvärt och därmed anses inte gränssnitten ha påverkat resultatet.

6.3 Arbetet i större sammanhang

Bilar påverkar både miljön och människor. Människor dör av både direkt och indirekt påverkan från bilar såsom kollisioner och föroreningar. Om programvaran som skapades inte är ordentligt testad innan användning kan både miljön och människors hälsa påverkas. Om programvaran ger felaktig respons, antingen genom en felaktigt utförd UDS-förfrågning eller om ett svar tolkades felaktigt kan det få ödesdigra konsekvenser för miljön, människan eller bådadera. Med en konfigurationsfil från en fordonstillverkare kan exempelvis bilens utsläpp läsas av och felaktig respons kan ge falska svar, både positiva och negativa. 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 fordonstillverkaren om bilar behöver återkallas. I värsta fall kan felaktiga svar resultera i dödsfall om bromsarna inte fungerar som de skall. I dessa scenarion påverkas även företaget som utförde testningen negativt om det kan påvisas att testföretaget brustit i sin testning.

7. Slutsats

Det här kandidatarbetet undersökte hur UDS kunde implementeras i en klient som utförde fordonsdiagnostik över CAN-bussen. En problemformulering formulerades och tre önskvärda krav fanns för programvaran som skapades. Problemen som skulle besvaras var:

1) Hur kan en mjukvara som ska utföra fordonsdiagnostik enligt UDS-protokollen konstrueras så att total körtid, inklusive uppstart av mjukvaran, är snabbare än referenstiden som sattes av CarDiagnosticsProgram123?
Denna fråga besvarades i kapitel 4.1.2 och i kapitel 5.1. I denna fråga finns det dock inget entydigt svar och den resulterande lösningen hade förbättringspotential. Konstruktionen av programvaran kunde utföras på andra sätt och detta arbete gav en lösning på problemet som skulle besvaras.

2) Mjukvaran ska kunna kommunicera på CAN-bussen med flera olika gränssnitt.
Då både Kvasers och PEAKs gränssnitt kunde användas tillsammans med CanDoRequests anses det här önskvärda kravet vara tillgodosett. Det ska även gå att använda andra gränssnitt som exempelvis Vector. För att använda andra gränssnitt måste däremot gränssnittets specifika mjukvara installeras på datorn, och sedan måste en bussanslutning för detta gränssnitt implementeras i backend.py.

3) Mjukvaran ska kunna live-visualisera svarsmeddelanden som kommer från UDS-förfrågningsmeddelandena.
Det här önskvärda kravet kunde tillgodoses till viss del. Data kunde visualiseras på utskriftsfönstret, men när filen med Φ UDS-förfrågningar skulle köras kom alla svar samtidigt när alla UDS-förfrågningsmeddelanden hade fått sina svar vilket inte anses vara live-visualisering.

4) Mjukvaran ska kunna visualisera de automatiskt sparade svarsmeddelandena.
Genom att skriva samma data som illustrerades på utskriftsfältet till en loggfil kunde det här önskvärda kravet tillgodoses.

Nyttan för Syntronic var att de fick en programvara som kunde utföra fordonsdiagnostik med en snabbare total körtid än CarDiagnosticsProgram123. Det gällde dock bara för den konfigurationsfilen 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. Det är dock inget omfattande arbete utan det tar ungefär 20–30 minuter och behöver endast utföras en gång. En konfigurationsfil från en fordonstillverkare kan vara aktuell i flera år så dessa 20–30 minuter anses vara försumbara. Om konfigurationsfilerna från fordonstillverkaren hade ändrats mer frekvent hade detta behövt lösas med en automatiserad konfiguration.

Resultatet anses hålla även om utvärderingen endast utfördes på en bil. Den största tidsvinsten låg under initieringen av programmet innan det första UDS-förfrågningsmeddelandet hade skickats, och denna tid är fordonsoberoende. De olika delmomenten tog ungefär lika lång tid för CanDoRequests med Kvaser Leaf Light HS och PCAN-USB FD som de snabbaste tiderna för CarDiagnosticsProgram123 med Vector VN1610. Därmed anses CanDoRequests även kunna prestera bra utöver tidsvinsten vid programmets uppstart.

Som framtida arbete skulle korrelationen mellan ett UDS-förfrågningsmeddelande och det mottagna svarsmeddelandet kunna undersökas och förbättras. Ett försök att skapa en trådpool gjordes och misslyckades men med mer tid kan det kanske lösas.

Litteratur

[1] W Dubitzky, T Karacay, *CAN – From its early days to CAN FD*, Can Newsletter, http://www.can-newsletter.org/uploads/media/raw/6b2563046de889524638725c61627661.pdf (hämtad 2020-02-09).

[2] M. Salcianu, C. Fosalau, "A new CAN diagnostic fault simulator based on UDS protocol", In: 2012 International Conference and Exposition on Electrical and Power Engineering, 2012. DOI: 10.1109/ICEPE.2012.6463580

[3] 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. DOI: 10.2991/icmia-16.2016.113

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

[5] V. Voss, "A Comprahensible Guide to Controller Area Network", Copperhill Technologies, 2005, ISBN: 0-9765116-0-6

[6] C. Miller, C. Valasek, "Adventures in Automotive Networks and Control Units", IOActive, Technical white paper, 2014.

[7] ISO15765-3. "Road Vehicles – Diagnostics on Controller Area Networks (CAN) – Part 3: Implementation of unified diagnostic services (UDS on CAN)". ISO, Geneva Switzerland 2004

[8] S. Dekanic, R. Grbic, T. Maruna, I. Kolak, "Integration of CAN Bus Drivers and UDS on Aurix Platform", In: 2018 Zooming Innovation in Consumer Technologies Conference (ZINC), 2018. DOI: 10.1109/ZINC.2018.8448921

[9] Pazul, "Controller Area Network (CAN) Basics", Microship Technology Inc, 1999

[10] CAN in Automation (Cia), *CAN lower- and higher-layer protocols*, https://www.can-cia.org/can-knowledge/ (hämtad 2020-04-06).

[11] Kvaser, *CAN Bus Error Handling*, 2020, https://www.kvaser.com/about-can/the-can-protocol/can-error-handling/ (hämtad 2020-03-26).

[12] Kvaser, *Using termination to ensure recessive bit transformation*, https://www.kvaser.com/using-termination-ensure-recessive-bit-transmission/ (hämtad 2020-03-26).

[13] ISO14229. "Road Vehicles – Unified diagnostic services (UDS) – Specifications and requirements". ISO, Geneva Switzerland 2006

[14] S. Godavarty, S. Broyles, M. Parten, "Interfacing to the On-Board Diagnostics System", In: Vehicular Technology Conference Fall 2000 IEEE VTS Fall VTC2000. 52nd Vehicular Technology Conference (Cat. No.00CH37152), 2000. DOI: 10.1109/VETECF.2000.886162

[15] S. Kelkar, R. Kamal, "Adaptive Fault Diagnosis Algorithm for Controller Area Network", In: IEEE Transaction on Industrial Electronics (Volume 61, Isssue: 10, Oct. 2014), 2014. DOI: 10.1109/TIE.2013.2297296

[16] 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 (hämtad 2020-04-03).

[17] D. Hu, D. Hou, K. Guo, C. Sun, “Design and implementation of Diagnostic system for Integrated body Controller based on CAN bus”, In: 2019 Chinese Automation Congress (CAC), 2019. DOI: 10.1109/CAC48633.2019.8996400

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

[19] P. Kharche, M. Murali, G. Khot, “UDS Implementation for ECU I/O Testing”, In: 2018 3rd IEEE International Conference on Intelligent Transportation Engineering (ICITE), 2018. DOI: 10.1109/ICITE.2018.8492642

[20] EmbededCLogic, *UDS-Remote Activation Of Routine*, https://embedclogic.com/uds-protocol/uds-remote-activation-of-routine/ (hämtad 2020-04-27).

[21] 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 (hämtad 2020-05-02).

[22] PiEmbSysTech, *CAN-TP Protocol*, https://piembsystech.com/can-tp-protocol/ (hämtad 2020-04-22).

[23] Kvaser, *Kvaser Leaf Light HS v2*, https://www.kvaser.com/product/kvaser-leaf-light-hs-v2/ (hämtad 2020-03-26).

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

Bilaga 1. Utvärderingssvar

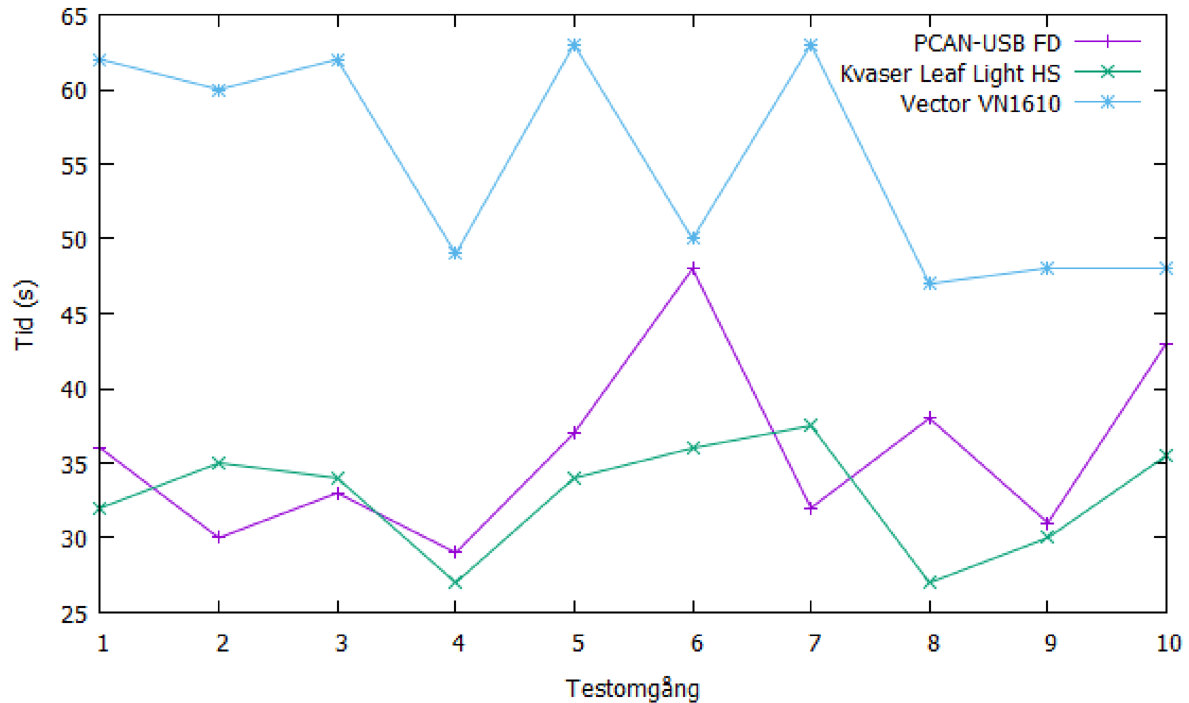
Två frågor ställdes efter att utvärderingen i kapitel 5.2 var klar. Nedan följer frågorna och svaren. Både frågorna och svaren utfördes muntligt och svaren antecknades samtidigt som de gavs.

1) Vilken funktionalitet från CarDiagnosticsProgram123 saknade du i CanDoRequests?
    - Testaren ville ha möjlighet att själv kunna namnge loggfilen när programmet var aktivt.
    - Testaren ville ha möjlighet att kunna välja vilken fil med testfall som skulle köras.
    - I CarDiagnosticsProgram123 behövde varken avsändarnodens eller mottagarnodens adress anges och testaren saknade detta i CanDoRequests.
    - Testaren föredrog att få svaren från UDS-förfrågningarna i versaler

2) Vad var bättre med CanDoRequests jämtemot CarDiagnosticsProgram123?
    - Både uppstart och nedstängning av CanDoRequests var betydligt snabbare än CarDiagnosticsProgram123.
    - Testaren uppskattade att funktionaliteten för att sända ett enskilt UDS-förfrågningsmeddelande och att köra filen med testfallen låg nära varandra på det grafiska användargränssnittet.
    - Det var bra att loggfilen skapades och sparade ned data automatiskt, men att kunna namnge filen själv istället för att få standardfilnamnet som skapades hade gjort det bättre.

Bilaga 2. Sluttiderna för alla testomgångar

Alla HV/MV-kombinationerna utförde 10 testomgångar och deras sluttider i varje testomgång visas i figur 25.



Figur 25. Sluttiderna för varje testomgång. I y-led visas tiden i sekunder och x-led visar testomgången tiden togs i. Den blåa linjen har kombinationen Vector VN1610 tillsammans med CarDiagnosticsProgram123. Den lila linjen har kombinationen PCAN-USB FD tillsammans med CanDoRequests och den gröna linjen visar tiderna för kombinationen med Kvaser Leaf Light HS tillsammans med CanDoRequests.

Anledningen till att tiderna varierar så mycket för CarDiagnosticsProgram123 var att det var enkelt att trycka fel när fordonskonfigurationsfilerna skulle väljas. Tre avvikande tider för PCAN-USB FD med CanDoRequests syns i testomgång 6, 8 och 10 och dessa tidsavvikelser berodde på felaktigt inmatat data i användargränssnittet. Man kan argumentera för att dessa testomgångar borde ha utförts igen, men då är frågan var gränsen ska dras för när en tid ansetts ha påverkats av den mänskliga interaktionen. Därmed togs beslutet att inte göra om testomgångarna med felaktigt inmatat data. Tiderna tagna med Kvaser Leaf Light HS tillsammans med CanDoRequests var förskonade från felaktigt inmatat data.