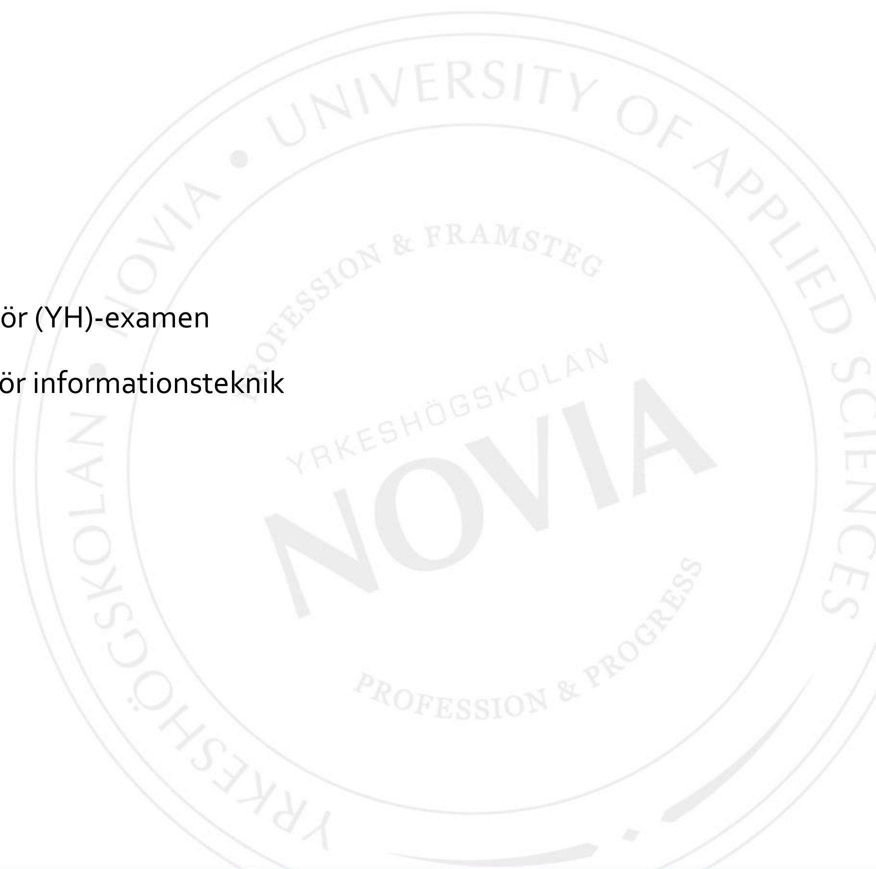
YRKESHÖGSKOLAN
NOVIA

Automatisk testning av CANopen

Matias Nordström

Examensarbete för ingenjör (YH)-examen

Utbildningsprogrammet för informationsteknik
Vasa och 2018



EXAMENSARBETE

Författare:	Matias Nordström
Utbildning och ort:	Informationsteknik, Vasa
Handledare:	Kaj Wikman

Titel: Automatisk testning av CANopen

Datum: 16.4.2018 Sidantal: 23

Abstrakt

Målet med detta examensarbete var att skapa automatiska tester för en CANopen-implementering. Genom att ersätta existerande manuella tester med automatiska tester var målet att spara tid och resurser. De skapade testerna skall användas i ett system för kontinuerlig integration där de kompletterar enhetstester för att förbättra täckningen av programkoden. Automatiska funktionella tester är en förutsättning för kontinuerliga leveranser av mjukvara.

Testerna skapades med programmeringsspråket Java och använder ett testramverk baserat på TestNG.

Resultatet blev en grunduppsättning av automatiska CANopen-tester som körs dagligen. Testerna verifierar att ingen kod ändring som kommit in har haft sönder existerande funktionalitet.

Språk: svenska Nyckelord: CANopen, testning

OPINNÄYTETYÖ

Tekijä:	Matias Nordström
Koulutus ja paikkakunta:	Tietotekniikka, Vaasa
Ohjaaja(t):	Kaj Wikman

Nimike: CANopenin automaattinen testaus

Päivämäärä: 16.4.2018 Sivumäärä: 23

Tiivistelmä

Tämän opinnäytetyön tavoitteena oli luoda automaattiset testit CANopen toteutukselle. Korvaamalla olemassa olevat manuaaliset testit automaattisilla testeillä pyrittiin säästämään aikaa ja resursseja. Luodut testit käytetään jatkuvaan integraatiojärjestelmään, jossa ne täydentävät yksikkötestejä ohjelmakoodin kattavuuden parantamiseksi. Automaattinen toiminnallinen testaus on edellytys ohjelmistojen jatkuvalle toimitukselle.

Testit luotiin Java-ohjelmointikielellä ja käytettiin TestNG: ään perustuvaa testauskehystä.

Tuloksena oli päivittäin käytössä oleva automatisoitu CANopen testien peruskokoelma. Testit vahvistavat, että koodin muutokset eivät riko olemassa olevia toteutuksia.

Kieli: ruotsi Avainsanat: CANopen, testaus

BACHELOR'S THESIS

Author:	Matias Nordström
Degree Programme:	Information Technology, Vasa
Supervisor(s):	Kaj Wikman

Title: Automatic testing of CANopen

Date: 16.4.2018 Number of pages: 23

Abstract

The goal of this Bachelor's thesis was to create automatic tests for a CANopen implementation. By replacing existing manual tests with automatic tests, the target was to save time and resources. The created tests will be executed in a continuous integration system where they complement unit tests to improve coverage of the program code. Automatic functional testing is a prerequisite for continuous software releases.

The tests were created with Java programming language and use a test framework based on TestNG.

The result was a core set of automated CANopen tests that are running daily. The tests verify that no committed code change has broken the existing functionality.

Language: swedish Key words: CANopen, testing

Innehållsförteckning

1 Inledning ... 1
1.1 Uppdragsgivare ... 1
1.2 Uppgift ... 1
2 Teori ... 2
2.1 CANopen ... 2
2.1.1 CAN fysiska skiktet ... 3
2.1.2 CAN datalänkskiktet ... 3
2.1.3 CANopen-applikationsskiktet ... 4
2.1.4 CANopen-enhetsmodell ... 4
2.1.5 Kommunikation ... 5
2.1.6 Process data object ... 6
2.1.7 Multiplex process data object ... 7
2.1.8 Service data object ... 7
2.1.9 Synchronization object ... 8
2.1.10 Time stamp object ... 9
2.1.11 Emergency object ... 9
2.1.12 Network management ... 9
2.1.13 Heartbeat ... 10
2.2 Automatiska tester ... 11
2.2.1 Typer av tester ... 11
2.2.2 Fördelar med automatiska tester ... 12
2.2.3 Utmaningar med automatiska tester ... 13
3 Teknik ... 13
4 Utförande ... 15
4.1 Definition av tester ... 15
4.2 Skapandet av testerna ... 15
4.2.1 Manuellt test av PDO ... 15
4.2.2 Automatiska testet av PDO ... 17
5 Resultat ... 20
6 Diskussion ... 21
7 Källförteckning ... 23

Förkortningar

ATF	Automatic test framework
CAN	Controller area network
CiA	CAN in Automation
COB	Communication object
EMCY	Emergency object
MPDO	Multiplex process data object
NMT	Network management
OSI	Open systems interconnection
PDO	Process data object
RTR	Remote transmission request
SDO	Service data object
SYNC	Synchronization object
TIME	Time stamp object

1 Inledning

I detta kapitel presenteras uppdragsgivaren samt uppgiften i korthet.

1.1 Uppdragsgivare

Wapice är ett snabbt växande IT-företag. Det grundades 1999 i Vasa och är privatägt. Huvudkontoret finns på Brändö i Vasa. Wapice har också kontor i Runsor, Tammerfors, Helsingfors, Seinäjoki, Hyvinge, Jyväskylä, Åbo och Uleåborg. För tillfället sysselsätter företaget över 300 mjukvaruutvecklare.

Wapice verkar inom energibranschen och är en del av Energy Vaasa som är nordens ledande energikluster. Största delen av omsättningen kommer från konsulttjänster och de flesta av kunderna finns bland de 100 största företagen i Finland. Verksamheten är indelad i tre segment; Business solutions, Embedded systems, Industrial systems.

Business solutions arbetar med affärssystem för företag. Embedded systems är som namnet säger inriktat på inbyggda system medan Industrial systems är inriktat på produktionssystem, PLC och mobila lösningar. (Wapice, 2017)

1.2 Uppgift

Wapice har gjort en implementering av CANopen till ett konfigurerings- och monitoreringsprogram för att kunna kommunicera med produkter från andra tillverkare. Uppgiften var att skapa automatiska tester för denna CANopen-implementering. CANopen är ett högnivåkommunikationsprotokoll som är baserat på CAN (Controller Area Network) protokoll.

Tidigare har det skapats manuella tester för att verifiera CANopen-funktionaliteten. Dessa kan användas som grund för att skapa automatiska tester.

I teoridelen redogörs för grundläggande egenskaperna i CANopen och vad automatisk testning betyder. I därpå följande kapitel beskrivs testningsramverket och vilka teknologier som använts. Sedan kommer utförandet, där det redogörs vad som blev gjort. Till sist behandlas och diskuteras resultatet.

2 Teori

I detta kapitel beskrivs protokollet CANopen och vad automatiska tester betyder.

2.1 CANopen

Controller Area Network (CAN) protokollet presenterades 1986 av Robert Bosch i Detroit. Året därpå producerade Intel det första fristående CAN-kontroller chipet. Idag finns CAN-nätverk i de flesta bilar producerade i Europa och i många inbyggda system. I början av 1992 grundade användarna och tillverkarna av CAN, organisationen CAN in Automation (CiA) och dit hör också Wapice. Organisationen utvecklar och understöder CAN baserade protokoll och ger ut standarder. CAN-protokollet definierar inte applikationsskiktet så det fanns ett behov att specificera detta. 1995 publicerade CiA CANopen. CANopen definieras i flera olika dokument. CiA 301 definierar grundläggande CANopen-modulen och kommunikationen, medan mera specialiserade standarder bygger på den grundläggande. (Pfeiffer, 2003:xv)

Ett exempel på användningsområde för CANopen är ett specialfordon, som t.ex. ambulans. Där behöver varningsljus och sirener monteras på en paketbil. Kommunikationen i bilarnas komponenter görs vanligen med CAN-buss men tillverkarna har sina egna versioner som inte är kompatibla med andra tillverkares komponenter. För att möjliggöra en lätt integration kan CANopen användas för att kommunicera med komponenter från andra tillverkare.

Kommunikationssystem kan beskrivas med OSI-modellen (Figur 1), namnet kommer från Open Systems Interconnection. Den består av sju skikt. Varje skikt tillhandahåller en specifik tjänst som är oberoende av de tekniker som används i övriga skikt. Varje skikt tjänar det skikt som ligger ovanför och blir tjänad av det skikt som ligger nedanför. Alla skikt behöver inte implementeras. Längst ner i modellen finns det fysiska skiktet som beskriver den fysiska kontakten mellan noder i nätverket. I kommunikationssystem betyder noder anslutna enheter. Datalänkskiktet ansvarar för dataöverföring mellan noder. Nätverksskiktet sköter om adressering och ser till att ett meddelande går hela vägen från avsändare till mottagare. Transportskiktet ser till att meddelanden kommer fram utan ändringar från avsändare till mottagare. Sessionsskiktet sköter om att inleda och avsluta kommunikation mellan två noder. Presentationsskiktet ser till data representeras på ett standardiserat sätt. I presentationsskiktet kan också komprimering och kryptering ske.

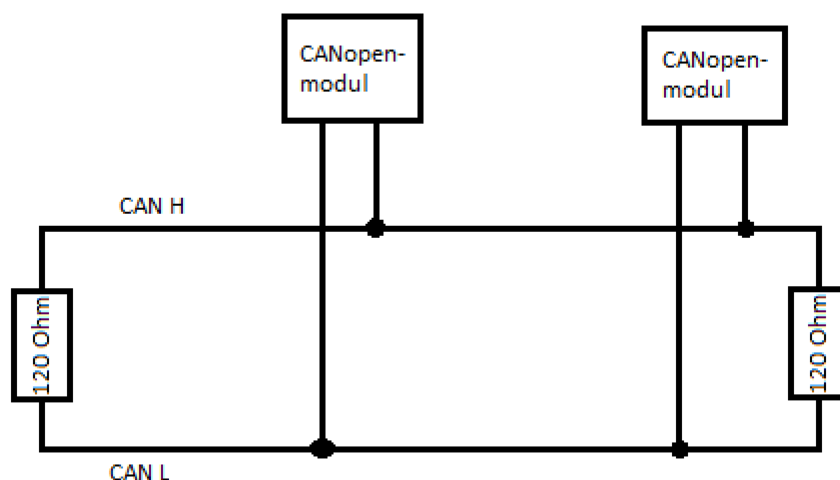
Applikationsskiktet är högst upp i modellen och sköter kommunikationen mellan program. (Pfeiffer, 2003:18–21)

<table>
<tr><td>Applikationsskiktet</td><td rowspan="5">CANopen-
applikationsskiktet</td></tr>
<tr><td>Presentationsskiktet</td></tr>
<tr><td>Sessionsskiktet</td></tr>
<tr><td>Transportskiktet</td></tr>
<tr><td>Nätverksskiktet</td></tr>
<tr><td colspan="2">Datalänkskiktet</td></tr>
<tr><td colspan="2">Det fysiska skiktet</td></tr>
</table>

Figur 1: OSI-modellen med sina 7 skikt på vänster sida. I CANopen-kommunikationsmodellen (till höger) implementeras inte alla skikt.

2.1.1 CAN fysiska skiktet

Den fysiska CAN-bussen består av två stycken ledningar som avslutas med 120 ohms motstånd i vardera ändan, se Figur 2. Vanligtvis används tvinnad par-kabel. Den ena ledningen kallas CAN H som betyder CAN hög och den har högre spänning än den andra ledningen CAN L. Det namnet kommer från CAN låg eftersom spänningen är lägre. En logisk etta representeras vanligtvis av en spänningsskillnad på två volt medan logisk nolla är noll volt, det vill säga ingen spännings skillnad. (Pfeiffer, 2003:206–207)

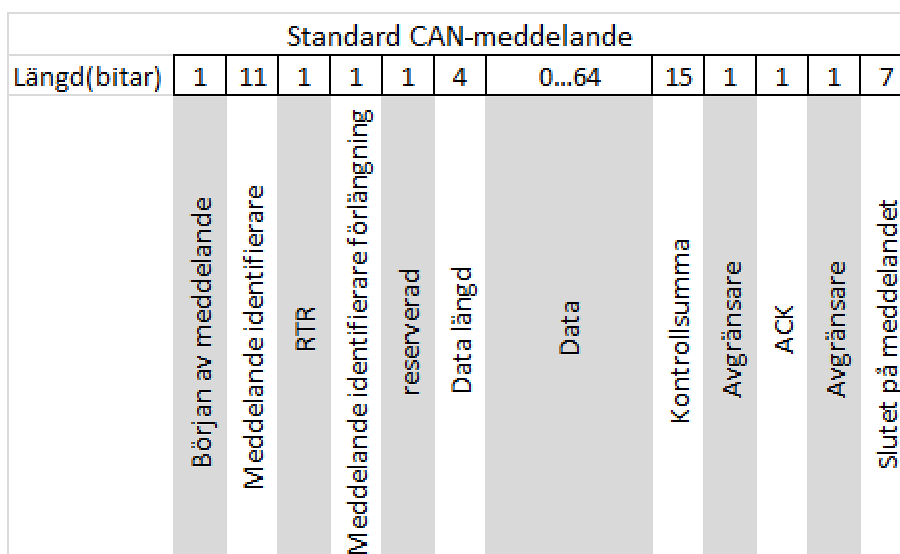


Figur 2: CAN-buss med ett par CANopen-moduler.

2.1.2 CAN datalänkskiktet

Standard CAN-meddelanden består av en elva bitars identifierare, en Remote Transmission Request (RTR) bit, data längden och upptill åtta byte av data (Figur 3). Dessutom finns det diverse andra bitar som används till exempel som kontrollsumma för att verifiera att

meddelandet är korrekt. RTR-biten används för att begära ett datameddelande från en nod. Det finns också förlängda CAN-meddelanden med 29 bitars identifierare, men de används inte i CANopen-standarden.



Figur 3: CANopen-meddelanden använder en elva bitars identifierare och kan innehålla upptill åtta byte av data.

CANopen-standarden delar in identifieraren i två delar, fyra bitar som beskriver funktionaliteten och sju bitar för CANopen-nodens identifierare (Figur 4). I CANopen kallas CAN-meddelandets identifierare för kommunikationsobjekt identifierare (COB-ID). Meddelande med lägre COB-ID har företräde ifall flera CANopen-noder vill sända samtidigt. (Pfeiffer, 2003:219–222)

	COB-ID	
	Funktionalitetskod	**Nod ID**
Längd	4 bitar	7 bitar

Figur 4: CANopen-meddelandets identifierare är indelat i funktionalitetskod samt nod ID.

2.1.3 CANopen-applikationsskiktet

Applikationsskiktet beskriver ett koncept för att konfigurera och kommunicera realtidsdata samt mekanismer för att synkronisera mellan CANopen-moduler. En applikation kommunicerar genom att kalla på tjänster av ett serviceobjekt i applikationsskiktet. För att förverkliga dessa tjänster utbyter objektet data via datalänksskiktet. (CiA 301, 2007:18)

2.1.4 CANopen-enhetsmodell

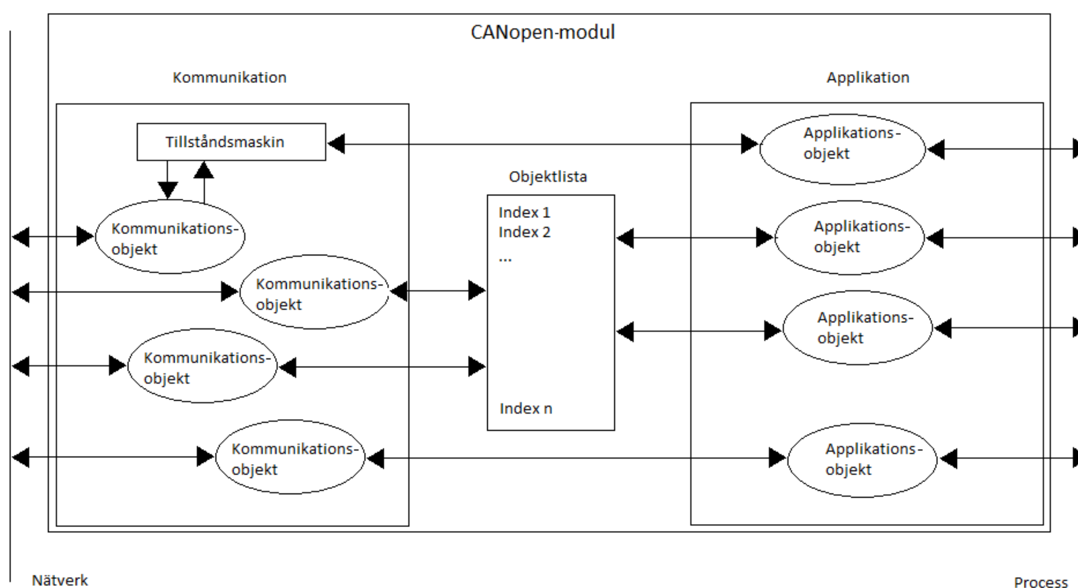
Varje CANopen-modul måste innehålla följande standardfunktioner (Figur 5):

- En kommunikationsenhet, som sköter om kommunikationen med andra noder i nätverket.

- En tillståndsmaskin, som kontrollerar startup och nollställning av enheten. Tillstånden som bör finnas är Initialization, Pre-operational, Operational samt Stopped. Initialization betyder att enheten är i initieringstillstånd. Pre-operational används främst för att konfigurera en enhet. Operational är tillståndet var enheten är operativ och utför sina normala funktioner. Stopped betyder att enheten inte är operativ.

- En objektlista, som är en samling av alla dataobjekt, som kan användas till att konfigurera modulen. Objektlistan beskriver hur modulen kommunicerar och tillhandahåller data.

- En applikationsdel, som sköter om modulens önskade funktionalitet när tillståndsmaskinen är i operativ status. (CiA 301, 2007:19–20)



Figur 5: CANopen-enhetsmodellen definierar en kommunikationsenhet som utbyter kommunikationsobjekt via underliggande nätverk. En applikationsdel som sköter om modulens funktionalitet och kommunicerar med processomgivningen. Objektlistan fungerar som ett gränssnitt mellan kommunikationen och applikationen.

2.1.5 Kommunikation

Det finns flera olika typer av CANopen-meddelanden och de indelas enligt funktionaliteten de tillhandahåller (Figur 6). Meddelandetypen definierar vilken typ av

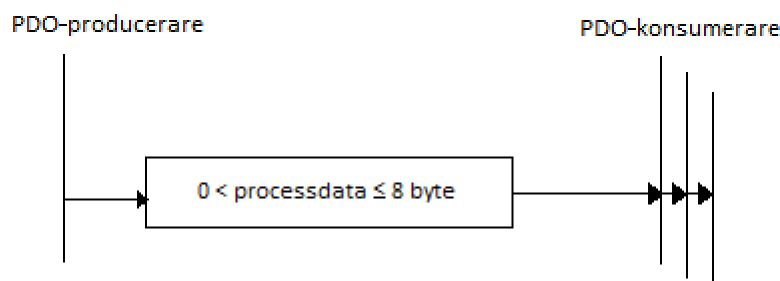
kommunikationsmodell som används. I en master/slave-modell är det alltid mastern som sänder eller begär data från en eller flera slavnoder, se Figur 13. Däremot i en client/server-modell är det endast en klient som kommunicerar med en servernod. Klienten sänder data som innehåller indexet i objektlistan på servern som skall läsas eller skrivas. Servern svarar med ett eller flera meddelanden med data innehållande indexets innehåll, se Figur 9. Den tredje modellen är en producer/consumer-modell där data skickas från en producent till en eller flera konsumenter. Sändningen kan vara begärd av konsumenten, kontinuerlig eller händelsestyrd, se Figur 7. (CiA 301, 2007:20–32)

Kommunikationsobjekt		Funktionalitets-kod		Resulterande COB-ID	
		dec	bin	dec	hex
broadcast-meddelanden	NMT-kommandon	0	0000	0	0
	SYNC-meddelande	1	0001	128	80
	Systemtid	2	0010	256	100
peer-to-peer-meddelanden	Tx-PDO1	3	0011	385-511	181-1FF
	Rx-PDO1	4	0100	513-639	201-27F
	Tx-PDO2	5	0101	641-767	281-2FF
	Rx-PDO2	6	0110	769-895	301-37F
	Tx-PDO3	7	0111	897-1023	381-3FF
	Rx-PDO3	8	1000	1025-1151	401-47F
	Tx-PDO4	9	1001	1153-1279	481-4FF
	Rx-PDO4	10	1010	1281-1407	501-57F
	Tx-SDO	11	1011	1409-1535	581-5FF
	Rx-SDO	12	1100	1537-1663	601-67F
	Heartbeat	14	1110	1793-1919	701-77F

Figur 6: Exempel på kommunikationsobjekt och hur COB-ID kan variera beroende på funktionalitetskod samt nodens identifierare.

2.1.6 Process data object

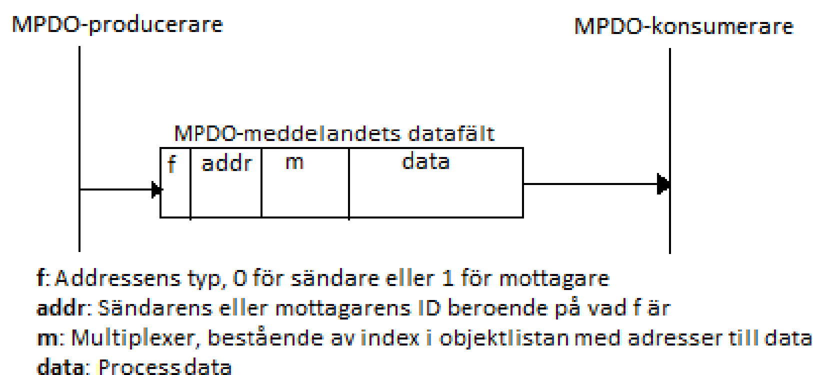
Process data object (PDO) är till för att överföra realtids data. PDO delas in i två typer, Transmit-PDO (TPDO) för att skicka och Receive-PDO (RPDO) för att ta emot data. PDO sändning kan vara antingen kontinuerlig eller händelsestyrd. Det finns också möjlighet för PDO-konsumenterna att begära data med PDO read där RTR-flaggan används. Detta är dock en frivillig extra funktionalitet. Hela meddelandets datafält kan användas till data, så att upp till 8 byte kan överföras med ett meddelande (Figur 7). (CiA 301, 2007:33–36)



Figur 7: Processdata överförs med ett PDO-meddelande till en eller flera konsumenter.

2.1.7 Multiplex process data object

Multiplex process data object (MPDO) ger direkt tillgång till en CANopen-nods objektlista. Detta minskar möjliga datamängden till högst fyra byte. Resten går till att definiera CANopen-noden och index i objektlistan (Figur 8). Fördelen med MPDO över PDO är flexibilitet. Samma meddelandeidentifierare kan användas för flera olika data genom att variera multiplexern, bestående av index i objektlistan med adresser till data. Detta kan till exempel vara användbart i ett system med 20 temperatursensorer vars data behöver överföras när deras värden ändrar. Istället för att reservera flera PDO-medelanden för överföringen kan ett MPDO användas och skickas 20 gånger. (CiA 301, 2007:36–38)

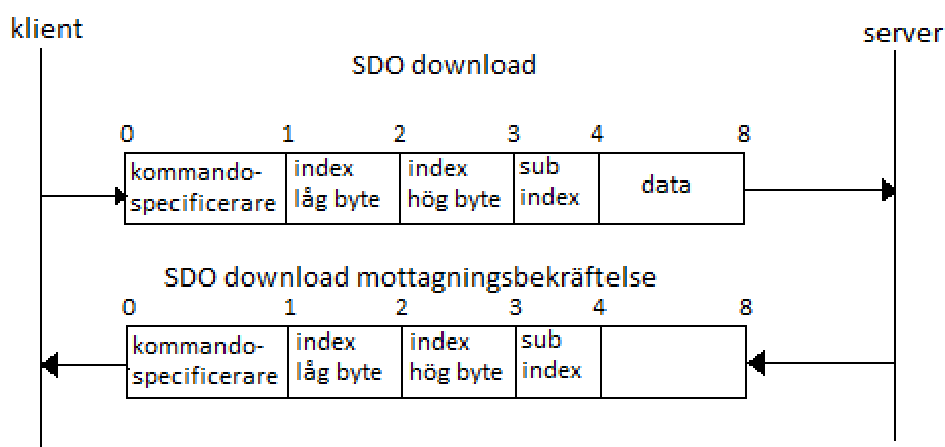


Figur 8: Hälften av MPDO-meddelandets datafält används till att definiera noden samt index för data.

2.1.8 Service data object

Service data object (SDO) används för att sätta och läsa värden från en nods objektlista och används främst till att konfigurera enheten. Vid kommunikation med SDO-meddelanden skickas alltid en mottagningsbekräftelse (Figur 9). Olika typer av SDO-meddelanden är SDO download, SDO upload och SDO abort transfer. SDO download används för att överföra data från klienten till serverns objektlista. SDO upload används för att överföra

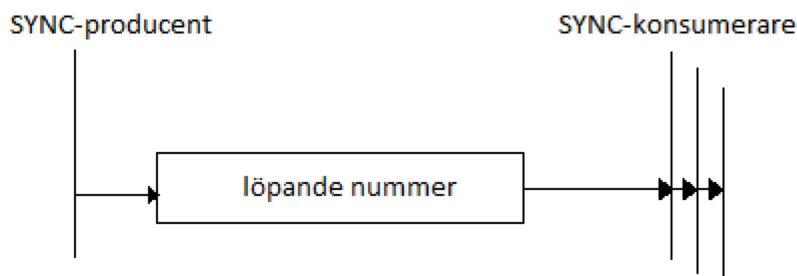
data från serverns objektlista till klienten. SDO abort transfer används för att avbryta en pågående SDO-operation och kan utföras av både klienten och servern. Meddelandena består av en kommandospecificerare som anger vilken typ av meddelande det är, objektlistans index samt delindex och data. Om data är högst fyra byte kan det överföras i ett meddelande. Vid större datamängder sker överföringen i flera meddelanden. (CiA 301, 2007:39–66)



Figur 9: SDO download meddelandets datafält används till en kommandospecificerare, objektlistans index samt delindex och data. Meddelandet besvaras med en mottagningsbekräftelse.

2.1.9 Synchronization object

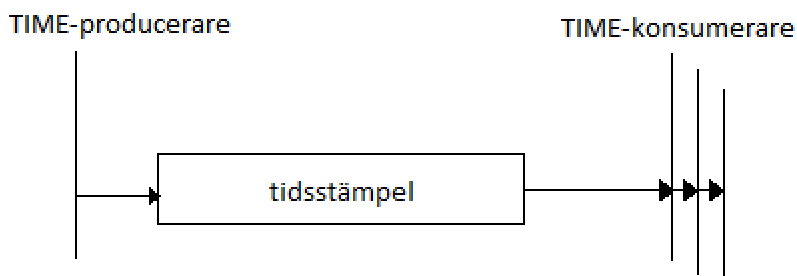
Synchronization object (SYNC) tillhandahåller grundläggande nätverkssynkroniseringsmekanism. SYNC producenten utsänder kontinuerligt ett synkroniseringsmeddelande som möjliggör rätt samordning i tiden för överföring av data (Figur 10). När SYNC konsumenterna mottar detta utför de sina uppgifter. Det kan vara att skicka PDO. (CiA 301, 2007:67–68)



Figur 10: SYNC-meddelandet innehåller en löpande nummer som börjar från ett.

2.1.10 Time stamp object

Time stamp object (TIME) tillhandahåller en enkel nätverksklocka. Datafältet innehåller 6 byte med tiden som passerat sedan första januari 1984 i millisekunder (Figur 11). (CiA 301, 2007:68–69)



Figur 11: Tiden kan distribueras till flera konsumenter med ett TIME-meddelande.

2.1.11 Emergency object

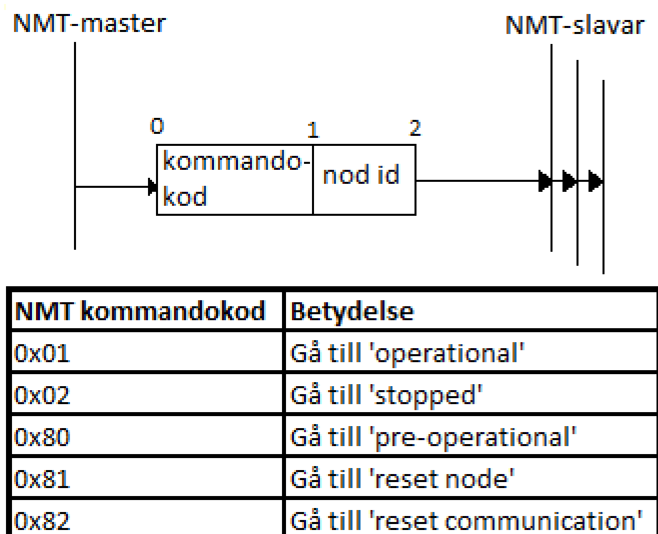
Emergency object (EMCY) utlöses ifall det sker ett internt fel i en CANopen-modul (Figur 12). De skall endast skickas en gång per händelse så inga fler medelanden skall skickas om inga nya fel uppstår. (CiA 301, 2007:69–72)

Figur 12: Innehållet i ett EMCY-meddelande.

2.1.12 Network management

Network management (NMT) används för att initiera, starta, övervaka, nolla eller stoppa CANopen-moduler (Figur 13). Alla CANopen-moduler ses som NMT-slavar. En NMT-slav är identifierad med sin nodidentifierare, som är ett värde mellan 1 och 127. NMT kräver att en CANopen-modul i nätverket fungerar som NMT-master. Identifieraren för CAN-meddelandet är alltid noll vilket betyder att funktionalitetskoden är noll samt identifieraren är noll. Detta medför att alla noder processerar detta meddelande. Den verkliga nodidentifieraren är en del av data i meddelandet. Ifall nodidentifieraren är noll, är

meddelandet avsett till alla noder. Då skall alla noder på bussen gå till det angivna läget. (CiA 301, 2007:72–79)

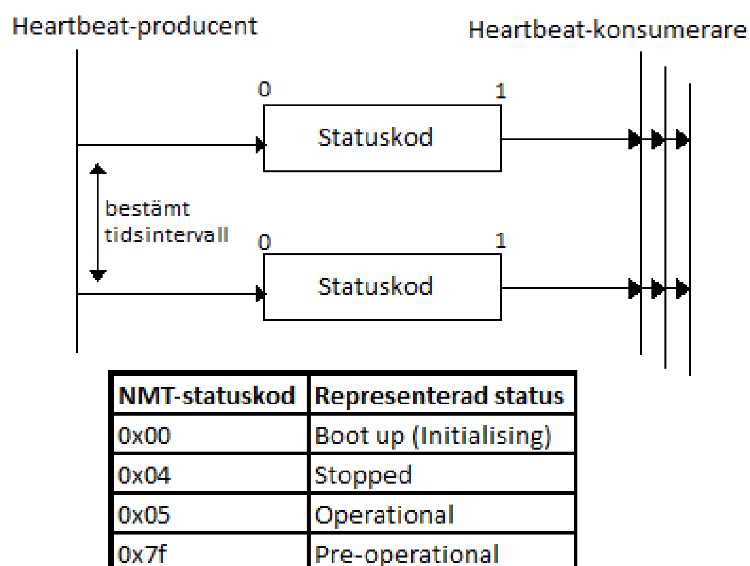


NMT kommandokod	Betydelse
0x01	Gå till 'operational'
0x02	Gå till 'stopped'
0x80	Gå till 'pre-operational'
0x81	Gå till 'reset node'
0x82	Gå till 'reset communication'

Figur 13: Innehållet i ett NMT-meddelande samt kommandokodernas betydelse.

2.1.13 Heartbeat

Heartbeat-meddelanden skickas kontinuerligt från CANopen-noden med bestämda tidsintervall. Meddelandena innehåller nodens status (Figur 14). Detta används för att övervaka noder och verifiera att de är i liv. Om meddelandet inte mottas inom utsatt tid kan NMT-mastern starta om noden eller indikera att ett fel har uppstått. (CiA 301, 2007:80)



NMT-statuskod	Representerad status
0x00	Boot up (Initialising)
0x04	Stopped
0x05	Operational
0x7f	Pre-operational

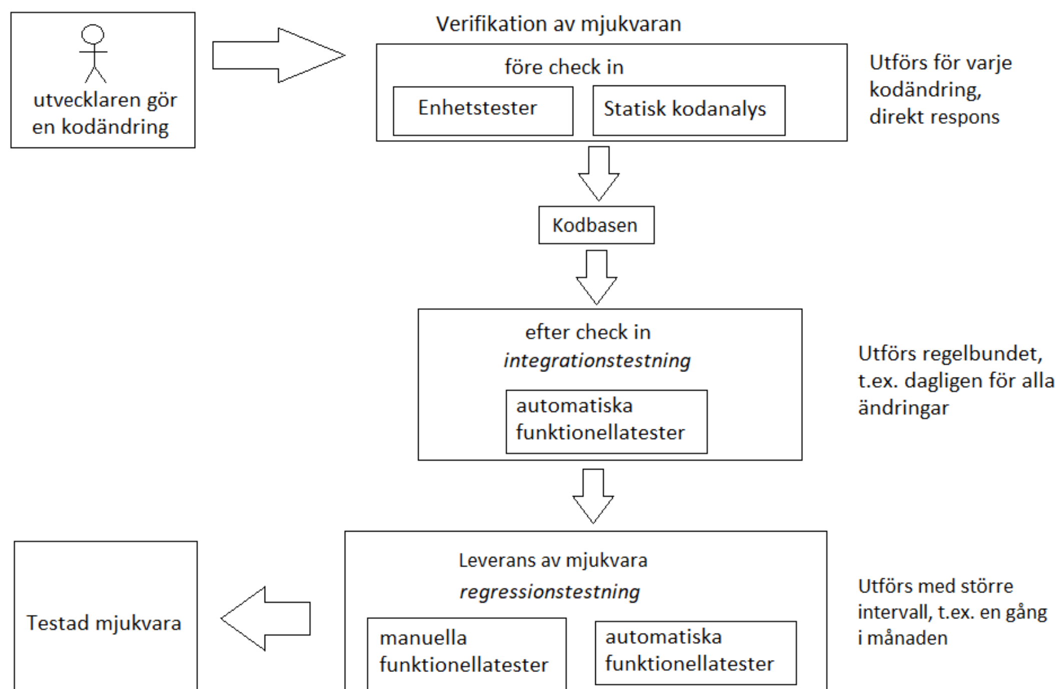
Figur 14: Heartbeat-meddelanden innehåller nodens status och skickas kontinuerligt. De används för att övervaka noden.

2.2 Automatiska tester

I detta kapitel presenteras vad automatiska tester är samt fördelar med automatisk testning över manuell testning och när det inte rekommenderas att använda dem.

2.2.1 Typer av tester

Med automatiska tester avses här funktionella tester som används för integrations- och regressionstestning. Testning av mjukvara kan delas in i flera delar under mjukvarans utvecklingscykel (Figur 15). Först utför utvecklaren tester lokalt medan han gör implementeringen. Innan ändringarna införs i källkoden kan t.ex. enhetstester samt statisk kodanalys utföras. Enhetstester fokuserar på enskilda funktioner medan funktionella tester ser på en större helhet av programmet. Statisk kodanalys körs utan att exekvera programmet och ser till att koden följer angivna regler. Sedan kommer integrationstestning som sker då ändringarna införts i källkoden. Integrationstestning utförs för att verifiera att nya ändringarna fungerar som förväntat tillsammans med existerande kod. Vid mjukvaruleveranser utförs regressionstestning för att verifiera att nya ändringar inte haft sönder gammal funktionalitet. Förutom funktionella tester kan också icke-funktionella tester utföras, dessa omfattar till exempel prestanda tester. (Sharma, 2017:39–40)



Figur 15: Före en kodändring läggs till i kodbasen kan ändringen verifieras med enhetstester och statisk kod analys. De tar kort tid att utföra och ger snabb respons till utvecklaren. Efter att ändringen är i kodbasen utförs integrationstester. De tar mera tid att utföras, så responstiden är längre. Vid leverans utförs regressionstestning för att verifiera mjukvaran.

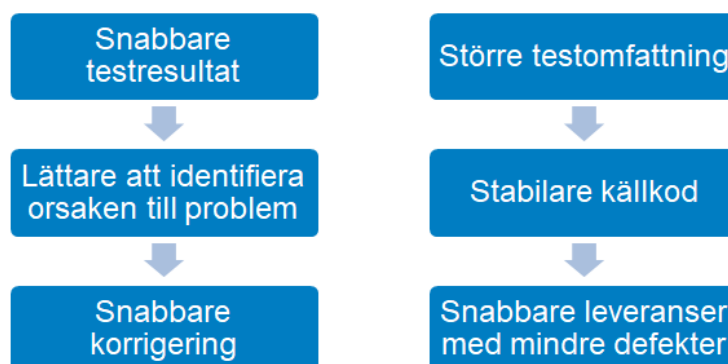
Automatiska funktionella tester ersätter inte behovet av enhetstester utan de komplementerar enhetstester.

2.2.2 Fördelar med automatiska tester

Med automatiska funktionella tester kan man spara resurser som tid och pengar. Detta eftersom en dator kan utföra operationer mycket snabbare än vad en människa kan. Då tester kan utföras snabbare kan de upprepas oftare och samtidigt kan flera kombinationer testas (Figur 16). Detta ger testerna en bättre täckning av programkoden. Fel som hittas i ett tidigare skede är billigare att korrigera, detta eftersom färre personer involveras.

Andra fördelar är att en dator inte tar genvägar, utan om testet specificerar att en sak skall utföras tio gånger, så utförs varje upprepning. Dessutom gör en dator inte mänskliga misstag, utan den gör exakt som angivet. Datorn är också striktare med tolkningen av resultatet. En människa kan se mellan fingrarna och godkänna ett resultat som inte är helt rätt, medan datorn ger felmeddelande.

I takt med att agilt utvecklande blir mera vanligt, blir också automatisering viktigare. Kontinuerlig integration kräver automatisering av tester. Regressionstester utförs dagligen, med fördel oftare. Det är en förutsättning för kontinuerliga mjukvaruleveranser. Testautomation kan vara framgångsrikt i både traditionellt och i agilt utvecklande, men agilt utvecklande kan inte lyckas utan testautomation. (Graham, 2011:4)



Figur 16: Fördelar som snabbare testresultat med automatiska tester kan medföra snabbare korrigering av fel. Större testomfattning kan i slutändan medföra mindre antal defekter i mjukvaran och snabbare leverans.

2.2.3 Utmaningar med automatiska tester

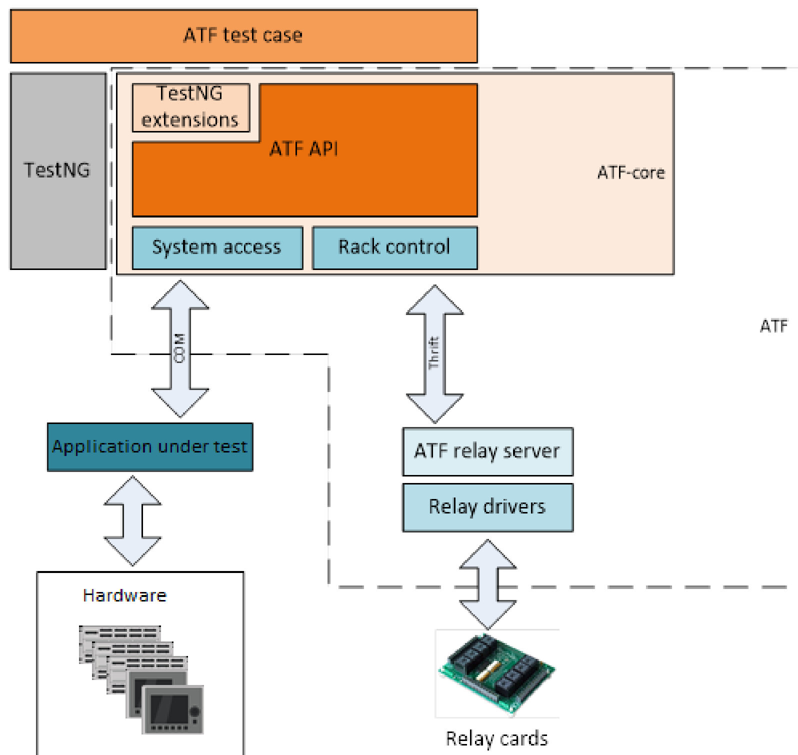
Automatiska tester betyder inte att de fungerar utan manuellt arbete. De utför uppgifter automatiskt medan resultatet utvärderas manuellt. För att hitta möjliga problem behövs det dessutom manuell felsökning. Detta betyder att bra automatiska tester skall ha vettiga felmeddelanden för att underlätta eventuell felsökning. Testet bör också vara lättläst och det bör framgå tydligt vad det testar samt hur. Detta underlättar underhåll av testerna när ändringar i mjukvaran görs.

Andra egenskaper som påverkar hur bra ett automatisk test är, är stabilitet och tiden som testet tar att utföra. Ett test som inte ger trovärdigt resultat är inte tillförlitligt. Därför bör slumpmässiga variabler elimineras. Fastän en dator kan utföra tester snabbare än en människa, bör tiden det tar för att utföra testerna beaktas och testplanerna optimeras. Testerna bör vara relevanta och verifiera krav. Om varje tänkbart testfall skulle automatiseras och användas i regressionstestplanen, skulle det kräva för mycket resurser och tid att utföra. Det i sin tur skulle leda till för lång responstid samt höga underhållskostnaderna. För att undvika detta behövs regelbundna genomgångar av de automatiska testerna, för att se till att de fyller sin uppgift. (Graham, 2011:25–26)

3 Teknik

För att utföra automatiska tester använder Wapice ett testramverk som kallas för Automatic Test Framework (ATF). Ramverket är byggt på TestNG. TestNG är ett verktyg för att skapa enhetstester och funktionella tester i Java. Några av huvudfunktionerna TestNG erbjuder är stöd av annotationer, stöd för parametriserad och data-driven testning. Det är flexibelt att exekvera testerna, antingen direkt från utvecklingsmiljön eller konsolen med Apache Ant. (TestNG, 2018)

ATF består av ett Java bibliotek, atf-core, som är direkt använt i testerna genom det offentliga applikationsprogrammeringsgränssnittet (Figur 17). Det erbjuder funktioner för att modifiera status av hårdvaran och styra mjukvaran. Gömt från det offentliga gränssnittet finns två delsystem som tillhör de viktigaste delarna i atf-core: system access och rack control. System access delsystemet gömmer detaljerna hur kommunikationen sker med applikationen som testas från resten av systemet, genom att inkapsla all relaterad kod. Delsystemet rack control sköter styrningen av hårdvaran. Det kommunicerar med ATF relay server, som är en separat process ansvarig för att styra reläerna som kontrollerar hårdvaran.



Figur 17: Testerna finns i separata klasser som har tillgång till testramverkets funktioner via det offentliga applikationsprogrammeringsgränssnittet ATF API. System access och rack control delsystemen används för att styra och kommunicera med hårdvaran.

Som programmeringsspråk för testerna används Java och utvecklingsmiljön som används för testerna är IntelliJ IDEA. IntelliJ IDEA erbjuder ett utbrett stöd för utveckling i Java. (Jetbrains, 2018)

Som versionshanteringsprogram används Git. Där sparas både testerna och testramverket. Största fördelen med Git över andra versionshanteringsprogram är hur lätt det är att skapa och arbeta med grenar. Git är dessutom mycket snabbt jämfört med andra versionshanteringsprogram. (Git, 2018)

För att undersöka och övervaka trafiken på CAN-bussen användes CANrunner, ett program utvecklat av Wapice. Det är gratis och finns att laddas ner från företagets hemsida: https://www.wapice.com/products/canrunner

För att koppla datorn till CAN-bussen användes CAN-till-USB-adapter från tillverkaren Kvaser. (Kvaser, 2018)

4 Utförande

I detta kapittel beskrivs hur uppgiften blev utförd.

4.1 Definition av tester

Wapice har utvecklat mjukvara för moduler som skall fungera som master i ett CANopen-nätverk. Tidigare har mjukvaran testats manuellt med funktionella tester. Det har varit mycket tidskrävande och därför behöver de grundläggande testerna automatiseras.

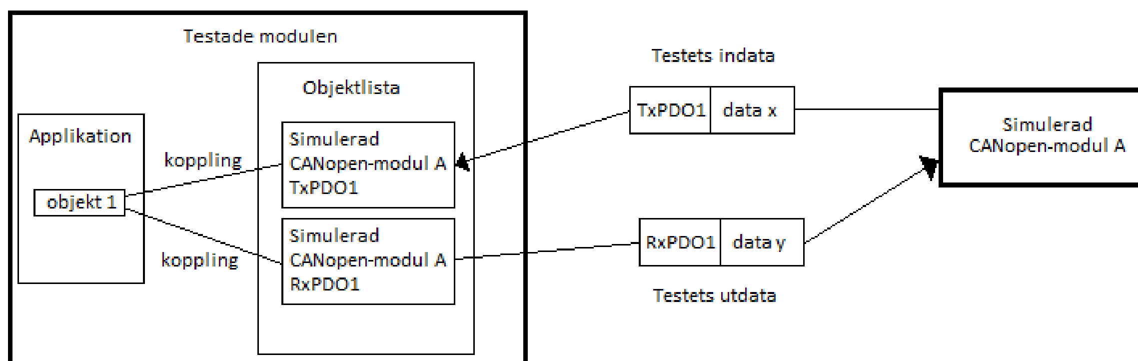
De existerande testerna kan användas som grund. För att täcka grundfunktionaliteten skall SDO, PDO, EMCY och NMT testas. Problemfall som överbelastning, kabelbrott och kortslutning av CAN-bussen bör också omfattas av testerna. Testerna omfattar inte användargränssnitt eller konfigurering.

4.2 Skapandet av testerna

Först presenteras den manuella versionen av ett test i detalj. Sedan visas hur det automatiska testet skapas steg för steg.

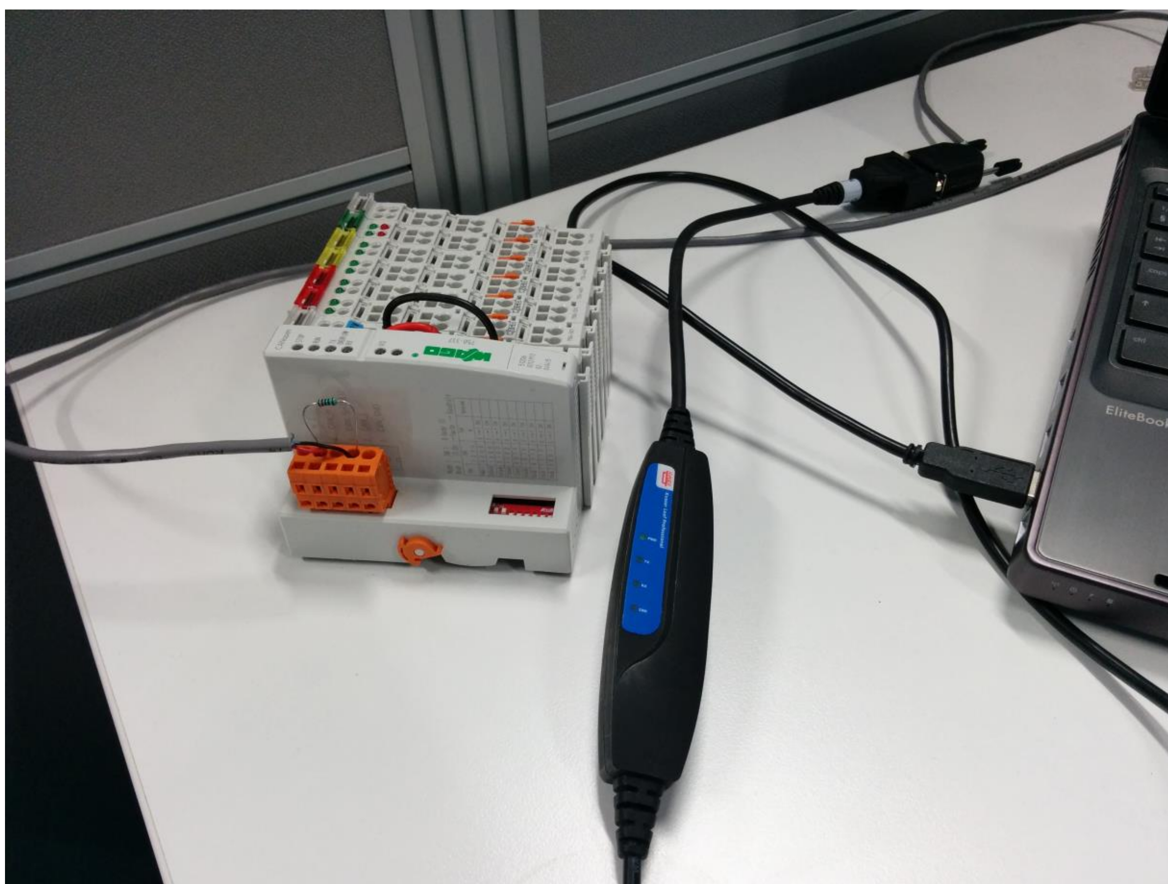
4.2.1 Manuellt test av PDO

Manuella versionen av testet som verifierar att skicka och ta emot process data ser ut på följande sätt. Först presenteras kravet på hårdvara för att utföra testet. I detta fall krävs modulen som testas samt en Kvaser CAN-till-USB-adapter för att ansluta datorn till modulens CAN-buss. För att analysera kommunikationen på CAN-bussen behövs CANrunner eller motsvarande mjukvara. Andra förutsättningar för testet är att den testade modulens konfiguration har en annan CANopen-moduls TxPDO och RxPDO kopplad till samma objekt (Figur 18). Detta medför att samma data som kommer in kommer att skickas tillbaka. Som period för att skicka PDO används 1000ms.



Figur 18: Testet simulerar en CANopen-modul A som skickar data x, eftersom både TxPDO och RxPDO är kopplade till samma objekt kommer data y vara lika med data x om testet fungerar.

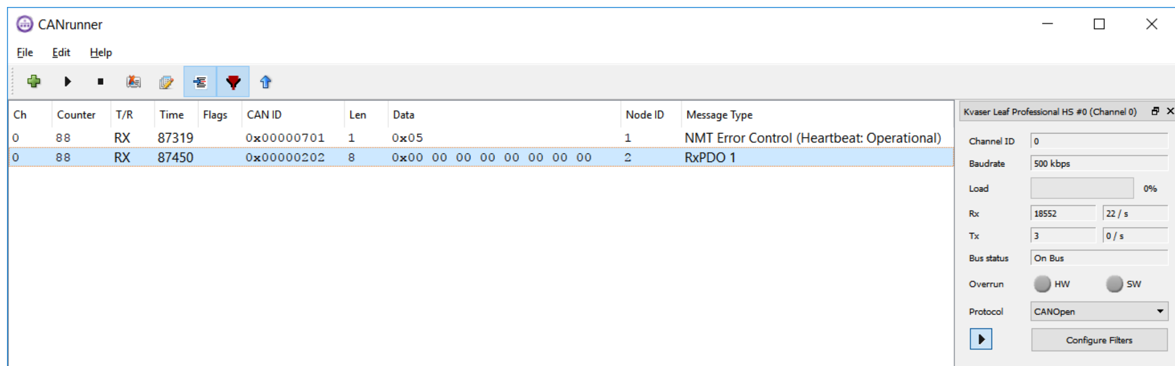
Första steget i utförandet av testet är att ladda ner mjukvaran till den testade modulen. Det förväntade resultatet är att nerladdningen går felfritt och modulen startar upp utan problem. Eftersom den konfigurerade CANopen-modulen saknas är det tillåtet att fel om saknad modul rapporteras.



Figur 19: Bild av datorn kopplad till en CANopen-modul med en CAN-till-USB-adapter.

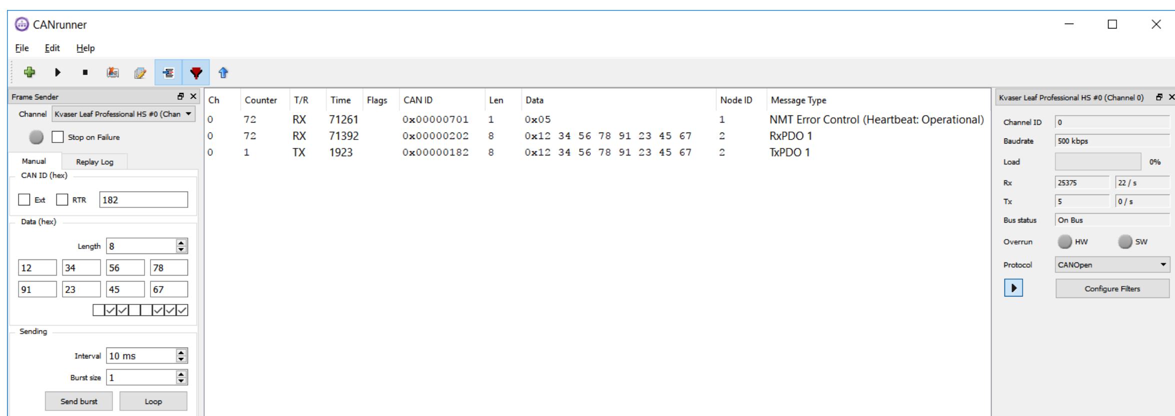
Andra steget är att starta CANrunner och initiera Kvasern på CAN-bussen. Sedan sätta till ett filter för att blockera förlängda CAN-meddelanden. Förväntade resultatet är att se CAN-medelanden med COB-ID 0x200 + konfigurerade CANopen-modulens identifierare.

Om modulens identifierare är 2 så är COB-ID 0x202 för meddelandet. Meddelandena bör komma kontinuerligt med 1000ms period och alla åtta byte av data skall vara noll.



Figur 20: Skärmdump som visar det förväntade resultatet efter andra steget i testet.

Sista steget i testet är att generera ett inkommande PDO till den testade modulen. Det kan göras med hjälp av CANrunner genom att skicka ett meddelande med COB-ID 0x180 + den konfigurerade CANopen-modulens identifierare. Som meddelandets data sätts 0x12 34 56 78 91 23 45 67. Eftersom ingående och utgående PDO är kopplade till samma objekt i den testade modulen, är det förväntade resultatet att utgående data uppdateras till samma som ingående. Meddelandet med COB-ID 0x200 + konfigurerade CANopen-modulens identifierare skall innehålla åtta byte av data och vara 0x12 34 56 78 91 23 45 67. Detta verifierar att modulen under test kan ta emot och skicka PDO-data samt att kopplingen fungerar korrekt.



Figur 21: Skärmdump som visar det förväntade resultatet efter sista steget i testet.

4.2.2 Automatiska testet av PDO

För att möjliggöra testning av flera olika system med samma tester skapades testerna i en basklass som de systemspecifika klasserna sedan ärver. De systemspecifika klasserna definierar vilken hårdvara som krävs. Hjälpfunktioner placeras i en hjälpklass så att

basklassen kan innehålla enbart själva testlogiken (Figur 22). Detta förbättrar läsbarheten och underhållet. Det ursprungliga manuella testet av PDO delades in i två separata tester. Ett för sändning och ett för mottagning av PDO för att göra testerna så tydliga som möjligt. Först måste testet verifiera att CANopen-modulen är i rätt läge. För att en CANopen-modul skall utbyta processdata måste den vara i operativt läge.

```
CANopenUtils.waitForHeartbeat(releaseTest, masterNodeId, CANopenUtils.CANopenState.CANopen_OPER, timeout: 1,
    TimeUnit.MINUTES, message: nmtMaster.getRackName() + " did not go to operational within timeout");
```

Figur 22: Hjälpklassen tillhandahåller funktionalitet för att verifiera att modulen är i rätt läge.

I testet som verifierar mottagningen av PDO hos den testade modulen, så skickas ett TxPDO-meddelande med åtta byte av data, där varje byte är kopplad till skilda objekt som vi kan avläsa och verifiera.

```
//Send TxPDO, generate a CAN frame with CAN ID = 0x182 and send 8 bytes of data
CANopenUtils.sendCANOMessage(releaseTest, canId: CANopenUtils.CANopenTxPDO1ID + deviceNodeId,
    new CANData( ...bytes: 0x12, 0x34, 0x56, 0x78, 0x91, 0x23, 0x45, 0x67));
```

Figur 23: Hjälp klassen tillhandahåller funktionalitet för att skicka ett CANopen-medelande.

Till sist jämförs det avlästa värdet med det förväntade (Figur 24). Om de inte är lika ges ett felmeddelande.

```
//Check that values were updated to mapped codes
assertThat(nmtMaster.getDCCode(CANO_PDO1_1).readValue())
    .as("Value for code mapped to PDO1 index 1 is not correct").isEqualTo(0x12);
```

Figur 24: Verifikation att det lästa värdet är samma som det förväntade värdet.

Hela testet som verifierar mottagningen av PDO ser ut som i Figur 25.

```
@Test
public void testTxPDO() {
    CANopenUtils.waitForHeartbeat(releaseTest, masterNodeId, CANopenUtils.CANopenState.CANopen_OPER, timeout: 1,
        TimeUnit.MINUTES, message: nmtMaster.getRackName() + " did not go to operational within timeout");

    ATFUtils.startStep( step: "Generate a PDO message and check that mapped codes are updated");

    //Send TxPDO, generate a CAN frame with CAN ID = 0x182 and send 8 bytes of data
    CANopenUtils.sendCANOMessage(releaseTest, canId: CANopenUtils.CANopenTxPDO1ID + deviceNodeId,
        new CANData( ...bytes: 0x12, 0x34, 0x56, 0x78, 0x91, 0x23, 0x45, 0x67));

    //Wait a moment
    ATFUtils.sleep( time: 1, TimeUnit.SECONDS);

    //Check that values were updated to mapped codes
    assertThat(nmtMaster.getDCCode(CANO_PDO1_1).readValue())
        .as("Value for code mapped to PDO1 index 1 is not correct").isEqualTo(0x12);
    assertThat(nmtMaster.getDCCode(CANO_PDO1_2).readValue())
        .as("Value for code mapped to PDO1 index 2 is not correct").isEqualTo(0x34);
    assertThat(nmtMaster.getDCCode(CANO_PDO1_3).readValue())
        .as("Value for code mapped to PDO1 index 3 is not correct").isEqualTo(0x56);
    assertThat(nmtMaster.getDCCode(CANO_PDO1_4).readValue())
        .as("Value for code mapped to PDO1 index 4 is not correct").isEqualTo(0x78);
    assertThat(nmtMaster.getDCCode(CANO_PDO1_5).readValue())
        .as("Value for code mapped to PDO1 index 5 is not correct").isEqualTo(0x91);
    assertThat(nmtMaster.getDCCode(CANO_PDO1_6).readValue())
        .as("Value for code mapped to PDO1 index 6 is not correct").isEqualTo(0x23);
    assertThat(nmtMaster.getDCCode(CANO_PDO1_7).readValue())
        .as("Value for code mapped to PDO1 index 7 is not correct").isEqualTo(0x45);
    assertThat(nmtMaster.getDCCode(CANO_PDO1_8).readValue())
        .as("Value for code mapped to PDO1 index 8 is not correct").isEqualTo(0x67);
}
```

Figur 25: Skärmdump av programkoden för TxPDO-test.

På motsvarande sätt skapas det automatiska testet för skicka ett PDO. Objekten i den testade modulen är kopplade till ett RxPDO som skickas kontinuerligt. Först skrivs data till objekten och sedan kan PDO-meddelandet avläsas från CAN-bussen och verifiera att det är korrekt. Se Figur 26.

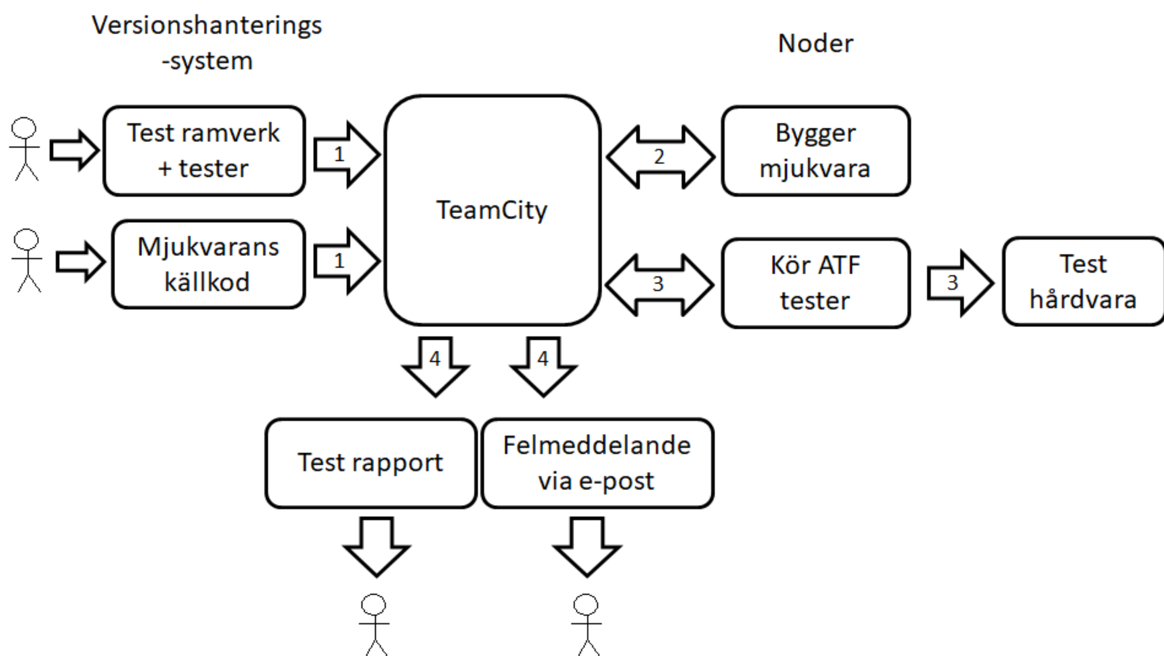
```
@Test
public void testRxPDO() {
    CANopenUtils.waitForHeartbeat(releaseTest, masterNodeId, CANopenUtils.CANopenState.CANopen_OPER, timeout: 1,
        TimeUnit.MINUTES, message: nmtMaster.getRackName() + " did not go to operational within timeout");

    ATFUtils.startStep( step: "Update mapped codes and verify that periodically sent PDO is updated accordingly");

    //update codes
    nmtMaster.getDCCode(CANO_PDO1_1).writeValue( value: 0x01);
    nmtMaster.getDCCode(CANO_PDO1_2).writeValue( value: 0x02);
    nmtMaster.getDCCode(CANO_PDO1_3).writeValue( value: 0x03);
    nmtMaster.getDCCode(CANO_PDO1_4).writeValue( value: 0x04);
    nmtMaster.getDCCode(CANO_PDO1_5).writeValue( value: 0x05);
    nmtMaster.getDCCode(CANO_PDO1_6).writeValue( value: 0x06);
    nmtMaster.getDCCode(CANO_PDO1_7).writeValue( value: 0x07);
    nmtMaster.getDCCode(CANO_PDO1_8).writeValue( value: 0x08);

    //Wait for the periodic RxPDO with CAN ID = 0x202
    CANopenUtils.waitForCANOMessage(releaseTest, deviceNodeId, CANopenUtils.CANopenRxPDO1ID, dataLength: 8,
        new CANData( ...bytes: 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08), dataMask: null, timeout: 10,
        TimeUnit.SECONDS, message: "Did not receive expected RxPDO message");
}
```

Figur 26: Skärmdump av programkoden för RxPDO-test.

5 Resultat

Automatiska tester skapades för grundfunktionaliteten i CANopen som flera olika typer av PDO, initierings SDO, EMCY och NMT-meddelanden. Tester för problemfall skapades inte ännu i detta skede, eftersom testramverket saknade stöd för att styra CANopen-moduler. De skapade testerna kompletterar de existerande enhetstesterna och minskar behovet av manuell testning av CANopen-funktionaliteten.

De automatiska testerna körs varje natt och verifierar att inga ändringar som kommit in har haft sönder existerande funktionalitet. Det kontinuerliga integrationssystemet som används hämtar senaste källkoden från versionshanteringssystemet, bygger mjukvaran och startar testerna via ATF (Figur 27). Mjukvaran laddas ner på hårdvaran som styrs av reläer. Resultatet av testerna returneras till integrationssystemet, där det sparas. Resultatet skickas också vidare till applikationer som presenterar resultatet i lättförståeliga former såsom grafer. Om det uppstår problem, till exempel att ett test eller mjukvarubyggandet misslyckas så informeras de berörda personerna automatiskt via e-postmeddelande.



Figur 27: Som kontinuerligt integrationssystem användes TeamCity. Först hämtas källkoden från versionshanteringssystemet. Sedan byggs mjukvaran och automatiska testerna utförs. Resultatet görs tillgängligt för användarna genom en test rapport och ifall testerna misslyckas informeras utvecklaren via e-postmeddelande.

När testerna går igenom felfritt innehåller testets loggfil endast informativa uppdateringar. Se Figur 28.

```
ATFLogger - === Starting test canopen.CANopen_Manager_DUT:testRxPDO ===
ATFLogger - Step: Update mapped codes and verify that periodically sent PDO is updated accordingly
ATFLogger - === Test canopen.CANopen_Manager_DUT:testRxPDO passed. ===
```

Figur 28: Exempel på hur resultatet rapporteras när RxPDO-testet går igenom felfritt

När ett test misslyckas, innehåller testloggen ett meddelande som berättar vad som gått fel. Loggen innehåller också en stack trace som visar hur testet exekverades. Dessa hjälper till att förstå vad som gått fel, men ibland behövs mera information än vad som ses i exemplet i Figur 29. Där kan man bara se att det förväntade meddelandet inte hittades. I detta fall behöver CAN-bussens logg undersökas. Den sparas av testramverket men det behövs manuellt arbete att gå igenom den.

```
ATFLogger - === Starting test canopen.CANopen_Manager_DUT:testRxPDO ===
ATFLogger - Step: Update mapped codes and verify that periodically sent PDO is updated accordingly

java.lang.AssertionError: [Did not receive expected RxPDO message]
Expecting:
 <false>
to be equal to:
 <true>
but was not.

    at canopen.CANopenUtils.waitForCANOMessage(CANopenUtils.java:148)
    at canopen.CANopenManager_Base.testRxPDO(CANopenManager_Base.java:353)
    ...

ATFLogger - === Test canopen.CANopen_Manager_DUT:testRxPDO FAILED
```

Figur 29: Exempel på hur resultatet rapporteras när RxPDO-testet misslyckas

6 Diskussion

Resultatet av utvecklingsarbetet är en grunduppsättning av 15 stycken automatiska CANopen-tester för att verifiera kommunikationen. Exekveringstiden för testerna är endast cirka 15 minuter. Detta sparar snabbt resurser eftersom de manuella testerna var tidskrävande. Det kunde ta mer än en dag att utföra testerna manuellt eftersom de krävde speciell hårdvarukonfiguration. Därför utfördes de manuella testerna endast vid mjukvaruleveranser. Nu kan de automatiska testerna utföras varje dag. Om det görs ändringar i CANopen eller andra komponenter som påverkar funktionaliteten så verifieras de kontinuerligt och möjliga fel upptäcks i ett tidigt skede av utvecklingen och kan åtgärdas fort.

Ett problem som kom fram i ett tidigt skede, var att testramverket saknade stöd för CANopen-moduler. Detta problem kan för det mesta kringgås genom att inte använda en

CANopen-modul, utan istället simulera kommunikationen genom att skicka förväntade CAN-medelanden från test ramverket till den testade modulen.

Andra problem var att ingen använt ATF för att skicka CANopen-meddelanden tidigare, vilket innebar att det fanns flera fel i ramverket som behövde korrigeras. Också i den testade mjukvaran upptäcktes flera fel och brister som rapporterades. Fel som förhindrar nödvändig konfigurering för att utföra testet, är problematiska. Sådana fel gör det svårt att veta om testet kommer att fungera rätt, innan felet som förhindrar konfigurering är tillrättat. Dessutom kan finnas flera fel, som inte blir upptäckta förrän konfigureringen är möjlig.

När de automatiska CANopen-testerna är gjorda samt är i körning, är det viktigt att följa med resultatet. Misslyckas testerna behöver test resultatet analyseras så fort som möjligt. Precis liksom med andra automatiska tester behöver de underhåll för att inte förfalla. Det behöver finnas resurser i form av personer som underhåller testerna. Det krävs också resurser i form av hårdvara för att köra testerna kontinuerligt. Ju oftare testerna kan köras, desto snabbare får utvecklarna respons på sitt arbete.

Framtida förbättring av testerna kunde vara att testa mot verifierad CANopen-hårdvara eller att använda ett CANopen-bibliotek för Java för att implementera en CANopen-simulator. Testerna kunde också utvidgas att omfatta problemsituationer och icke-funktionella tester som prestanda och stresstestning.

7 Källförteckning

CAN in Automation (CiA). (2007)
CiA 301 Draft Standard Proposal. Version: 4.2.0

Git, 2018. [Online] https://git-scm.com/ (hämtat 20.3.2018)

Graham, D & Fewster, M (2011)
Experiences of test automation : case studies of software test automation
Addison-Wesley
ISBN 978-0-321-75406-6

Jetbrains, 2018. [Online] https://www.jetbrains.com/idea/ (hämtat 20.3.2018)

Kvaser, 2018. [Online] https://www.kvaser.com/ (hämtat 20.3.2018)

Pfeiffer, O & Ayre, A & Keydel, C (2003)
Embedded Networking with CAN and CANopen
Copperhill Technologies Corporation
ISBN 978-0-9765116-2-5

Sharma, M (2017)
Software testing 2020 : preparing for new roles
CRC Press
ISBN 978-1-4987-8887-8

TestNG, 2018. [Online] http://testng.org/ (hämtat 13.3.2018)

Wapice, 2017. [Online] https://www.wapice.com/company (hämtat 2.5.2017)