CHALMERS

Diagnostikverktyg för tolkning av kommunikation

Examensarbete

ALESANDRO SANCHEZ
MARTIN SONESSON

Data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige, 2012

Innehållet i detta häfte är skyddat enligt Lagen om upphovsrätt, 1960:729, och får inte reproduceras eller spridas i någon form utan medgivande av för fattaren. Förbudet gäller hela verket såväl som delar av verket och inkluderar lagring i elektroniska och magnetiska media, visning på bildskärm samt bandupptagning.

© Alesandro Sanchez, Martin Sonesson, Göteborg 2012

Abstract

The work done that this report will focus on is about figuring out a way to inspect and analyze communication between components in an automatized welding system. The purpose is to make troubleshooting easier and be able to quickly gain a good overview over the communication. The communication is sent over CAN or EtherNet/IP. This means that two different solutions/tools will have to be developed in order to achieve the goal. The solution for analyzing Ethernet-communication ended up being creating a dissector for Wireshark (an open source software used to read network communication) in order to read the data sent over EtherNet/IP. To analyze the CAN-communication a program was created in C# which is meant to be used together with a CAN-to-USB adapter of the brand IXXAT in order to capture CAN-messages and plot them in graphs.

Sammanfattning

Arbetet som den här rapporten redovisar handlar om att ta fram ett sätt för att granska och analysera kommunikation mellan komponenter som utgör svetsutrustning. Syftet är att underlätta felsökning och snabbt kunna få en bra överblick över kommunikationen. Kommunikation sänds antingen över CAN eller över EtherNet/IP. Alltså krävs då två olika lösningar/verktyg för att kunna åstadkomma detta. Lösningen blev att skapa en dissector för Wireshark (ett open source program som används för att läsa av nätverkskommunikation) för att kunna läsa av och tolka data som sänds över EtherNet/IP. För CAN-delen skapades ett program i C# som är tänkt att användas ihop med en CAN-to-USB adapter av märket IXXAT för att fånga CAN-meddelanden och kunna visa upp de i grafer.

Innehåll

1. Inledning 7
1.1 Bakgrund 7
1.2 Problem 7
1.3 Syfte 8
2. Metod 9
2.1 Efterforskningar 9
2.1.1 Ethernet-trafik 9
2.1.2 CAN-trafik 10
2.1.3 Sen efterforskning 10
2.2 Planerad Lösning 11
2.2.1 Wireshark dissector 11
2.2.2 CAN 11
2.2.3 Obesvarat problem 11
2.2.4 Veckoplanering 12
3. Teknisk Bakgrund 13
3.1 Svetsning 13
3.2 Svetsutrustning och dess komponenter 13
3.2.1 Robotarmen 13
3.2.2 Strömkällan och svetsdataenheten 13
3.2.3 Robotkontrollern 14
3.3 Kommunikation 14
3.3.1 CAN 14
3.3.2 Ethernet 14
3.3.3 CIP 14
3.3.6 Minnesmodell 15
3.4 Wireshark 16
3.4.1 Dissectors 16
3.4.2 PCAP-filformat 17
3.5 Inläsning av CAN-trafik 17
3.5.1 CAN-to-USB adaptrar 17
3.5.2 CSV-Loggar 18
3.5.3 IXXAT API 18
3.6 Visual Studio 19
3.7 C# 20
3.7.1 Zedgraph 20
4. Genomförande 21

4.1 Efterforskning.....21
4.2 Vecka 1.....21
4.3 Vecka 2.....23
4.4 Vecka 3.....25
4.5 Vecka 4.....27
4.6 Vecka 5.....28
5. Resultat.....30
6. Slutsats.....31
6.1 Resumé.....31
6.2 Kritisk diskussion.....31
6.3 Generaliseringar.....32
6.4 Fortsatt forskning.....32
6.4.1 Mer än bara en CAN avläsare.....32
6.4.2 Esab CAN Message Analyzer ersätter Wireshark.....33
Referenser.....34
Appendix.....35

Figurförteckning

Figur 1. Överblick över veckoplaneringen för arbetet. 12

Figur 2. En exempelbild över Wiresharks interface som visar hur Wireshark representerar paketens data för användaren. 16

Figur 3. Pcap-filformatets struktur. 17

Figur 4. En CAN-to-USB adapter av market IXXAT. 18

Figur 5. En överblick över det grafiska användargränssnittet för Visual Studio 2010 Professional. 19

Figur 6. En bild som sammanfattar hur ENIP2- och ESMM-dissectorerna hänger ihop. 22

Figur 7. Ett exempel på hur fångade paket sedan visas upp i Wiresharks gränssnitt samt hur dess data tolkas med hjälp av ESMM och representeras under info-kolumnen. 22

Figur 8. Den första versionen av CMA (Can Message Analyzer). 23

Figur 9. En förklaring på hur ENIP2-, ESMM- och CMAP- dissectorerna samarbetar. 24

Figur 10. Ett exempel på hur det kan se ut när man plottar grafer med Wiresharks IO Graph funktion. 25

Figur 11. Graph Picker, det fönster i ECMA där användaren specifierar vilka värden och flaggor som man vill plotta i en graf. 26

Figur 12. Ett exempel på hur en Zedgraph-graf kan se ut. 26

Figur 13. Skärmdump av ESMM Plugin Installer – installern som används för att implementera ESMM-pluginet till Wireshark. 27

Figur 14. En överblick över den högra panelen i ECMA som representerar trafiken i realtid. 28

Figur 15. Hur det färdiga programmet ECMA till sist såg ut. 29

1. Inledning

1.1 Bakgrund

Inom industrier och fabriker där man har behov av att svetsa ihop större metallblock krävs oftast god precision och ett kliniskt utförande för att få ett perfekt resultat. För vissa ändamål räcker det med en handhållen svetspistol för att uppnå önskat resultat, men för vissa mer krävande och mer komplexa uppgifter krävs det att man har ett automatiserat system bestående av en svetsrobot samt andra komponenter som arbetar tillsammans för att svetsa felfritt och klara av uppgifter som antingen är för tidskrävande eller för svåra för hand.

Sådan svetsutrustning består av flera komponenter som kommunicerar med varandra och som utför olika uppgifter. En person som använder svetsroboten förbereder den genom att ställa in hur den vill att roboten ska svetsa. Man anger till exempel vilket material det är på svetsobjektet, vart svetsningen ska börja och vart den ska sluta. Svetssystemet gör sedan det mesta av jobbet på egen hand. Det skickas signaler fram och tillbaka med hög frekvens som anger t.ex. spännings- och strömvärden. Det är viktigt att dessa värden regleras för att undvika fel i svetsningen eller risker för användaren. Förutom dessa värden skickas även flaggor fram och tillbaka som ger information eller beordrar komponenter att utföra vissa uppgifter.

Dessa system använder sig alltså av en form av datakommunikation för att samtala med varandra. Datapaket skickas mellan komponenterna som innehåller nödvändig information. Överföringen sker antingen över DeviceNet, ett CAN-baserat protokoll, eller över EtherNet/IP, ett Ethernet-baserat protokoll. Det finns andra möjligheter, såsom CANopen, men det är dessa två som arbetet kommer att fokusera på, dvs EtherNet/IP över Ethernet samt DeviceNet över CAN.

Det finns mycket som kan gå fel i ett svetssystem och att felsöka kan vara mycket problematiskt och tidskrävande. Ett sätt för att lokalisera eventuella fel är just genom att övervaka den kommunikationen som finns mellan de olika komponenterna. Det finns sätt att övervaka kommunikationen, vare sig då den förs över DeviceNet eller över EtherNet/IP, dock så ser man då bara rådata, vilket inte säger en mycket. Ifall man vill felsöka är det alltså upp till användaren att försöka tyda rådata, läsa av information över alla paket och försöka komma fram till vad som kan ha gått fel. Detta är väldigt tidskrävande, speciellt ifall sessionen som man undersöker sträcker sig över en lång tidsperiod. Eftersom paket skickas med så pass hög frekvens så har man redan efter någon minut flera hundratusentals paket.

1.2 Problem

Problemområdet är alltså att kunna felsöka och analysera kommunikationen mellan de olika komponenterna. Arbetet den här rapporten handlar om är följande frågeställning.

- Hur kan man underlätta granskning av kommunikation mellan komponenterna på ett effektivt sätt? Detta oavsett om den går över Ethernet eller över CAN samt även om sessionen består av stora mängder paket?

1.3 Syfte

Syftet med att underlätta analys av kommunikation är huvudsakligen för att förenkla felsökning i ett svetssystem genom att göra det mindre tids- och resurskrävande. Som redan beskrivits så är det i nuläget väldigt svårt att effektivt kunna granska den och komma fram till fel i systemet. Ifall en tekniker försöker felsöka ett svetssystem som har krånglat så är det väldigt viktigt för honom att på ett snabbt och smidigt sätt kunna granska informationen för att sedan tänka ut vad som är problemets orsak och vad man kan göra åt det. Att granska informationen i denna bemärkelse kan t.ex. vara att se när en viss flagga tänds, eller se ifall ett visst värde, till exempel spänningen, överstiger det normala. Ifall det finns ett snabbare och effektivare sätt vinner man mycket tid då det hade gått mycket fortare att snabbt komma fram till orsaken till problemet. Genom att kraftigt reducera tiden det tar för att utföra en felsökning tjänar man även på det rent ekonomiskt.

Den slutgiltiga lösningen som krävs i detta fall behöver dock inte vara begränsat till att bara kunna användas vid felsökning. Det kan finnas andra situationer där det kan vara nyttigt att snabbt kunna få en överblick i svetssystemets kommunikation. Det behöver inte alltid ha att göra med att det finns ett känt fel i systemet, utan man kanske ibland av andra orsaker är intresserad av att till exempel se hur värden förändras under en session.

2. Metod

2.1 Efterforskningar

För att komma fram till en bra lösning på problemet så har en hel del efterforskning behövts. Det var mycket ny information och det fanns också flera olika sätt att lösa vissa delar, vilket innebar att det behövdes att man kollade upp vad som var mest gynnsamt.
Då kommunikationen som ska analyseras skickas antingen över CAN eller över Ethernet delades efterforskningen upp i Ethernet- och CAN-trafik.

2.1.1 Ethernet-trafik

Det föreslogs från Esabs sida att Wireshark kunde användas för att uppnå en lösning. Med detta förslag i åtanke var man tvungen att ta reda ifall möjligheten fanns att använda Wireshark som en del av lösningen samt hur det fungerar. Wireshark är ett open source program som används för att kunna läsa av nätverkskommunikation. Att använda Wireshark för att kunna analysera kommunikationen som skickades över Ethernet var något som krävde en del efterforskning. Då Wireshark är ett open source program betyder det att det är möjligt för alla att modifiera Wiresharks källkod. Så första delen av efterforskningen blev att undersöka om det var möjligt att på något sätt skapa/modifiera kod som då skulle få Wireshark att inte bara visa datakommunikationen som fördes över EtherNet/IP som rådata, utan även tolka denna kommunikation.

Första steget var självklart, att kolla Wiresharks officiella hemsida (www.wireshark.org). Det visade sig att allt man behövde veta fanns där. Sidan innehöll en guide som beskrev i detalj hur man arbetar med Wiresharks källkod.

För att tyda data som skickas över nätverk så använder Wireshark så kallade ”dissectors”. När Wireshark fångar data så kollar den vilka rutiner det finns i programmet som har registrerat sig att kunna tolka denna. Sådana rutiner kallas för dissectors. Där beskrevs det bland annat på vilka sätt man skrev sådana rutiner samt hur man kunde registrera att de kunde tolka en viss slags kommunikation. Då var det klart att det fanns möjlighet att skriva en sådan rutin som skulle kunna tolka den data som skickades över EtherNet/IP. Det beskrevs bland annat hur man lyckades att få den här koden som plugin.

Efterforskningen för EtherNet/IP var därför nästan färdig. Det sista man behövde ta reda på var vilka fördelar och nackdelar det fanns med att skapa dissectorn som plugin. Fördelen var att efter att man skapat koden och sedan kompilerat Wireshark, så skapas en dll fil som kan flyttas till vilken Wireshark installation som helst (kan dock i vissa fall begränsas av hur gammal Wireshark versionen som är installerad är) och då kan Wireshark använda denna dissector. Om det inte skapades som plugin så räcker det inte med att flytta enbart den filen, utan man måste flytta hela Wireshark mappen om man skulle vilja använda det på en ny dator. Alltså måste man installera Wireshark igen på den datorn och inte bara en fil. Dessutom så krävs det att man uppdaterar Wiresharks källkod och kompilerar om varje gång det finns en ny version av Wireshark ute.
Nackdelarna som fanns var bland annat att man inte fick tillgång till vissa resurser i Wireshark om dissectorn är ett plugin. Dessa var inte nödvändiga och därför så vägde fördelarna mer än nackdelarna. Av denna anledning togs beslutet att dissectorn bör skapas

som ett plugin. Det förenklar för användaren och det reducerar risken för problem och krångel som hade inneburit då nyare versioner av Wireshark kommer ut.

Ett eget skapat protokoll användes för att testa skapandet av dissectors. Protokollet bestod av 5 bytes. Första byten innehöll flaggor, med detta menar man att varje bit betydde någonting. T ex så betydde första biten ”Starting”, andra ”Running” mm.. De andra byten hade bara data som skulle läsas antingen som i decimalt värde eller hexadecimalt. När protokollet var bestämts gjordes det en test dissector som skulle tolka datan som skickades uppbyggd i det protokollet. Med hjälp av ett egenutvecklat program sändes paket genom nätverket till samma dator som var strukturerade enligt detta test-protokoll. Detta var en succé och därför var EtherNet/IP efterforskningen då färdig.

2.1.2 CAN-trafik

För CAN-delen var det också mycket ny information och nya tekniker som krävde vidare granskning. Till en början var ju en av de ursprungliga frågorna ifall vi kunde tillämpa EtherNet/IP-lösningen, dvs. Wireshark-dissectorn, som en del av lösningen till CAN-delen. Efter efterforskning om CAN visade det sig att det inte var möjligt, i alla fall inte direkt då Wireshark inte lyssnar på CAN trafik. Till en början krävs det extra hårdvara för att över huvud taget kunna upprätta en kommunikation mellan CAN och dator. Det skulle sen krävas att få Wireshark att kunna på något sätt läsa CAN kommunikationen. Antingen ändrar man i Wiresharks källkod eller så ”lyfter” man kommunikationen från CAN och för den därefter över Ethernet så att Wireshark på så sätt kan fånga den. Exempelvis hade man kunnat göra ett program som läser av meddelanden och sedan skickar dem vidare över nätverket. Det gjordes vidare efterforskningar om den här möjligheten.

Efter diskussion med beställaren ställdes frågan hur de hittills hade gjort för att läsa och tolka CAN-kommunikation. Det framgick att det fanns möjlighet att spara loggar och sen arbeta med dem. Därför så fanns det nu en ny möjlighet, att använda dessa loggar. Första tanken var att man kunde tyda loggarna och sedan skicka över Ethernet datan så att den fångas upp av Wireshark. En annan möjlighet var att omvandla loggfilerna till ett format som Wireshark förstår. Det senare visade sig vara möjligt och då det är en bättre lösning än att fejka trafik så fortsatte man att förstå den biten.

Under samtalet med beställaren så visade det sig att CAN-to-USB adaptrarna av märket IXXAT som de använder ger en möjligheten att skapa egna program som kan lyssna dess trafik. Därför undersökte man även detta. IXXAT’s dokumentation om detta visade att det var möjligt att skriva program i C och även i .NET plattformen. Det fanns även en del exempel program som visade hur man kunde skriva sitt eget program. Med IXXAT’s API fanns alltså den hjälp som behövdes för att kunna utveckla egna program som kan samtala med CAN-to-USB adaptern.

Även en del efterforskning om CAN gjordes. Med allt detta så räknades då efterforskningen om CAN delen tillräcklig.

2.1.3 Sen efterforskning

Längre in under arbetets gång (se ”Vecka 2” under kapitel 4) föreslogs det av beställaren att man skulle kunna rita upp grafer som enkelt och tydligt kunde rita upp informationen för användaren. Även detta krävde efterforskning då vi behövde två olika sätt, en för Wireshark och en för CAN-delen. Det visade sig att Wireshark har redan en inbyggd funktion för

skapandet av grafer och för CAN-delen så fanns det publika bibliotek till .NET som kunde skapa grafer.

2.2 Planerad Lösning

Efter efterforskningsfasen framgick en potentiell lösning till problemet som beskrivs i 1.2. Denna hypotetiska lösning består av två faser. Den första skulle innebära att skapa ett plugin till Wireshark som skulle kunna tolka EtherNet/IP trafiken. Den andra fasen skulle innebära att skapa ett program som kunde tolka CAN-loggar och omvandla dem till PCAP-filer som Wireshark kunde förstå och senare kunna använda samma plugin för att tolka datan. Tid beräknades finnas för att även kunna få programmet att kunna fånga CAN-trafik och därmed kunna skapa PCAP-filer. De två faserna beskrivs i detalj nedan.

2.2.1 Wireshark dissector

För att kunna läsa och tolka information som skickas via EtherNet/IP valdes det att skapa en dissector som kan behandla information, vilket vi kom fram till under efterforskningsfasen beskriven i kapitel 2.1.1.
Dissectorn ska skapas som ett plugin och tanken är att det slutgiltiga resultatet ska vara en DLL-fil som kan enkelt implementeras i Wireshark.

2.2.2 CAN

För att kunna tolka CAN-kommunikationen beslutades det att man till en början skapar en dissector anpassad för CAN-meddelanden. Därefter krävs det ett program som kan fungera som mellanhand. Ett sådant program kommer att skrivas i C# och dess syfte kommer att vara att behandla CAN-kommunikation. Programmet kommer att ha möjligheten att öppna CAN-loggar och spara dessa som pcap-format så att de sedan kan behandlas av Wireshark.
När detta är klart så ska programmet även ges möjligheten att på egen hand kunna upprätta en kommunikation med en CAN-to-USB adapter av märket IXXAT, för att kunna läsa av inkommande CAN-meddelanden. Meddelanden som programmet har fångat ska sedan också kunna sparas i pcap-format.
Ifall tid och möjlighet finns kommer programmet även att ges utökad funktionalitet, dvs mer valmöjligheter för att kunna behandla informationen. Målet med detta är att göra programmet så pass bra på att analysera kommunikationen att det kan användas självt, utan att behöva blanda in Wireshark.

2.2.3 Obesvarat problem

Ett problem vid analys av kommunikation är att på grund av den snabba frekvens som paket skickas så har man efter kort tid stora mängder paket att behandla, som tidigare nämnt i kapitel 1.1. Vid analys av trafiken är det ofta nödvändigt att ta reda på var en förändring har skett. Stora intervall av identiska paket på rad är mindre intressant att granska. Det vore därför nödvändigt att tillämpa ett sätt att enkelt kunna identifiera var någonstans i trafiken som förändringar har skett bland paketen. En exakt lösning på detta kom man inte fram till under planeringsfasen, men uppskattades att kunna lösas under arbetets gång. Detta problem lämnades alltså tillsvidare obesvarat.

2.2.4 Veckoplanering

Arbetets olika moment planerades enligt följande:

	v 12	v 13	v 14	v 15	v 16	v 17	v 18	v 19	v 20	v 21	v 22
Planering/Efterforskning	■	■									
EtherNet/IP			■	■							
CAN				■	■						
Testning/Finslipningar						■					
Rapport						■	■	■	■	■	■
Slutredovisning								■	■	■	■

Figur 1. En överblick över veckoplaneringen.

De två första veckorna var enligt plan schemalagda att läggas ner på efterforskning och planering. Efterforskningen som utfördes beskrevs tidigare i kapitel 2.1.
Därefter planerades ungefär två veckor in att gå åt till EtherNet/IP-delen, dvs Wireshark-dissectorn. Ungefär två veckor planerades in åt CAN-delen. Efter det uppskattades testning och finslipningar att ta ungefär en vecka. Detta syftar på att testa både EtherNet/IP- och CAN-delen mot hårdvaran för att se så att de fungerar som tänkt, samt att införa utökad funktionalitet mest åt CAN-delen, för att göra CAN-programmet att fungera självständigt tillräckligt bra så att den inte behöver användas tillsammans med Wireshark.

För mer detaljerad information om den ursprungliga planeringen, vänligen läs den medföljande bilagan ”Planering – Detaljerad sammanfattning av den ursprungliga planeringen”.

3. Teknisk Bakgrund

3.1 Svetsning

För att kunna förstå en del av robotens agerande behöver man känna till några viktiga delar av hur svetsning går till, dessa förklaras kortfattat enligt följande:

Svetsning går ut på att smälta metalltråd på något objekt. När den smälta metalltråden har hårdnat kommer den att sätta sig fast på objektet. På detta sätt kan man sätta ihop olika objekt eller kan användas för andra ändamål.

Under den processen så får det inte komma in syre i metalltråden då det kan orsaka problem, rost bland annat. Därför är det viktigt att skydda metallen från syre. För att lyckas med detta så finns det en del metoder, men den som används av svetsutrustningen som detta projekt behandlar är gas. Olika typer av gaser kan användas för att blåsa iväg syren, t.ex. argon.

3.2 Svetsutrustning och dess komponenter

Som det tidigare beskrivits i kapitel 1.1 så är det kommunikationen i svetsutrusningen som projektet är baserat på. Svetsutrustningen består av tre viktiga delar: robotarmen, strömkällan och robotkontrollern. Den kommunikationen som är viktig för detta projekt är den som sker mellan strömkällan och robotkontrollern, dock behövs det även förståelse om robotarmen.

3.2.1 Robotarmen

Robotarmen är den komponenten som utför svetsningen. För detta så får den hjälp från andra komponenter som är kopplade till den (dock kommer robotarmen och dess extra komponenter att räknas som en komponent i sig), nämligen, svetsmunstycket, trådmataren och gasblåsaren. Svetsmunstycket tar hand om själva svetsningen samt håller även glasblåsaren och använder metalltråden som kommer från trådmataren för att kunna svetsa. Robotarmen har då som uppgift att flytta runt svetsmunstycket.

3.2.2 Strömkällan och svetsdataenheten

Även i detta fall så räknar vi två komponenter som en. Strömkällan är den komponenten som har uppgiften att ge den nödvändiga elströmmen till robotarmen så att den kan utföra svetsningen (Observera att det enbart gäller svetsningen inte robotarmens rörelser). Den tar hand då om alla tre nämnda viktiga delar av svetsningen i 3.1: Själva svetsningen, bortblåsning av syre samt trådmatning.
Strömkällan är inte bara en strömkälla utan den har sin egen processor för att kunna utföra nödvändiga beräkningar för att kunna hålla svetsningen igång samt andra viktiga beräkningar. De här beräkningarna varierar beroende på vad det är för material man svetsar eller andra förhållanden. Dessa variabler hämtas till strömkällan från svetsdataenheten som i sin tur också tar hand om andra beräkningar (beräkningar som inte är relevanta för denna rapport eller projektet överhuvudtaget).
En svetsdataenhet som kan hantera manuell inmatning av inställningar kan även kopplas till strömkällan.

3.2.3 Robotkontrollern

Robotkontrollern är den som styr hela svetsningsprocessen. Den bestämmer robotarmens rörelser samt när den börjar svetsa, blåsa gas, mata tråd, etc. För just svetsningsdelen så skickar den inte ovan nämnda instruktioner direkt till robotarmen utan till strömkällan som är den som styr robotarmens svetsning.

3.3 Kommunikation

Robotkontrollern och strömkällan kommunicerar med varandra genom CAN eller även genom Ethernet om hårdvaran tillåter det. Industrin i dagsläget använder oftast CIP baserade protokoll för kommunikation i deras industriella maskiner och det är fallet även i kommunikationen mellan strömkällan och robotkontrollern.
Då kommunikationen sker genom CAN är det DeviceNet som beställaren använder för att kommunicera, dock så kan de även använda CANopen, när kommunikationen sker genom Ethernet så är det EtherNet/IP som gäller. Detta avsnitt kommer att ge läsaren en kort förklaring på vad dessa protokoll handlar om.

3.3.1 CAN

CAN, som står för Controller Area Network, är en nätverksstandard som används inom många industrier, men ursprungligen inom fordonsindustrin. CAN kan skicka data upp till 1Mbit/sekund och data överförs genom sekvenser på 8 bytes var. Ett CAN-meddelande består av flera delar, bl.a. ett ID-nummer som bestämmer meddelandets prioritet. I ett CAN nätverk innebär lägre ID-nummer en ökad prioritet. Varje nod på en CAN-buss kan ta emot och skicka meddelanden, men inte samtidigt. Om två meddelanden skickas från olika noder samtidigt prioriteras det meddelande med lägst ID-nummer.
Fördelar med CAN och anledning till att det blivit så pass populärt inom många industrier är dess pålitlighet, liten kostnad (tack vare de små systemkraven) och att CAN är öppen teknologi. En av CANs begränsningar är bitraten på max 1Mbit/sekund, fast denna maxgräns räcker mer än väl inom många områden där CAN används.

3.3.2 Ethernet

Ethernet är en samling av teknologier som tillhandahåller möjligheten att låta datorer kommunicera med varandra genom kabel i ett lokalt nätverk. Ett av kännetecknen för Ethernet är dess höga hastighet (100 gigabits/sek), låga pris och höga tillförlitlighet. Detta har gjort Ethernet till ett populärt alternativ som idag används i majoriteten av alla lokala nätverk.

3.3.3 CIP

CIP står för Common Industrial Protocol och det är ett kommunikationsprotokoll som används för att överföra data mellan två enheter. Med CIP-protokollet representeras varje nätverksenhet som en grupp av objekt. Objekten kan grupperas in i tre olika kategorier: Required Objects, Applications Object och till sist Vendor Specific Objects.

Required Objects är obligatoriska och måste inkluderas i varje CIP-enhet. Required Objects består i sin tur av tre undergrupper av objekt; **Identity Object**, **Message Router Object** och **Network Object**. Identity Object är det objekt som innehåller relevant data för att identifikation. Datan är uppdelad i attribut såsom till exempel vendor ID, datum för tillverkaren, enhetens serienummer etc.

Message Router Object är ett objekt vars uppgift är att skicka vidare meddelanden från objekt till objekt i en enhet.
Network Object innehåller data som redogör om den fysiska anslutningen mellan enhet och objekt. För en CIP-enhet över DeviceNet innehållerr Network Object MacID samt övrig data som beskriver gränssnittet för CAN-nätet.

Application Objects är objekt som definierar datan som är sammanfattad av enheten. Dessa objekt är specifika och beror på enhetens typ och funktion. Till exempel en viss sorts enhet kan ha attribut som redogör om frekvens och ström medans en annan enhet kan ha attribut såsom upplösning och input-värden.

Vendor Specific Objects är objekt som inte ingår i profilen av själva enheten och kallas därför Vendor Specific. Dessa objekt innehåller attribut som är specifika för enheten. CIP-protokollet möjliggör åtkomst till dessa specifika objekt. Datan är helt och hållet utvald av den specifika enhetens tillverkare och ordning samt struktur väljs utefter vad som passar bäst för enheten i fråga.

Här nedan presenteras två olika CIP-baserade protokoll, nämligen EtherNet/IP, som går över Ethernet, samt DeviceNet, som går över CAN.

OBS: *Kapitel 3.3.3 som handlar om CIP-protokollet är delvis taget och sedan översatt från en annan källa. Var god se källförteckningen artikel nr 10.*

3.3.4 EtherNet/IP

EtherNet/IP, där IP står för Industrial Protocol och bör därmed inte förväxlas med Internet Protocol, är ett Application-layer protokoll som sänds genom TCP/IP paket. EtherNet/IP redogör alltså bara hur data i ett TCP eller UDP paket är uppbyggt och strukturerat.

Första byten i ett EtherNet/IP paket berättar antal objekt (kom ihåg EtherNet/IP är CIP-baserat) som finns i paketet och sedan följer objekten. Varje objekt börjar med en första byte som berättar vad det är för slags objekt, nästa byte berättar längden av objektet i bytes och sedan objektets data.

3.3.5 DeviceNet

DeviceNet är ett Application-level protokoll som används inom industriella applikationer. DeviceNet kopplar ihop enheter såsom sensorer med enheter såsom programmerbara kontrollers. DeviceNet är byggt på CAN kommunikation och är en kombination mellan CIP protkollet och CANs Physical layer.

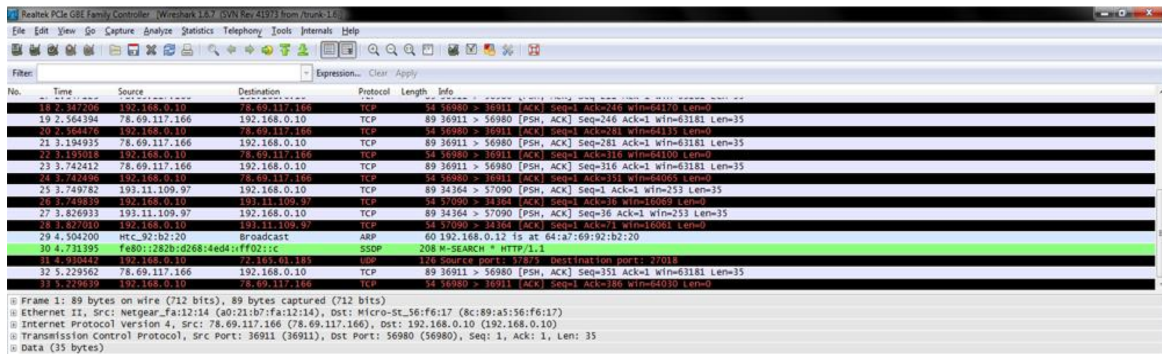
3.3.6 Minnesmodell

Minnesmodellen syftar på den struktur och uppbyggnad paketen, som arbetet har inriktat sig på, har. Minnesmodellen är en standard som används för DeviceNet och den kan återfinnas i Aristo W8$_2$ Service Manual på sidan 31 (se källförteckning, artikel nr 1).

Minnesmodellen är 16 bytes stor och den innehåller både flaggor och värden. Strukturen skiljer sig beroende på ifall det gäller paket som går in i svetsroboten från kontrollern eller tvärtom. Det finns dessvärre ingen information i själva minnesmodellen som talar om åt vilket riktning paketet kommer eller vem som skickade den.

3.4 Wireshark

Wireshark är ett program som används för att lyssna på och analysera nätverkskommunikation. Wireshark kan fånga paket och läsa av dess innehåll. Dess användningsområde är stor men det är bl. a. ett speciellt effektivt verktyg när det kommer till nätverksdiagnostik och felsökning. Wireshark erbjuder många olika funktioner såsom att filtrera paket och rita grafer över datan den fångar. Wireshark känner också till de flesta av de vanligast använda nätverksprotokollen och kan på så sätt känna igen vilket protokoll som används av ett visst paket, t ex. TCP, UDP, http, etc.
Wireshark finns tillgängligt för flera plattformar såsom Windows, Mac OS X, Solaris, BSD och Linux. Programmet är skrivet i C och är ett open source program som använder sig av GNU General Public License. Detta innebär att Wiresharks källkod är öppen för allmänheten och vem som helst får titta på den samt modifiera källkoden för att skapa egna versioner av Wireshark. Wiresharks utvecklare uppmuntrar till vidareutveckling och har på sin officiella hemsida (www.wireshark.org) hjälpdokumentation för vidareutveckling av Wireshark och skapande av dissectorer etc.



Figur 2. Ett exempel på hur Wiresharks gränssnitt ser ut

3.4.1 Dissectors

Rutiner i Wireshark programmet som används för att tolka och tyda ett visst fördefinierat protokoll kallas för ”dissectorer”. Det engelska ordet ”dissect” betyder bokstavligen att dissekera, separera på eller att noggrant granska.
En dissector känner alltså igen ett visst protokoll, bl.a. genom att titta på paketets struktur men i de flesta fall genom att titta på vilken port som paketet skickats igenom. Ifall en dissector känner igen ett paket som tillhör det angivna protokollet kommer dissectorn att behandla paketets information och dela upp informationen utefter de olika delarna som paketet består av. Dessa delar kommer sedan att skrivas upp i Wiresharks användargränssnitt och göra det möjligt för användaren att kolla igenom paketets information. Med andra ord, dissectorn tyder datan och gör det möjligt för användaren att granska paket utan att själv behöva tyda rådatan, vilket är problematiskt.
Wireshark har på sin hemsida en sektion som heter ”Developers Guide” (se källförteckning, artikel nr 4) där information finns tillgänglig om hur man skapar och tillämpar en egen dissector. Det finns möjlighet att skapa en dissector som plugin (tillägg) till Wireshark. I sådana fall då dissectorn skrivits som ett tillägg så kommer det, efter kompilering av Wireshark, att finnas som en DLL-fil i Wiresharks mapp, det är denna fil som en användare

sedan kan implementera i sin egen version av Wireshark för att använda dissectorn som ett plugin.

3.4.2 PCAP-filformat

Pcap, som står för packet capture, är ett API (Application programming interface) som används för att tillhandahålla data som utgörs av en session av fångad nätverkstrafik och dess paket. Pcap kommer i två huvudsakliga versioner, libpcap som oftast används i Unix-system och WinPcap som är en Windows portning av libpcap.
Filformatet pcap är alltså en sorts fil som innehåller fångade paket och dess data. Wireshark har möjlighet att läsa dessa filer samt skapa egna efter att ha fångat nätverkstrafik. En pcap-fil består av en global header som sedan följs av segment av paket, eller inga segment alls. Varje sådant segment består i sin tur av en ”packet header” och en ”packet data”. Varje pakets header innehåller bl.a. tidsstämpel för meddelandet. Den globala headern innehåller global information såsom tidsstämpel för hela sessionen samt vilken Link layer kommunikationen fångades ifrån. Vilken Link layer kommunikation som används definieras av vissa standardvärden som representerar protokoll. Dock finns det rum för att specificera egna protokoll i Wireshark inom området som kallas DLT 147 – DLT 162.

Figur 3. *Pcap's struktur*

3.5 Inläsning av CAN-trafik

3.5.1 CAN-to-USB adaptrar

I det här arbetet har två olika sorters adaptrar använts. Dels de adaptrar som fungerar som dataloggers, som kan spara och logga information. Förutom dessa har arbetet även inriktat sig mycket på adaptrar av märket IXXAT som erbjuder mer än bara loggning. Mer information om dessa sorters adaptrar följer härmed.

3.5.1.1 Datalogger

En "data logger" är en CAN-to-USB adapter som är speciellt framtagen för att kunna logga sessioner av CAN-trafik. De typer av dataloggning adaptrar som kommer att tas upp i detta arbete är av märket Kvaser (se källförteckning, artikel nr 8), specifikt Kvaser Memorator. Denna adapter kan fånga CAN meddelanden och sedan skriva en logg över sessionen. Loggen sparas genom Kvasers mjukvara och går att konvertera till flera olika format. Filformatet som detta arbete kommer att behandla är CSV (se kapitel 3.5.2 för mer information om csv).

3.5.1.2 Ixxat

De CAN-to-USB adaptrar som detta arbete i huvudsak kommer att fokusera på är de av märket IXXAT. Dessa adaptrar kan precis som andra adaptrar som de nämnda ovan fånga CAN-trafik för analys, skillnaden är dock att de erbjuder möjligheten att göra detta i realtid.

Kvaser's Data Logging adaptrar t.ex, som nämns ovan, kopplas in till den specifika hårdvaran, fångar en session och sessionen behandlas senare i dess mjukvara genom att t.ex. skapa en logg eller liknande. IXXAT's adaptrar tillåter analys av CAN-trafik samtidigt som man behandlar information. Detta gör det möjligt att med hjälp av en sådan adapter skapa ett program som på något sätt behandlar CAN-meddelanden i realtid.

Figur 4. *En CAN-to-USB adapter av market Ixxat*

3.5.2 CSV-Loggar

Det finns möjlighet att spara loggar över en CAN-session med vissa CAN-to-USB adaptrar såsom Kvaser Memorator (se kapitel 3.5.1). Dessa loggar innehåller information över varje meddelande i kronologisk ordning. Loggarna redogör för varje meddelandes tidsstämpel, avsändar-ID, datastorlek och förstås även själva rådatan.
Att kunna spara dessa loggar är en viktig funktionalitet i felsökningssyfte. Ifall man sparar en logg över en session som man vill analysera kan man sedan granska loggen för att försöka komma fram till ett svar. Dock blir ju detta högst problematiskt, speciellt med en stor logg som täcker en längre session, utan ett verktyg som kan bearbeta loggen och bearbeta dess information. Dessa CAN-loggar går att spara i olika filformat som alla har lite annorlunda struktur. Den sortens loggar som detta arbete kommer att arbeta med är dock av filformatet csv. CSV-loggarna skrivs i plain-text (ej binärt) och kan därför enkelt öppnas i t.ex. Notepad. På varje rad definieras ett CAN-meddelande vars delar står i flera kolumner. Olika delar separeras av semikolon eller komma. På grund av dess struktur är de enkla att arbeta med samt skriva program som kan tolka och dela upp de olika delarna.

3.5.3 IXXAT API

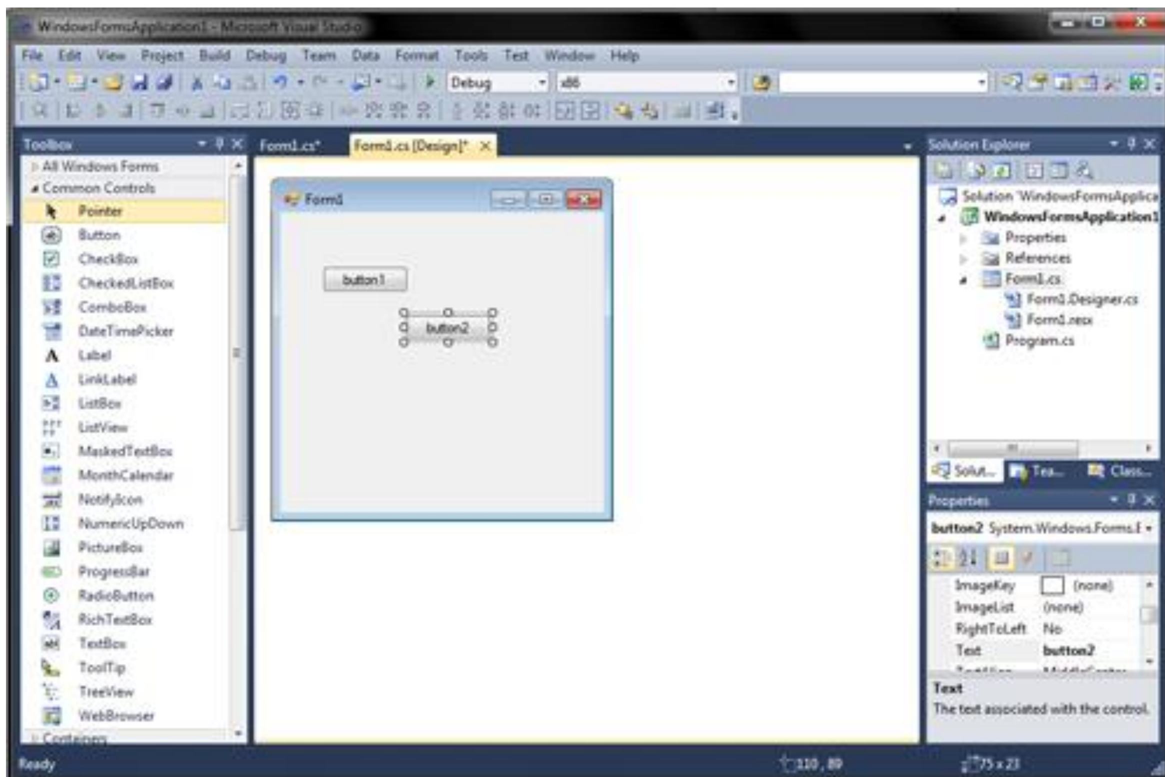
IXXAT har på sin hemsida hjälpdokumentation och API för utvecklare som vill använda deras produkter. I detta arbete har deras ”Programmer's Manual” för IXXAT CAN-to-USB adaptrar varit till stor nytta (se källförteckning, artikel nr 2).
Denna dokumentation redogör hur man utvecklar ett program som använder sig av en CAN-to-USB adapter av märket IXXAT. Detta omfattar att skriva ett program som kan hitta adaptern, sedan kunna upprätta en anslutning med den och slutligen kunna starta en session där man fångar eller skickar CAN-meddelanden.
För att kunna använda detta krävs det att man har installerat drivrutiner för den adapter man vill använda. Efter installation av drivrutiner finns det en DLL-fil tillgänglig som innehåller nödvändiga funktioner som används vid kommunikation med CAN-to-USB adaptern. Det är möjligt att skapa ett program, i C eller i .NET-relaterade programmeringsspråk, som använder sig direkt av DLL-filen och anropar funktioner direkt från den.

I ”Programmer's Manual” finns även ett API över funktioner som är nödvändiga. Det finns även på IXXAT's hemsida färdigskrivna exempelprogram i C som kan utföra enkla

operationer som att skicka eller läsa ett meddelande och dessa körs i kommandotolken och är skrivna i C. Att studera denna färdigskrivna kod är också en viktig del för att förstå hur det går till att skriva egna program som kommunicerar med adaptern.

3.6 Visual Studio

Microsoft Visual Studio är en utvecklingsmiljö som finns tillgänglig för Windows operativsystem. Det finns både en gratisversion (Express) samt betalversioner som har större funktionalitet. Visual Studio stödjer flera olika programmeringsspråk från grunden såsom C/C++, VB.NET, C# och F#. Med separata tillägg kan även Visual Studio stödja Python, M och Ruby.



Figur 5. *Visual Studio 2010 Professional. I bilden visas programmet designer som tillåter snabbt och enkelt skapande av grafiska användargränssnitt.*

Visual Studio erbjuder flera funktioner för en utvecklare som är mycket användbara, inte minst en grafisk editor. Denna funktion tillåter användaren att snabbt kunna skapa ett grafiskt användargränssnitt genom att använda färdiga komponenter, kallade controls, som inkluderar knappar, textfält, färdiga popup-fönster för att öppna och spara filer etc. Användaren definierar sedan i koden vad komponenter ska ha för uppgift och hur de ska kunna användas. Editorn gör alltså så att skapandet av ett gränssnitt går mycket fortare än vad det hade gjort om man kodade det från grunden. Detta är mycket effektivt ifall man har hög tidspress och

inte hinner lägga ner mycket tid och kraft på GUI delen men ändå vill att den ska se respektabel ut. Det finns även möjlighet att skapa egna controls, komponenter som är egenskapade och inriktade på en specifik uppgift.

3.7 C#

C# (vanligen kallat C-sharp) är ett objektorienterat programmeringsspråk utvecklat av Microsoft. C# är en del av .NET-plattformen och utvecklat genom C++. C# liknar Java även om C# är mera plattformsberoende än Java då utvecklingsverktyg bara finns tillgängliga för Windows operativsystem.

3.7.1 Zedgraph

Zedgraph är ett klassbibliotek vars källkod är öppen för allmänheten och tillåts för privatpersoner att använda det i sina egna projekt. Zedgraph är kompatibelt med .NET-baserade språk och kan användas ihop med ett C# projekt där den fungerar som en grafisk komponent i Visual Studios GUI editor.
Zedgraph har också ett öppet API med mängder av användbara funktioner som ger användaren mycket valfrihet när det kommer till implementeringen av Zedgraph i sitt projekt.

4. Genomförande

4.1 Efterforskning

De två första veckorna av arbetet lades till efterforskning. Resultat av denna beskrevs tidigare i avsnitt 2.1. När efterforskningen var färdig trodde man sig veta hur man skulle kunna lösa problemet (läs 1.2) och därmed skrev man en planering på hur man skulle ta sig till väga. Härifrån följer utförlig beskrivning av arbetet som genomfördes för resterande veckor. Arbetet försöktes att följa planeringen, vilket till den största delen kunde utföras. Planeringen finns beskriven i avsnitt 2.2.

4.2 Vecka 1

Första veckan började med att fånga trafik från svetsutrustningen, både den genom CAN och den genom EtherNet/IP (ENIP). CAN trafiken lagrades med en vanlig datalogger (läs 3.5.1) och ENIP trafiken fångades med Wireshark (läs 3.4). Enbart ENIP loggfilerna behandlades denna vecka.

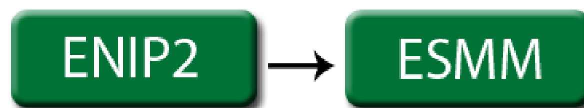
Undersökning av ENIP loggarna visade att datan som skickades genom ENIP inte bestod enbart av 16 bytes utan den innehöll även andra bytes. Tack vare de extra bytes kunde man skilja på IN- och UT-paket, dock så förstod man inte betydelsen av dem. IN-paket innehöll 22 bytes medan UT-paket innehöll 18 bytes.

Att kunna skilja IN- och UT-paket enbart på det här sättet är inte optimalt. De extra bytes är inte ENIP-relaterade utan de är extra bytes som maskinen lägger till. Tyvärr så kunde man inte få svar från företaget som tillverkade svetsutrustningen och därför kunde man inte veta vad de andra bytes var.
Efter att noggrannare undersökning visade sig att för IN-paketen, som innehöll 22 bytes, var de extra bytes sekvensnumret. Detta var konstigt att förstå då det upptäcktes pga. att ENIP redan visar sekvensnummer och det är samma värden på de extra bytes. Dock så använder ENIP fyra bytes för sekvensnummer och här handlar det om sex extra bytes. Dock så använder den inte alla sex bytes för sekvensnummer utan enbart två bytes. Vad är då poängen med att använda två bytes som sekvensnummer om samma info redan visas i en annan del av ENIP-paketet med bättre precision? Detta lyckades man inte få svar på, inte heller för vad de andra fyra bytes då representerar.
För UT-paketen, som innehåller 18 bytes, visade sig att de två extra bytes visar att data har förändrats. Det är en två-bytes-räknare som ökar med ett varje gång ny data skickas ut. Den räknar dock inte när ny data skickades in. Kanske är de två bytes i IN-paketet som visar sekvensnummer tänkta att också räkna då datan som skickades inte var samma som i förra paketet men att det blev något fel och den räknar per paket?

Även om detta var oklart nöjde man sig med att kunna skilja på IN- och UT-paket genom att skilja på antalet bytes och då påbörjades byggandet av dissectorn. Ett problem uppkom på grund av missförstånd/otillräcklig efterforskning om ENIP i Wireshark och i allmänhet. En antagelse man gjorde var att betrakta ENIP som ett transportprotokoll som skulle bete sig mycket likt TCP. När man skapar en dissector så kan man välja vilken port man vill lyssna på och då får man den data som skickades genom TCP. I det här fallet antog man då att man kunde välja rätt port och den data som skickades genom ENIP var det enda man skulle få,

man kanske skulle vara tvungen att ange att ENIP skulle anropas först och sedan dissectorn man skapat. Det är helt enkelt så att då ENIP går genom TCP så får man istället inte bara minnesmodellen som man var ute efter utan även ENIPs payload.

Problemet löstes genom att istället för att skapa en helt ny dissector så arbetade man vidare på ENIP dissectorn och gav den möjlighet att dissecta den data som går genom ENIP. Arbetet fortsatte sedan med att flytta koden till en ny dissector istället som man kunde sedan göra som plugin. Den dissectorn delades därefter i två delar: en som behandlar ENIP, som kallades ENIP2, och en som enbart behandlar Esabs minnesmodell, den dissectorn kallades och kommer härefter att kallas som ”ESMM” (Esab Memory Model). Fördelen med detta är bl. a. möjligheten att återanvända ESMM dissectorn när man senare behandlar CAN kommunikation. Notera att ESMM själv inte kan kontrollera om ett paket tillhör minnesmodellen, utan den anropas av andra dissectorer, såsom ENIP2 och senare CMAP (beskrivs under ”Vecka 2”).



Figur 6. ENIP2 anropar ESMM för vidare dissectering

Dessa två dissectors arbetade väl ihop och ESMM var kapabel att förstå datan och visa den på ett användarvänligt sätt i Wiresharks gränssnitt och även på info-kolumnen, där man kunde då vissa vilka flaggor som var på eller av och även andra mätvärden. Resultatet blev två DLL-filer som då kunde användas som plugin till Wireshark. De filerna testades även för äldre versioner av Wireshark och det visade sig inte ge goda resultat. Anledningen till detta var att för att skapa den dissectorn som skulle behandla ENIP delen så *lånade* man mycket av den kod som redan fanns i Wireshark (Observera att koden var och är under GNU licensen). Den koden utnyttjade vissa nya resurser som finns i den senaste (inte publika stabila) versionen av Wireshark. Då dessa resurser inte var nödvändiga för dissecteringen av ENIP så valde man istället att *låna* koden från den publika versionen av Wireshark då den inte hade sådana problem utan ägnar sig helt åt dissectering av ENIP. Efter den här förändringen så kunde nu pluginen arbeta i äldre versioner samt i den senaste. Man testade enbart från version 1.6.5 till 1.7.1 (publika stabila versionen var 1.6.7).

No.	Time	Source	Destination	Protocol	Length	Info
3515	35.1016	192.168.0.2	192.168.0.1	ESMM	78	Dir = Out, Weld Busy, Arc. Ackn., WFU 1, Gas Active, Syn. Act., Inch. Act., Volt. (Meas.): 1925, Current: 209, Power: 3860
3516	35.1205	192.168.0.1	192.168.0.2	ESMM	82	Dir = In, Weld On, Gas Purge, W. Data Nr: 201, Volt.: 65535
3517	35.1215	192.168.0.2	192.168.0.1	ESMM	78	Dir = Out, Weld Busy, Arc. Ackn., WFU 1, Gas Active, Syn. Act., Inch. Act., Volt. (Meas.): 1925, Current: 209, Power: 3860

Figur 7. *Bilden nedan visar exempel på hur Wireshark sedan med hjälp av ENIP2 och ESMM dissectors kan tolka paket och under info-kolumnen visa upp relevant information som paketen innehåller för användaren.*

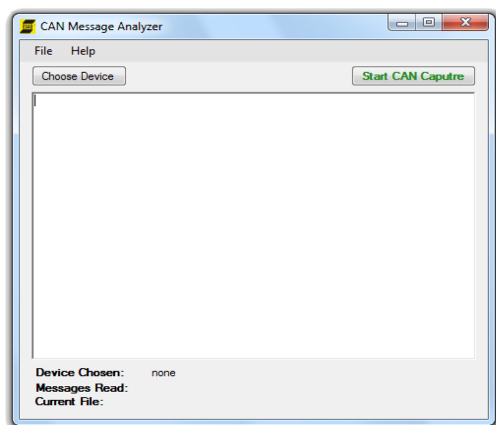
Även under denna vecka gjordes ett försök att förenkla arbetet att förstå kommunikationen för användaren genom att enbart visa de paket där förändring hade skett. Försöket gjordes genom att man försöker dissecta ett paket och lagrar info om den, när nästa paket kommer kontrollerar man om det har blivit någon skillnad. Detta blev inte lyckat då det visade sig att när Wireshark går igenom en loggfil så läser den inte paketen i rätt ordning utan till synes mer slumpmässigt. Teoretiskt sett borde det dock fungera när Wireshark läser trafik i realtid men inga tester gjordes då eftersom ifall det inte fungerar korrekt då man läser loggfiler så är det ingen bra lösning alls.

Denna vecka blev mycket framgångsrik. Mycket av det som hade planerats för denna vecka (läs bilaga ” Planering – Detaljerad sammanfattning av den ursprungliga planeringen”) kunde uppnås. Det enda som inte kunde uppnås var att kunna förenkla det problemet som finns med att det är svårt att se när data förändras (som tidigare förklarat i kapitel 1.1 så skickas väldigt många paket som innehåller samma data, t ex ”Weld On”-flaggan är fortfarande på eller spänningen ligger på samma nivå). Ett annat problem som inte kunde lösas var att skilja mellan IN och UT på ett smart sätt. Än så länge så skiljer man enbart på hur många bytes som skickas och fortfarande förstår man inte vad de resterande bytes är för några, i all fall för IN-paketen.

4.3 Vecka 2

Analysering av CAN-loggarna man fångade under Vecka 1 påbörjades. Loggfilerna innehöll flera tusen rader där varje rad representerade ett CAN-paket. CAN stödjer att man transporterar max 8 bytes per paket, vilket då innebär att hela minnesmodellen inte kan få plats i ett enda CAN-paket utan måste delas upp i minst 2 paket. Det visade sig att 3 CAN-paket behövdes för att överföra minnesmodellen. Två paket av 8 bytes samt ett tredje paket av 3 bytes. En första antagelse gjordes om att de 3 första bytes i det första paketet var överflödiga och var DeviceNet specifika. Detta efter jämförelse med det man hade fångat från Wireshark. Det visade sig senare (läs ”Vecka 5”) att detta var felaktigt. Även om det där var felaktigt kunde man fortsätta arbetet. Att skilja på IN- och UT-paket var väldigt mycket enklare den här gången jämfört med EtherNet/IP. CAN paket med lägre Id nummer har högre prioritet och detta hjälpte oss att upptäcka vilka meddelanden som var IN och vilka som var UT. Strömkällan är master i kommunikationen och har därmed lägre Id nummer.

Ett program skapades i Visual Studio som kallades då för CAN Message Analyzer som kommer nu att refereras som CMA. CMA kunde läsa loggfilerna och visa upp CAN-kommunikationen i form av en tabell. En kolumn visade paketets tidsstämpel, en annan kolumn visade paketets Id nummer och den sista kolumnen visade rådatan som skickades i CAN-paketet i hexadecimalform.

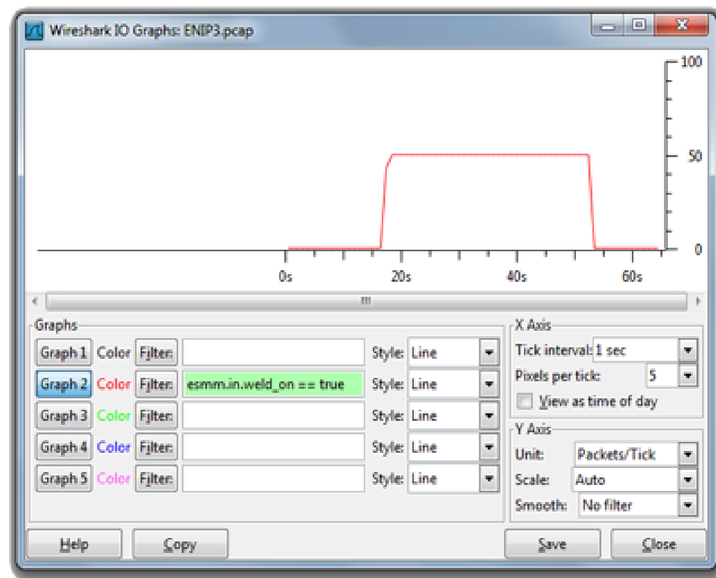


Figur 8. *Bilden till vänster visar den första versionen av CMA*

CMA utökades sedan med möjligheten att kunna skapa PCAP-filer så att Wireshark skulle kunna läsa kommunikationen. När man använder CMA för att skapa PCAP-filer så beskriver det i filen varje paket av data som mottagits förutom de paket som tillsammans transporterade minnesmodellen. I de fallen så sätter CMA ihop de 3 paketen till ett paket. CMA lägger till dessutom en extra byte som berättar om det är ett IN- eller UT-paket eller om det inte är något av dem (alltså andra CAN-meddelande och inte relaterat till minnesmodellen). Det krävdes lite förändring i ESMM dissectorn för att den skulle förstå vad som var IN och UT. Senare delades koden i två. ESMM dissectorn behandlade återigen enbart behandlingen av minnesmodellen och en ny dissector skapades som kallades för CMAP (CMA Protocol). CMAP dissectorn är det första som anropas, den avgör om det handlar om ett vanligt CAN-meddelande eller om det är ett paket som innehåller minnesmodellen och i så fall om det är ett IN- eller UT-paket. Om det är ett IN- eller UT-paket så skickar den de 16 bytes som är minnesmodellen till ESMM dissectorn. Som beskrivit under 3.4.2 så kan man sätta ett eget värde för vilket link layer protokol man använder. Det finns redan ett standard värde man kan använda för CAN men då man ville att CMAP skulle anropas så valde man ett eget värde (148) för link layer. Sedan ställer man in Wireshark så att den ska anropa CMAP då den läser PCAP-filerna skapta av CMA.

Figur 9. *I bilden ovan redovisas den slutgiltiga strukturen om hur ENIP2, ESMM samt CMAP samarbetar med varandra.*

Under veckan så kom det ett förslag från Esab om att använda grafer för att kunna lösa problemet med att kunna se när data har förändrats, som nämns i kapitel 2.2.3. Wireshark har en inbyggd funktion för att visa grafer och efter lite undersökning kunde man se att allt som krävdes för att använda den hade redan implementerats. När ESMM dissectorn skapades så gav man namn till varje del som fanns i minnesmodellen. Med hjälp av dessa namn kunde man då filtrera i grafen som Wireshark skapar. På det sättet kan man nu undersöka när vissa förändringar har skett, t ex. genom att filtrera efter ”Weld On” så kan man då se när den flaggan har varit på eller av. Se figur 10 nedan. Grafen som Wireshark visar är dock begränsad då y-axeln är i antal paket per tidsenhet format, detta räcker för flaggor men inte för andra mätvärden.



Figur 10. *Ett exmpel på Wiresharks IO Graph funktion. Här ser man "Weld On"-flaggans utveckling under en svetssession.*

I princip så hade man nu lyckats hitta lösningar till att avlyssna både CAN- och EtherNet/IP-trafik och kunna visa datan som skickas på ett användarvänligt sätt. Användningen av grafer ger användaren också möjligheten att kunna enklare se när data hade förändrats. Dock inte lika lätt när det inte är flaggor man vill undersöka. Enligt planering så skulle man dock försöka lyckas bättre än så här. Så här långt så kan man använda CMA för att översätta loggfiler till PCAP-filer men det där kräver för många steg. Man skulle kunna förkorta ett steg genom att göra CMA kapabelt till att fånga CAN-trafik och sedan skapa en PCAP-fil utifrån det. Då behöver man inte skapa en loggfil och sedan översätta den med CMA till PCAP-format.

Under efterforskningstiden (se kapitel 2.1.2) så hittade man färdig kod som kunde läsa CAN-trafik med hjälp av en IXXAT adapter. Tanken var då att under denna vecka få den koden, som var skriven i C++, till CMA. Man skulle få den koden till en DLL-fil för användning som bibliotek. Detta då det fanns där färdiga metoder som behandlade CAN-trafiken genom IXXAT adaptern. Viss modifikation behövdes för att få mer funktionalitet. Detta skrotades och det bestämdes att göra all kod i C# från grunden. IXXAT har i sin hemsida bra dokumentation om hur man skriver kod som kan arbeta med deras adaptrar så det fanns inga problem för att lyckas med sagd uppgift. Byggandet av koden påbörjades.

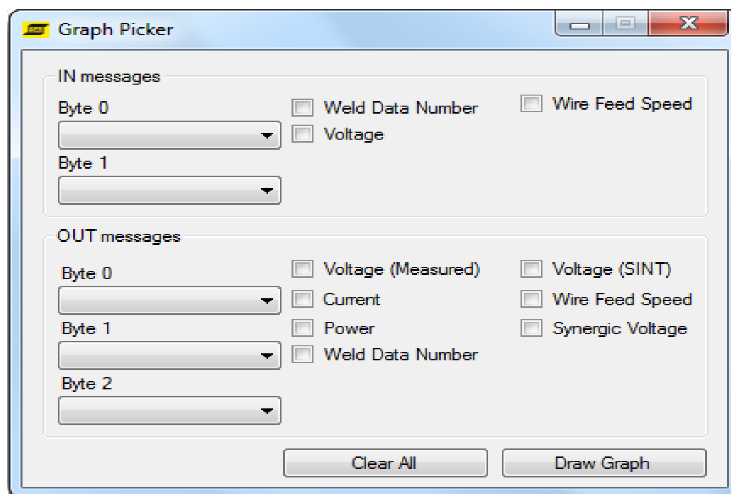
4.4 Vecka 3

Arbetet fortsatte där man lämnade det Vecka 2. CMA blev kapabel till att visa vilka IXXAT adaptrar som var kopplade mot datorn och kunde visa relevant data om dem. Man skrev vidare på koden och där angavs hur programmet skulle lyssna på CAN-trafiken. Detta kunde dock inte testas då det inte fanns tillgång till lediga svetsrobotar eller någon simuleringsmiljö. Man lämnade testningen då till ett senare tillfälle.

Då användningen av grafer i programmet inte var något planerat så hade man inte gjort någon efterforskning om grafer i C#. Därför tog man tid till att göra det under denna vecka. Visual Studio har inga färdiga funktioner för grafer så därför hittades dem på nätet. De bästa man kunde hitta var GraficDisplay och ZedGraph (se källförteckning, artikel nr 5 & 6). Efter olika

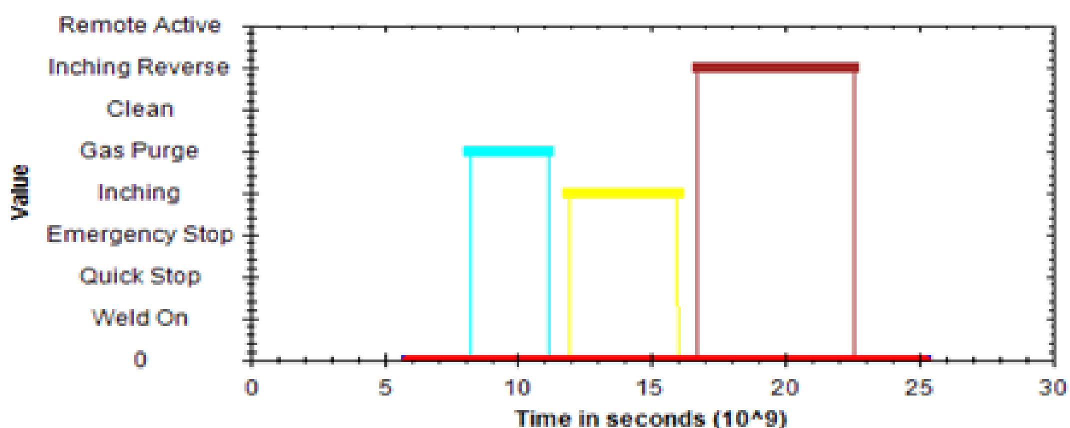
tester och analysering om dem så visade sig vara ZedGraph som var den bättre grafritaren. Detta pga. att den har bättre dokumenterad API som får en att kunna sätta sig in i den snabbare.

Arbetet fortsatte då med att skapa en ny form som kallades ”Graph Picker” (se figur 11 nedan). Den kunde anropas från CMA och därifrån gavs möjligheten att välja vilken data man ville se om kommunikationen man hade läst, vare sig det var från en loggfil eller om man hade fångat den från CMA.



Figur 11. Bilden ovan visar Graph Picker fönstret som används för att skapa grafer

Denna vecka nådde man de uppgifterna man hade satt upp att kunna lyckas med. CMA kunde visa datan på bättre sätt än Wireshark kunde och faktiskt på ett ypperligt sätt. Då man kunde skapa graferna här som man ville och inte som det var förutbestämt av något annat program så lyckades man med ett riktigt bra verktyg. Den här grafritare kunde, och kan, skapa grafer som visar aktiviteten för flaggorna och mätvärdena under tiden. Detta ger en mycket klar bild över vad som hänt under tiden man loggat trafiken.

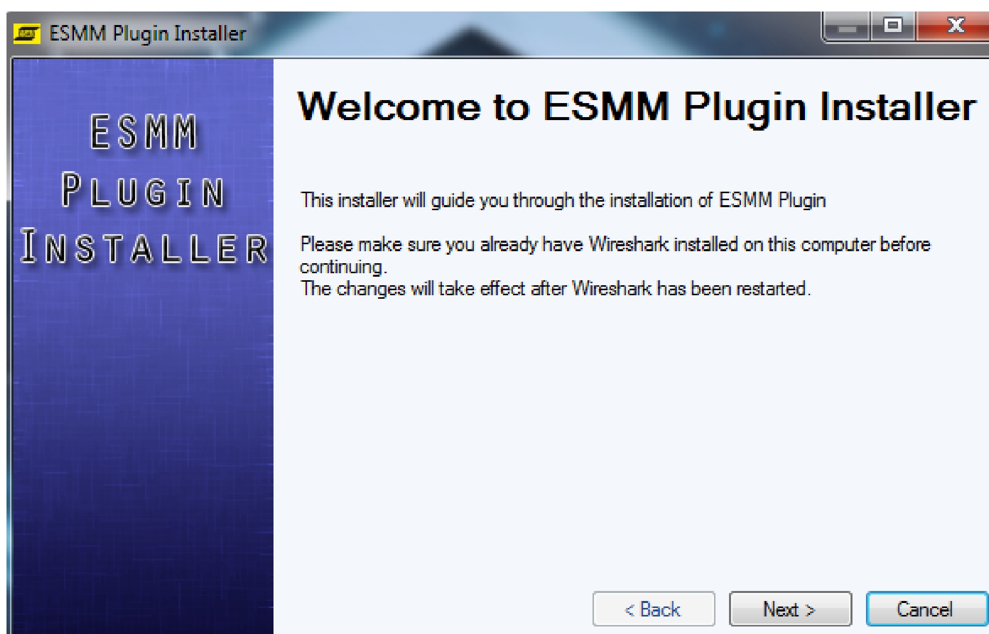


Figur 12. *Ett exempel på hur grafer ser ut i programmet Esab Can Message Analyzer*

4.5 Vecka 4

Denna vecka skulle finnas till för att göra nödvändiga tester och dessa gjordes. Det som var kvar att testa var om CMA kunde verkligen lyssna på CAN-trafik. Det visade sig att CMA kunde fånga CAN-trafik utan problem. Det återstod att testa hur bra den förstod kommunikationen då man inte hade tillgång till svetsutrustning eller simuleringsverktyg.

Veckan ägnades inte enbart till tester. För CMA fanns det redan ett installer-program som skapades enkelt och snabbt med hjälp av Visual Studio. Visual Studio kan generera ett installer-program automatiskt så därför tog det inte lång tid att skapa det för CMA. Det ansågs vara nödvändigt att också skapa ett program som skulle installera plugin-filerna till Wireshark för att förenkla arbetet för den som skulle behöva dem. Programmet ser inte enbart till att installera DLL-filerna till rätt mapp utan ändrar Wiresharks inställningar så att CMAP dissectorn anropas som Link layer protokoll då man läser PCAP-filer skapta av CMAP. Detta program gjordes helt från grunden utan att använda något standard från Visual Studio eller från andra standard sätt att skapa ett installer program såsom NSIS (se källförteckning, artikel nr 14) m.m. Detta pga. att man behövde ändra inställningar i Wireshark.



Figur 13. *Bilden ovan visar hur installern, som utvecklades för enkelt implementering av Wireshark pluginet, ser ut.*

Under veckan lades även till möjlighet att skapa CSV-loggfiler med CMA. På detta sätt kan man spara en logg över trafiken man fångat med CMA.

Fler tester lämnades till den kommande veckan då det förväntades finnas möjlighet till åtkomst till svetsutrustning då.

4.6 Vecka 5

Arbetet tog längre tid än förväntat. En extra vecka. Detta pga. att åtkomst till svetsutrustning under vecka 4 saknades vilket innebar att några tester fick vänta. Testerna som skulle gjorts under Vecka 4 kunde göras under denna vecka, nämligen att testa realtidsavläsning av kommunikationen i svetsutrustningen.

Man testade Wireshark pluginen samt CMA som numera kallas för Esab CAN Message Analyzer och förkortas till ECMA. Det visade sig att ECMA gjorde fel då den försökte förstå CAN-trafik. Detta gällde även för loggar då den delen i programmet som försöker förstå kommunikationen är den samma för loggar som för ”live” trafik. Detta kunde snabbt lösas. Test genomfördes också som testade ifall ECMA kunde skapa pcap-filer av en inläst CAN-trafik, vilket fungerade felfritt. Efter några slutgiltiga tester så kunde ECMA sedan ses som färdigt.
Strax efter att programmet hade setts som färdigt fördes ett samtal med en av Esabs service tekniker. Nya önskemål uppkom om hur ECMA skulle hjälpa mer med att hitta fel. Detta är inget man hade tid med och därför kunde det inte implementeras. Förslaget beskrivs bättre i avsnitt 6.4.1.
Även om tid för den förbättring inte fanns så fanns det tid att få ECMA att kunna visa kommunikationen i realtid, som i bilden till höger. Under vecka 5 skrevs även användarmanualer både för ECMA samt för Wireshark pluginet.

Figur 14. Bilden ovan illustrerar panelen i ECMA som har som uppgift att representera trafik i realtid.

Figur 15. *En överblick över det färdiga programmer Esab Can Message Analyzer*

5. Resultat

Den grundläggande frågeställningen, som kan återfinnas i kapitel 1.2, beskriver det ursprungliga problemet. För att kunna besvara frågeställningen krävdes efterforskning som beskrivs i kapitel 4.1. Resultatet av efterforskning beskrivs därefter i kapitel 2.1.

Efter att ha utfört efterforskningen uppkom en hypotetisk lösning som man, vid den punkten i arbetet, ansåg kunna besvara frågeställningen. Denna hypotetiska lösning blev då grunden för planeringen som lades upp för resten av arbetet, se kapitel 2.2.

För att besvara frågan om hur man kan underlätta granskning av kommunikation mellan komponenterna kan detta formuleras på följande sätt. För Ethernet-kommunikation blev svaret skapande av en dissector (se kapitel 4.2). Dissectorn kan avläsa den trafik som skickas genom EtherNet/IP och sedan tolka datan och representera den i användargränssnittet på ett effektivt sätt. Detta förenklar analys avsevärt då användaren nu kan se kommunikation utan att behöva läsa av rådata och på egen hand försöka klura ut vad den innebär. Ett problem som kvarstår dock är hur man kan göra analysen enkel trots att man har väldigt stora mängder paket varav många är identiska. Hur identifierar man förändring? Svaret på detta blev att använda sig av Wiresharks inbyggda IO Graph funktion (se kapitel 4.3). Genom att använda detta inbyggda verktyg kan verktyg kan en användare med enkelhet fånga kommunikationen och sedan använda grafen för att få en bättre överblick över vad som har hänt under sessionens gång.

När det kommer till CAN-kommunikation blev svaret på problemet istället att utveckla ett eget program som med hjälp av en CAN-to-USB adapter kan läsa av CAN-trafiken och tolka den (se sista stycket i kapitel 4.3 samt kapitel 4.4). Innan var det vanligaste tillvägagångssättet att fånga trafik med en datalogger och sedan skapa en logg av den (se kapitel 3.5.1.1). Loggen kan sedan öppnas i exempelvis Microsoft Excel där man kan iaktta paketens data. Detta är dock ej användarvänligt då datan inte egentligen tolkas, utan förblir hexadecimal rådata. Dock finns möjligheten med ECMA att inte bara öppna loggfiler för att se rådata, utan även att skapa en graf som ger användaren en tydlig överblick över trafiken (se kapitel 4.4). Grafen kan användas då man redan har inläst trafik i programmet, vare sig det kommer från en loggfil eller genom en fångad trafik. Dock ifall man vill analysera kommunikation i realtid så implementerades en panel som visar upp alla flaggor och värden som paketen består av (se kapitel 4.6). Användaren kan sedan med hjälp av denna se hur dem förändras under en inläsning av trafik.

Med både dissectorn och ECMA skapade öppnas det möjligheter för att på ett enklare och effektivare sätt, nämligen att kunna skapa grafer som ger en bra överblick över kommunikationen, kunna analysera både Ethernet- och CAN-kommunikation. Att kunna se kommunikation i grafer gör även att det blir betydligt lättare att granska trafik även i de fall då man har väldigt stora mängder paket att hantera. Med detta gjort ansågs ursprungsfrågan i kapitel 1.2 vara besvarad och uppgiften om att kunna förbättra analys utfört.

6. Slutsats

6.1 Resumé

Uppgiften som den här rapporten fokuserar på går ut på att förenkla diagnostik av datakommunikation mellan komponenter i ett svetssystem. Datakommunikation går antingen över CAN eller över Ethernet. Syftet med uppgiften är att undersöka hur man kan möjliggöra en effektivare granskning av trafiken och på så sätt spara både tid och resurser.

En undersökning utfördes angående möjligheten att skapa en dissector för Wireshark som kan läsa och tolka Ethernet-trafiken. Det undersöktes även om hur en sådan dissector hade kunnat användas vid tolkning av CAN-trafik, vilket hade krävt ett externt program. Därför undersöktes även möjligheten att låta programmet på egen hand kunna tolka och redovisa CAN-trafiken.

De två först nämnda undersökningarna gav goda resultat. Wireshark-dissectorn kunde redovisa kommunikationen mellan svetsutrustningens komponenter (strömkällan och robotkontrollern) samt kunde ge en bra överblick över en session tack vare Wiresharks inbyggda funktionalitet att skapa grafer. Den sista undersökningen visade att programmet som användes för att skapa PCAP-filer från CAN-trafik, vare sig det översattes från loggar eller om trafiken fångades med sagt program, kunde utökas att kunna bli ännu bättre än Wireshark. Detta då det fanns mycket frihet i hur man skulle bygga programmets kod.

Man kom fram till att vid fortsatt utveckling hade en bra idé varit att vidare utveckla programmet som används för att fånga och analysera CAN-trafik. Ifall programmet ges möjlighet att även kunna läsa av Ethernet-trafik så skulle den ersätta dissectorn då programmet redan har bättre funktionalitet. På så sätt hade man dessutom erhållit en enda lösning som fungerar för både CAN- och Ethernet-trafiken.

6.2 Kritisk diskussion

Ett av de största problemområden under arbetets gång har varit att försöka skilja på in- och ut-paket vid skapandet av Wireshark dissectorn. När det kom till CAN-delen var det betydligt lättare. I CAN-meddelanden angavs det id-nummer för varje meddelande, och systemet är uppbyggt så att enheten med lägst id är mest prioriterad och därmed master i systemet. Det var på så sätt enkelt att lista ut vilken komponent som ett meddelande kom ifrån.
Med EtherNet/IP-delen däremot var det krångligare. Det finns inget i minnesmodellen som talar om i vilken riktning ett paket skickas. Eftersom in- och ut-paket har olika struktur är det givetvis viktigt att ta reda på varifrån ett paket kommer innan dissectorn kan börja med att bearbeta informationen. Lösningen var att titta på paketens totala storlek. Förutom minnesmodellen på 16 bytes lades det till 2 respektive 6 bytes på paketen. Dessa ytterligare bytes är med stor sannolikhet information som är specifikt tillagt av utrustningens tillverkare och inget som dissectorn egentligen behandlar då det är skilt från minnesmodellen. Genom att skilja på paket med 18 bytes storlek och dem med 22 bytes storlek kunde dissectorn skilja på in- och ut-paket. Detta är dock en osäker lösning som innebär att metoden för att skilja på paketen kanske inte längre kommer att fungera i framtiden ifall antal bytes som läggs till ändras. För att lösa detta problem krävs det ett mer stabilt sätt att skilja på in- och ut-data på. Någon sådan lösning kunde inte fastställas dock trots efterforskning då det ändå inte finns

mycket information att gå efter i paketen som skickas för att avgöra riktning och avsändare. Man kunde inte heller kontakta utrustningens tillverkare.

Ett annat problem var otillräcklig efterforskning inom EtherNet/IP. Under efterforsknings-fasen lades mycket fokus på Wireshark och nästan ingen fokus på EtherNet/IP. Detta ledde senare till en felberäkning. Det antogs att en dissector skulle räcka för att kunna avlyssna EtherNet/IP trafiken mellan strömkällan och robotkontrollern. Därför skulle det inte finnas några problem med att skapa dissectorn som ett plugin till Wireshark. Det visade sig dock att det krävdes ändring i EtherNet/IPs dissector för att kunna anropa vår dissector. Detta innebär att det inte längre resulterar i ett plugin. Man blev dock tvungen att *låna* EtherNet/IPs dissectors kod för att göra en separat dissector som då skulle kunna göras som plugin.

Trots att viss efterforskning var otillräcklig så resulterade efterforskningen i en planerad lösning som kunde följas nästan helt. Den planeringen låg som grund till hur arbetets gång skulle vara vilket resulterade i ett lyckat arbete.

6.3 Generaliseringar

Arbetet som denna rapport går igenom handlar om hitta en lösning för att enklare diagnostisera datakommunikation mellan olika komponenter i svetssystem. Felsökning är annars väldigt tidskrävande och problematisk. Detta eftersom det inte finns något effektivt verktyg som kan tolka och tyda denna kommunikation som sker i form av tusentals paket per minut. Det finns sätt att läsa av kommunikation dock inte att tyda den. Kommunikationen det här arbetet fokuserar sig på är den kommunikationen som sker mellan två viktiga komponenter i svetssystemet. Båda dessa komponenter skickar mängder av paket mellan varandra som informerar om viktig information så att de kan automatiskt justeras efter varandras behov. Till exempel är det viktigt att reglera ström- och spänning på rätt sätt beroende på robotarmens status.

Kommunikation sker antingen över Ethernet eller över CAN. Därför undersöktes möjligheten att hitta en lösning som täckte båda kommunikationssätt eller eventuellt separata lösningar.

6.4 Fortsatt forskning

6.4.1 Mer än bara en CAN avläsare

Vad innebär detta?

I dagsläget så är Esab CAN Message Analyzer ett verktyg som kan bevaka CAN trafik och förstå den informationen som skickas, om det nu är Esab Memory Model paket som är det som skickas.
Ett nästa steg skulle vara att programmet även kan skicka data för att agera som kontroller samt simuleringsverktyg.

Detta skulle kunna användas för att tvinga fram signaler på ett enkelt sätt och kunna då se om önskat resultat sker. T ex om man manuellt styr robotkontrollen och försöker få robotarmen att börja mata tråd men inget händer, kan man redan med det här programmet se ifall något fel finns hos robotkontrollern som gör att den aldrig skickar den signalen. Men genom att få den att även simulera dessa signaler kan man också bevisa att det inte heller är något fel hos

strömkällan. Om man fortsätter på samma exempel, så skulle man kunna skicka till strömkällan den signalen som skulle skickats från robotkontrollern, alltså ”mata tråd”. Om tråd då matas kan man se att strömkällan fungerar och då vet man att fel finns enbart hos robotkontrollern.

Tillvägagångssätt

I nuläget har ju programmet redan en panel till höger som visar upp alla de tillgängliga flaggorna och värden, där flaggor respresenteras av en ruta som ändrar färg beroende på om den för tillfället är sann eller falsk. Ett förslag hade ju varit att om man klickar på en av dessa rutor så skickar den motsvarande signalen. T.ex. om man trycker på rutan för ”Weld On” så skickas denna signal in till robotarmen.
Detta hade man kunnat åstadkomma genom att sända CAN-meddelanden som till roboten där den önskade flaggan är satt till true. Programmet har redan en färdig klass som heter IxxatHandler som tillhandahåller de funktioner som används vid kommunikation via CAN-to-USB adaptern. Genom att låta den klassen även ha funktioner för att skicka meddelanden (vilket stöds av Ixxat’s API) hade man kunnat använda dessa för att låta programmet skicka CAN-meddelanden.

6.4.2 Esab CAN Message Analyzer ersätter Wireshark

I slutändan så blev Esab CAN Message Analyzer det bästa verktyget för undersökning och visning av datakommunikationen. Möjligheterna som detta program erbjuder är överlägsna de som finns tillgängliga genom att använda Wireshark som huvudsakligt program för att läsa av trafiken. Inte minst på grund av att Esab CAN Message Analyzer är ett eget skapat program som därför kan utökas med fler funktioner och hjälpmedel ifall så önskas, medans det finns vissa restriktioner när man gör en dissector åt Wireshark.

Därför så skulle man helt kunna strunta i översättningen av Can kommunikation till pcap och gå andra vägen: Från pcap till ECMA, som därefter borde byta namn. På så sätt hade en användare kunnat läsa Ethernet-trafik i Wireshark och sedan spara detta som en pcap fil som man sedan öppnar i ECMA. På så sätt hade man sen kunnat använda programmets graffunktion även i en analys av Ethernet-trafik.

Man skulle kunna utöka programmet ännu mer och skippa mellanvägen och få ECMA själv att kunna bevaka Ethernet trafik och fånga EtherNet/IP packetena och dissecta dem ungefär som det redan gör för Can trafik. På så sätt finns det ingen anledning att hellre vilja använda Wireshark.

Referenser

1. ESAB, Aristo $W8_2$ Service Manual
http://www.welding.org.uk/esabmanuals/service%20Manuals/Aristo%20W82.pdf

2. IXXAT, Virtual CAN Interface .NET-API Programmers Manual
http://www.ixxat.com/download/vci_341_net-api-manual_1.5.pdf

3. Texas Instruments, Controller Area Network Introduction
http://www.ti.com/lit/an/sloa101a/sloa101a.pdf

4. Wireshark, Developers Guide
http://www.wireshark.org/docs/wsdg_html_chunked/

5. Zedgraph
http://zedgraph.sourceforge.net

6. Grafic Display
http://www.codeproject.com/Articles/32836/A-simple-C-library-for-graph-plotting

7. C#
http://en.wikipedia.org/wiki/C_Sharp_(programming_language)

8. Kvaser Memorator (Data Logger adapter)
http://www.kvaser.com/en/products/can/data-logger.html

9. IXXAT CAN-to-USB adapter
http://www.ixxat.com/usb-to-can-compact-interface_en.html
http://www.ixxat.com/usb-to-can-ii-interface_en.html

10. EtherNet/IP & CIP Overview
http://www.rtaautomation.com/ethernetip/

11. CIP, CAN & DeviceNet Overview
http://www.rtaautomation.com/devicenet/

12. OSI Model
http://compnetworking.about.com/cs/designosimodel/g/bldef_osi.htm

13. Libpcap/pcap
http://wiki.wireshark.org/Development/LibpcapFileFormat

14. NSIS
http://nsis.sourceforge.net/

Appendix

- ESMM Plugin Användarmanual
- Esab CAN Message Analyzer Användarmanual
- Planering – Detaljerad sammanfattning av den ursprungliga planeringen

ESMM Plugin

Användarmanual

ALESANDRO SANCHEZ

MARTIN SONESSON

Innehållsförteckning

1. Krav 3

2. Installern 3

3. Manuell installation 4

4. Användning 5

4.1 Läsa av trafik 5

4.2 Filtrera 6

4.3 Visa graf 8

1. Krav

För att kunna använda ESMM pluginet till för att kunna läsa av EtherNet/IP-trafik och tolka minnesmodellen med Wireshark krävs det att följande krav uppnås:

- En installerad version av Wireshark (version 1.6.5 eller senare)
- Något av följande operativsystem:
  - Windows 7
  - Windows 7 Service Pack 1
  - Windows Server 2003 Service Pack 2
  - Windows Server 2008
  - Windows Server 2008 R2
  - Windows Server 2008 R2 SP1
  - Windows Vista Service Pack 1
  - Windows XP Service Pack 3
- .NET Framework 4 (Finns att hämta på http://www.microsoft.com/en-us/download/details.aspx?id=17851)

2. Installern

För att på ett enkelt sätt kunna installera pluginet och möjliggöra användning genom Wireshark på enklaste möjliga sätt kan du använda den medföljande installern.

Steg 1

Starta installern genom att dubbelklicka på ”ESMM Plugin Installer.exe”. En ruta bör komma upp som frågar om tillåtelse att få köra den som administratör, tryck ja.

Steg 2

Tryck på ”Next”.

Steg 3

Läs igenom licensavtalet och tryck på ”I Agree” om du går med på villkoren (licensavtalet i fråga är GNU General Public License).

Steg 4

Installern kommer nu automatiskt att försöka hitta din Wireshark-mapp och sökvägen till den kommer i så fall att stå i textfältet. Om det inte lyckades så tryck på ”Browse” och välj din Wireshark-mapp där programmet är installerat och tryck sedan ”OK”. Tryck därefter ”Install”.

Steg 5

Såvida inget fel inträffade fram tills detta steget är installationen nu klar och ESMM Plugin är redo att användas.

Ifall installationen av någon anledning inte lyckades kan du testa att göra en manuell installation (se kapitel 3).

3. Manuell installation

För att installera ESMM plugin manuellt följ bara de nästkommande stegen.

Steg 1

Lokalisera mappen C:\Program Files\Wireshark\plugins\1.6.7

Sökvägen beror förstås på vart du har installerat programmet samt att namnet på mappen 1.6.7 beror på vilken version av Wireshark som är installerat. Till denna mapp ska du klistra in de tre medföljande filerna ”enip2.dll”, ”cmap.dll” och ”esmm.dll”.

Steg 2

Starta Wireshark.

Steg 3

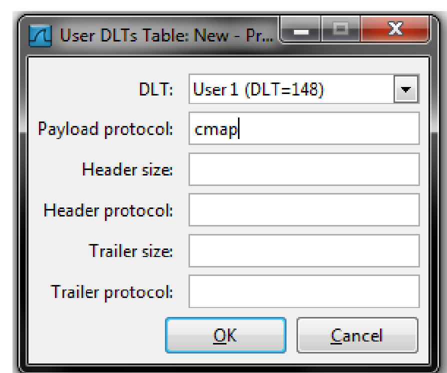
Gå till Edit > Preferences.

Steg 4

Under ”Protocols” leta fram och tryck på fliken som heter ”DLT_USER”. Tryck sedan på ”Edit…”

Steg 5

Tryck på ”New”. Fyll i rutan som kommer enligt bilden till höger. Tryck sedan på OK.



Steg 6

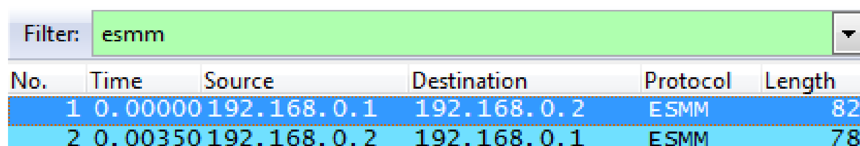
Den manuella installationen är nu klar!

4. Användning

4.1 Läsa av trafik

För att sedan i realtid kunna läsa av EtherNet/IP trafik från ett svetssystem krävs det att du kopplar in datorn till robotkontrollerns kort via Ethernet porten. Starta sedan Wireshark och starta en ”capture” genom att klicka Capture > Interfaces. I rutan som kommer upp kryssar du i det nätverk som du vill läsa trafik ifrån. Det bör ha samma namn som ditt nätverkskort. Tryck sedan OK.

Wireshark läser nu av paket som skickas genom nätverket. Ifall installationen av pluginet är korrekt kommer paket som går över EtherNet/IP-protokollet och som är uppbyggt enligt minnesmodellen att stå som ”ESMM” under kolumnen ”Protocol”. Ifall du enbart vill se dessa paket skriv in ”ESMM” i filter-textfältet och tryck sedan Enter, enligt bilden nedan.



Under info-kolumnen ser du lite generell information om paketen. Till exempel Dir, som står för direction (riktining), som informerar om paketet är antingen på väg in till svetsroboten från kontrollern eller tvärtom; ut från svetsroboten till kontrollern.

För att kunna se mer detaljerad information om ett paket så är det bara att klicka på det. I rutan under i Wiresharks användargränssnitt går det att se information om paketet som man valt (se bild till höger). För att se information relaterat till minnesmodellen, tryck på plusset vid ”ESAB Memory Model Protocol”. Där under bör du nu se alla värden och flaggor som minnesmodellen innehåller. Notera att vissa av värdena såsom spänning, ström och trådmatarhastighet mäts hexadecimalt, vilket är indikerat av ”0x” före värdet.

```
⊟ ESAB Memory Model Protocol
    .... ...0 = Weld Busy: False
    .... ..0. = Arc Acknowledge: False
    .... .0.. = Touch Sense Response: False
    .... 0... = Remote Active: False
    ...1 .... = Wire Feed number 1 active: True
    ..0. .... = Wire Feed number 2 active: False
    .0.. .... = Wire Feed number 3 active: False
    0... .... = Wire Feed number 4 active: False
    .... ...0 = Error Type 1: False
    .... ..0. = Error Type 2: False
    .... .0.. = Error Type 3: False
    .... 0... = Error Type 4: False
    ...0 .... = Error Type 5: False
    ..0. .... = Error Type 6: False
    .0.. .... = Error Type 7: False
    0... .... = Error Type 8: False
    .... ...0 = Error in Weld Data Unit: False
    .... ..0. = Error in Power Source Unit: False
    .... .0.. = Error in Wire Feed Unit: False
    ...0 .... = Gas Active: False
    ..0. .... = Collision Detection: False
    .1.. .... = Synergy Active: True
    0... .... = Inching Active: False
    Voltage (Measured): 0x0000
```

4.2 Filtrera

Man kan även filtrera mer exakt i Wireshark, till exempel ifall man är ute efter specifika värden. Nedan följer en tabell över de värden som går att filtrera efter.

1. **IN-paket**
   - 1.1. Byte 0
     - 1.1.1. esmm.in.weld_on — Boolean
     - 1.1.2. esmm.in.quick_stop — Boolean
     - 1.1.3. esmm.in.emerg_stop — Boolean
     - 1.1.4. esmm.in.inching — Boolean
     - 1.1.5. esmm.in.gas_purge — Boolean
     - 1.1.6. esmm.in.clean — Boolean
     - 1.1.7. esmm.in.inc_reverse — Boolean
     - 1.1.8. esmm.in.remote_active — Boolean
   - 1.2. Byte 1
     - 1.2.1. esmm.in.wire_feed_one — Boolean
     - 1.2.2. esmm.in.wire_feed_two — Boolean
     - 1.2.3. esmm.in.wire_feed_three — Boolean
     - 1.2.4. esmm.in.wire_feed_four — Boolean
     - 1.2.5. esmm.in.analog_active — Boolean
     - 1.2.6. esmm.in.release_wire — Boolean
     - 1.2.7. esmm.in.touch_sense — Boolean
   - 1.3. Byte 2
     - 1.3.1. esmm.in.weld_data_number — Värde
   - 1.4. Byte 8-9
     - 1.4.1. esmm.in.voltage — Värde
   - 1.5. Byte 10-11
     - 1.5.1. esmm.in.wire_feed_speed — Värde

2. **UT-paket**
   - 2.1. Byte 0
     - 2.1.1. esmm.out.weld_busy — Boolean
     - 2.1.2. esmm.out.arc_ackn — Boolean
     - 2.1.3. esmm.out.touch_sense_resp — Boolean
     - 2.1.4. esmm.out.remote_active — Boolean
     - 2.1.5. esmm.out.wire_feed_one_active — Boolean
     - 2.1.6. esmm.out.wire_feed_two_active — Boolean
     - 2.1.7. esmm.out.wire_feed_three_active — Boolean
     - 2.1.8. esmm.out.wire_feed_four_active — Boolean
   - 2.2. Byte 1
     - 2.2.1. esmm.out.error_type_one — Boolean
     - 2.2.2. esmm.out.error_type_two — Boolean

2.2.3. esmm.out.error_type_three Boolean
2.2.4. esmm.out.error_type_four Boolean
2.2.5. esmm.out.error_type_five Boolean
2.2.6. esmm.out.error_type_six Boolean
2.2.7. esmm.out.error_type_seven Boolean
2.2.8. esmm.out.error_type_eight Boolean

2.3. Byte 2
2.3.1. esmm.out.error_wdu Boolean
2.3.2. esmm.out.error_ps Boolean
2.3.3. esmm.out.error_wfu Boolean
2.3.4. esmm.out.gas_active Boolean
2.3.5. esmm.out.collision_detec Boolean
2.3.6. esmm.out.synergy_active Boolean
2.3.7. esmm.out.inching_active Boolean

2.4. Byte 3-4
2.4.1. esmm.out.voltage_measured Värde

2.5. Byte 5-6
2.5.1. esmm.out.current Värde

2.6. Byte 7-8
2.6.1. esmm.out.power Värde

2.7. Byte 9
2.7.1. esmm.out.weld_data_number Värde

2.8. Byte 10-11
2.8.1. esmm.out.voltage Värde

2.9. Byte 12-13
2.9.1. esmm.out.wire_feed_speed Värde

2.10. Byte 14-15
2.10.1 esmm.out.synergic_voltage Värde

Dessa är värdena och flaggorna som ingår i minnesmodellen och det är dessa som du kan filtrera med i Wireshark. Används filter-textfältet i Wireshark. För att se till exempel enbart se paket där flaggan ”Weld on” existerar skriv följande i textfältet: *esmm.in.weld_on*

(Notera att du måste trycka på Enter efter att ha angett ett filter för att den ska börja gälla)

Detta kommer förstås bara att visa in-paket eftersom flaggan inte existerar i ut-paketen. Notera dock att detta kommer att göra så att ALLA in-paket visas, eftersom alla in-paket har ”Weld on”-flaggan, vare sig den är sann eller falsk. För att bara se paket flaggan är på (alltså sann) skriv följande i filter-textfältet: *esmm.in.weld_on == true*

Wireshark kommer nu enbart att visa in-paket där ”Weld on”-flaggan är sann. Du kan även skriva ”false” istället för ”true” för motsatt resultat. Även 1 eller 0 kan användas istället för true eller false. Notera att detta endast går på flaggor eftersom de är booleaner, värden som bara kan anta sant eller falskt. För värden såsom t.ex. esmm.out.current kan du inte filtrera efter sant eller falskt. Som med flaggor kan du även skriva enbart ”esmm.out.current” i filter-textfältet för att se alla paket som innehåller detta värde. Du kan också filtrera genom att se t.ex. när värdet är 0. Detta genom att skriva följande: *esmm.out.current == 0*

Följande operatorer kan användas:

== Lika med

!= Skilt från (ej lika med)

> Större än

>= Större eller lika med

< Mindre än

<= Mindre än eller lika med

Du kan även filtrera genom att kombinera flera påståenden genom att använda följande operatorer:

|| Eller

&& Och

Till exempel kan du skriva: *esmm.in.weld_on == true && esmm.in.inching == true*

Detta kommer att bara visa in-paket där både “Weld on”- och “Inching”-flaggan är sanna.

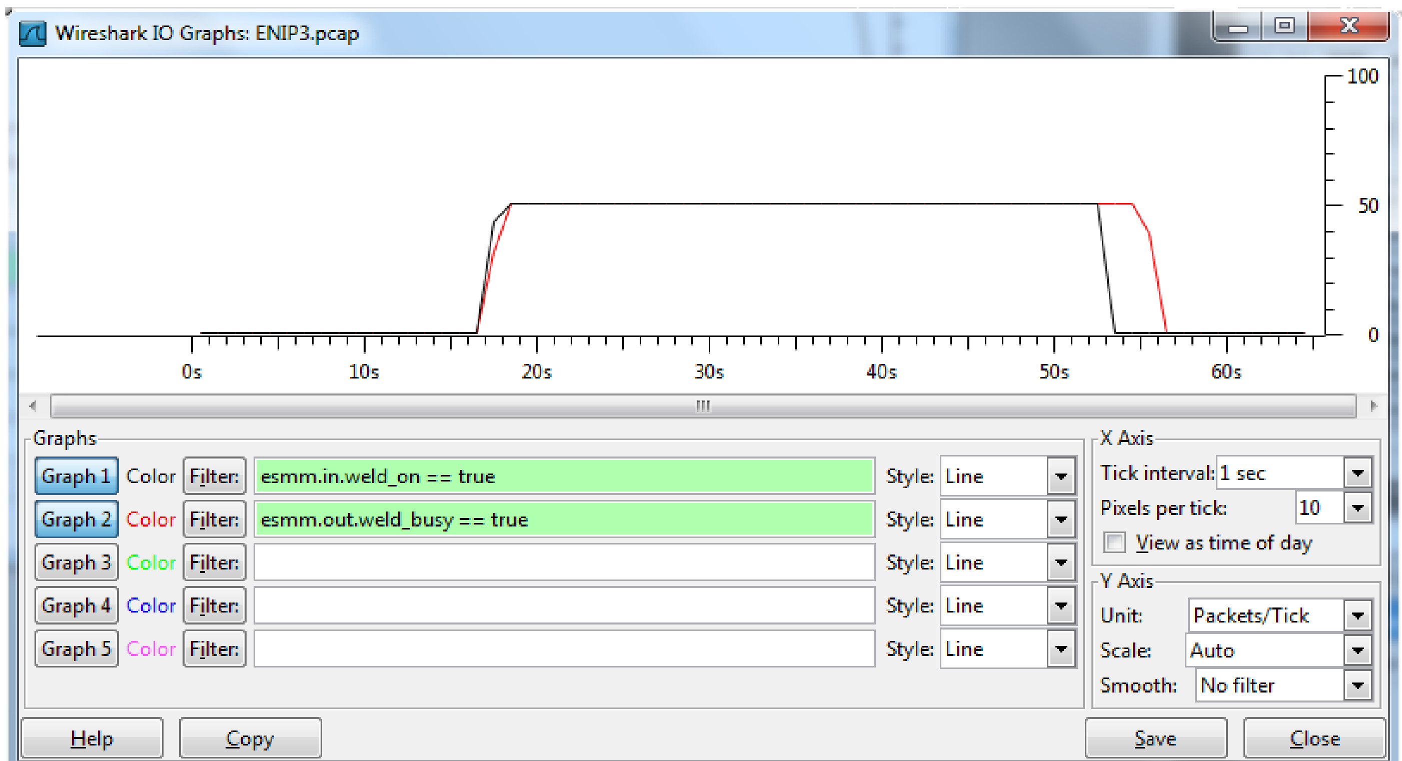
4.3 Visa graf

Ibland kan det vara fördelaktigt att kunna se en graf över en hel session. Speciellt om man fångat paket från en lång session så finns det mycket information att behandla. För detta är det väldigt användbart att använda sig av Wiresharks inbyggda graf-funktion.

Gå till Statistics > IO Graph

För att se en graf måste du definiera vilka flaggor du vill visa. Notera att Wiresharks IO grafer visar upp antal paket av den definierade typen genom en viss tidsenhet som värde på y-axeln. Detta gör att t.ex. flaggor kommer inte att växla med skarpa vändningar på grafen mellan sant eller falskt, utan den kommer snarare att sjunka då sessionen går in i ett intervall då flaggan inte längre är sann. Detta innebär också att Wiresharks IO grafer är bäst för att se just flaggor (booleaner) men inte så mycket för värden såsom voltage, current etc.

För att definiera en flagga som du vill se i grafen måste du ange ett filter. Detta fungerar på samma sätt som beskrivit i föregående kapitel.



Man kan visa 5 olika värden samtidigt. Skriv enbart in ett filter i en av textfälten och tryck på Enter för att filtret ska börja fälla. Notera att graf-knappen (Graph 1, Graph 2 etc) måste vara markerad vid filtret du angivit för att den ska synas på grafen.

Vid **Style** kan du ändra utseendet linjen i grafen. Det finns Line, Impulse, FBar och Dot. Ifall man visar flera flaggor samtidigt är det ibland bra att ha olika styles för att göra det enklare att skilja på dem.

Under **X Axis** ställer du in grafens storlek i sidled. För att få en graf som är lätt att överskåda så gäller det att välja bra värden.

Bilden ovan är ett exempel på hur grafen kan användas. Den har ställts in till att visa både in-paket med flaggan ”Weld on” som sann samt ut-paket med flaggan ”Weld busy” som sann.

Esab Can Message Analyzer

Användarmanual

ALESANDRO SANCHEZ

MARTIN SONESSON

Innehåll

1.Krav 3

2. Överblick..... 3

3. Inläsning av CAN-loggar..... 4

4. Läsning av CAN-Kommunikation..... 4

4.1 Quick CAN capture 4

4.2 Manual capture..... 5

4.3 Välj rätt Baudrate 6

4.4 Realtidsläsning 6

5. Spara filer 7

5.1 Spara som libpcap (*.pcap) fil..... 7

5.2 Spara som CSV 7

6. Rita grafer..... 8

1. Krav

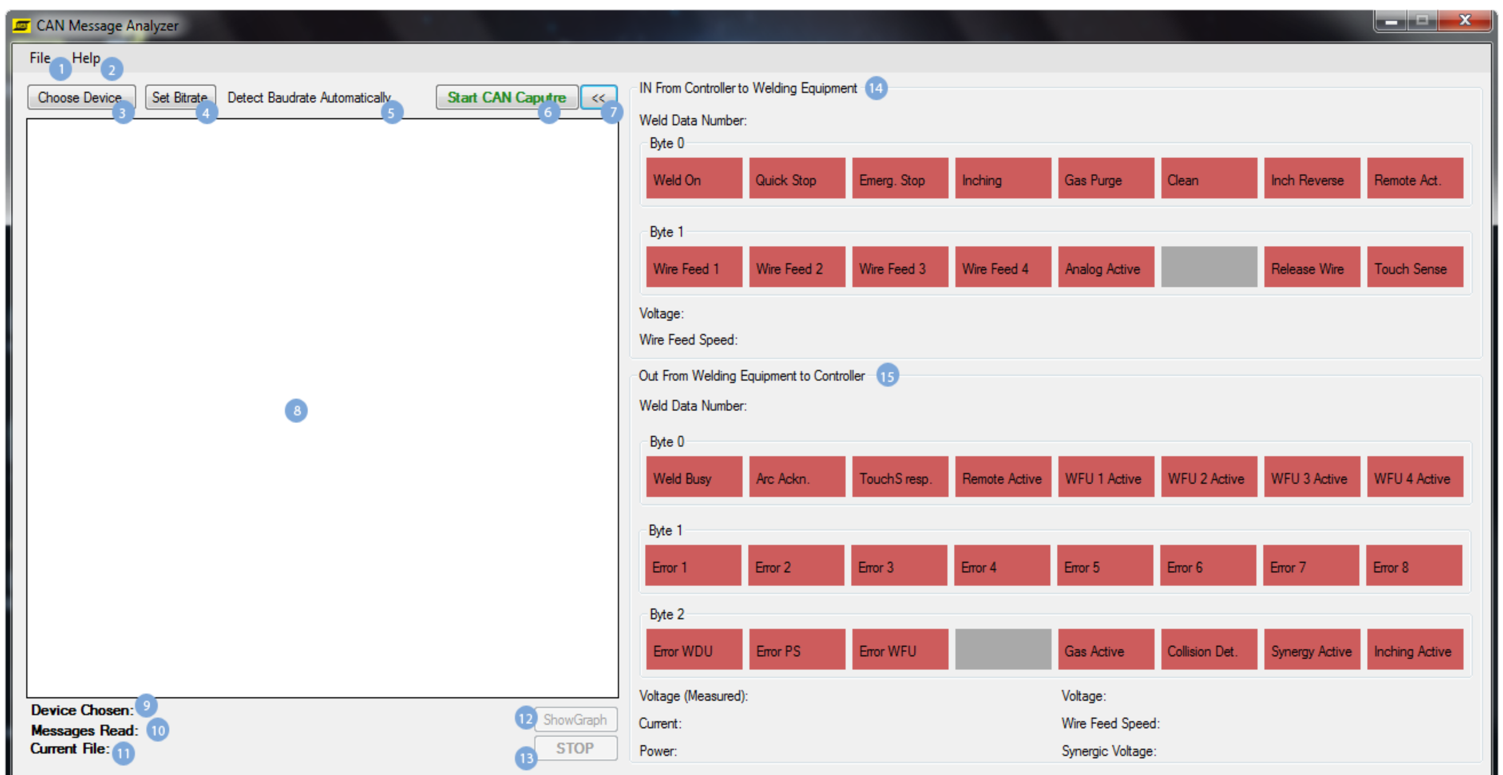
Följande krav behöver tillgodoses för att kunna använda Esab Can Message Analyzer.

- IXXAT VCI 3.5 Driver

Esab CAN Message Analyzer använder IXXATs drivrutiner för att kunna läsa av CAN trafiken som går igenom IXXATs CAN-to-USB adaptrar. Kontrollera att drivrutinen finns installerad på din dator.

Det är inget måste för att kunna använda programmets andra funktioner, det är bara då man vill ha inläsning av CAN meddelanden med hjälp av CAN-to-USB adaptrar som det krävs (Se avsnitt 3).

2. Överblick

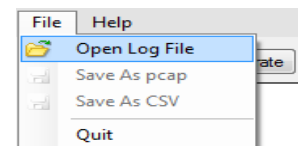


1. **File-fliken.** Härifrån hittar man funktioner för att t.ex. spara och öppna filer.
2. **Help-fliken.** För att kunna se information om programmet.
3. **Choose Device.** Härifrån väljer man vilken CAN-to-USB adapter man vill använda.
4. **Set Bitrate.** Här väljer man vilken bitrate man vill använda under en sökning
5. **Curent Bitrate.** Här anges den valda bitraten. Som standard används ”Detect Baudrate Automatically” vilket innebär att programmet själv försöker hitta vilken bitrate som bör användas.
6. **Start CAN Capture.** Med denna knapp börjar programmet att fånga CAN-trafik.

7. **Hide/Show realtime data.** Med denna knapp visar eller döljer man högra delen av programmet som visar realtidsdata.
8. **Captured Data.** Här visas en lista över denna trafik som har fångats. Notera att man måste ha stoppat sessionen innan datan visas här.
9. **Device Chosen.** Här visas den valda CAN-to-USB enheten.
10. **Messages Read.** Här anges antalet CAN meddelanden som har lästs in efter en avslutad session.
11. **Current File.** Här visas den valda filen som är öppen.
12. **Draw Graph.** Härifrån kan man rita grafer som speglar den inlästa trafiken, vare sig det är från en loggfil eller efter en session inläsning av CAN-meddelanden.
13. **STOP.** Här stoppar men en pågående inläsning.
14. **IN from Controller to Welding Equipment data.** Här syns data som utgör in-paket (från controller till robotarm) i realtid under en inläsning av CAN-trafik. De röda rutorna representerar flaggor i minnesmodellen. Är de röda innebär det att de är ej satta, dvs false. Är de gröna är de true.
15. **OUT from Welding Equipment to Controller data.** Samma som 14 fast här visas istället ut-paket, från robotarm till controller.

3. Inläsning av CAN-loggar

Esab CAN Message Analyzer kan öppna loggar skapta av ”Kvaser Memorator” eller andra program som skapar filer med samma format. Filformatet ska vara csv.



Under fliken ”File” finns en undermeny kallad ”Open Log File”. Använd den här knappen för att sedan välja vilken fil som skall öppnas.
Programmet kommer då att försöka läsa filen och om inläsningen lyckas så kommer då alla CAN meddelanden att visas på fönstret. Programmet är nu dessutom redo att visa grafer baserade på de inlästa CAN meddelanden om dessa är strukturerade enligt den använda minnesmodellen. Vidare förklaring om hur man visar grafer ges i kapitel 5.

4. Läsning av CAN-Kommunikation

Esab CAN Message Analyzer stödjer inläsning av CAN meddelanden. Detta med hjälp av CAN-to-USB adaptrar av märket IXXAT. Drivrutinerna måste vara installerade för att detta skall fungera (se kapitel 1).

4.1 Quick CAN Capture

Att kunna snabbt och enkelt kunna fånga CAN trafiken tillsammans med en IXXAT adapter kan åstadkommas med bara en knapptryckning, nämligen att trycka på ”Start CAN Capture”.

För att inläsningen av CAN meddelanden ska fungera felfritt så krävs det att enbart en CAN-to-USB adaptern är kopplad. I fallet då adaptern har fler än en kanal, så skall CAN 1 vara kopplad.

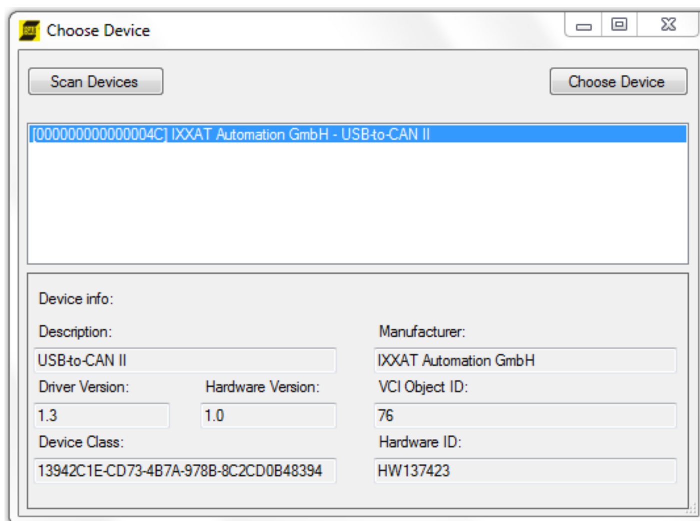
Om de här kraven inte uppfylls så kommer ”Start CAN Capture” knappen inte att starta en ”Quick capture” och kommer inte heller att visa några felmeddelanden

Om kraven är uppfyllda så kommer programmet då att försöka börja inläsningen av meddelanden. Om inte tidigare valt av användaren, så kommer programmet att försöka hitta själv rätt baudrate. Om detta misslyckas så kommer den att be användaren mata in rätt baudrate eller bittiming värden. Ifall du inte vill köra en Quick Can Capture utan vill specifiera mer exakt så startar man en Manual Capture.

4.2 Manual capture

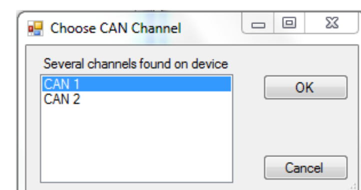
I de fall då man har kopplat fler än en adapter eller då man måste koppla adaptern mot en annan kanal än CAN 1, så följer man den här proceduren. I de fallen då man kopplat en enda adapter med bara en kanal, eller då man kan koppla CAN 1, så rekommenderas Quick CAN capture istället.

Det första steget är att klicka på ”Choose Device” knappen. Ett nytt fönster öppnas innehållandes en lista av de olika enheterna som hittats kopplade mot datorn. Fönstret visar bland annat enhetens Hardware ID, vilket kan också hittas på enhetens baksida. Ifall du behöver uppdatera listan, till exempel ifall du kopplar in en ny adapter, tryck på knappen ”Scan Devices”.



Välj en genom att se till att den är vald och sedan klicka på ”Choose Device” knappen i detta fönster (se bild till vänster), eller genom att dubbelklicka på en av dem. Genom att markera en enhet kan du dessutom se information om enheten under, såsom drivrutinversion, hårdvaru-ID etc.

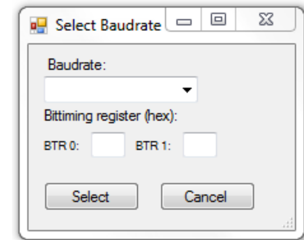
Ifall adaptern du använder har fler än en kanal kommer du bli ombedd att välja vilken som används, se figur till höger.



När detta är gjort kan man nu starta inläsningen genom att klicka på ”Start CAN Capture” knappen.

4.3 Välj rätt Baudrate

I de flesta fall så skall detta inte behövas. Programmet kan hitta rätt baudrate för de vanligaste systemen. Men då det är möjligt att programmet inte hittar rätt baudrate så kan man sätta det genom att klicka på ”Set bitrate” knappen. Om man väljer ”automatiskt” från listan så kommer Esab CAN Message Analyzer att försöka hitta rätt baudrate själv. Om man inte hittar rätt baudrate från listan så kan man mata in själv baudrate genom att skriva in rätt bittiming värden och sedan klicka på ”Select”.



Om man har valt att låta programmet hitta rätt baudrate själv så finns det risk att det inte lyckas med det. I dessa fall kommer programmet att be användaren mata in själv rätt bittiming värden innan inläsningen startar.

4.4 Realtidsläsning

Ifall man vill kunna fånga CAN-trafik och samtidigt övervaka kommunikationen i realtid finns det en möjlighet för detta. Tryck på den lilla knappen till höger om Start CAN Capture knappen för att visa eller dölja den högra panelen som används för att läsa av trafik i realtid.

Nu syns panelen till höger för realtidsanalys. Den övre delen är för in-paket, från kontroller till svetsarm. Den undre är för ut-paket, från svetsarm till kontroller.

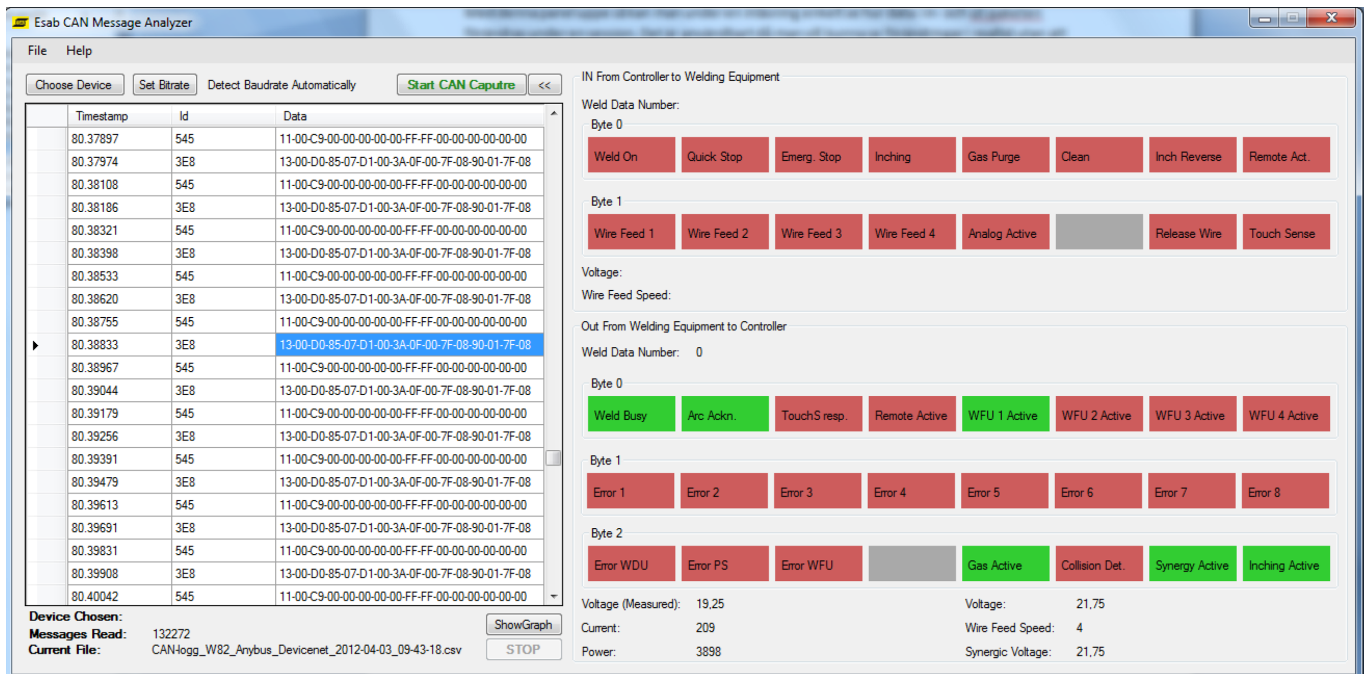
De röda rutorna representerar flaggor i minnesmodellen. När en sådan ruta är röd innebär det att flaggan är ej satt, dvs false. När rutan är grön däremot betyder det true.

Förutom flaggorna står även värden för både in- och ut-paket såsom Voltage och Wire Feed Speed. Dessa värden visas också upp under en inläsning.

Med denna panel uppe så kan man under en inläsning enkelt se hur data i in- och ut-paketen förändras under en session. Det är användbart då man vill kunna se förändringar i realtid utan att behöva använda sig av graffunktionen. För att spara prestanda och inte låta programmet ta upp för mycket kraft så ändras inte denna panelen för varje paket den läser. Med tanke på den höga frekvensen som paketen skickas i hade detta tagit för mycket datorkraft. Istället uppdateras denna panel ungefär två gånger per sekund.

5. Paket dissekering

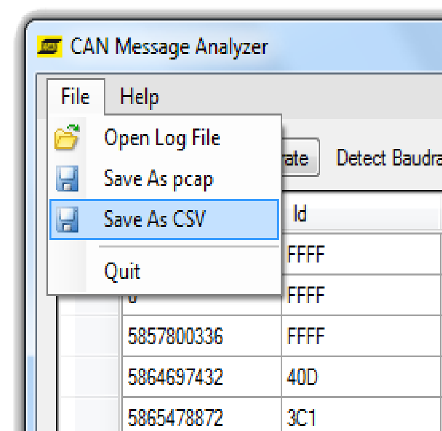
När man har fångat trafik, antingen genom att ha läst trafik genom en CAN-to-USB adapter eller genom att ha öppnat en CAN-logg, kan man se varje individuellt pakets innehåll. Detta gör man genom att i listan för infångade paket (se punkt 8 i kapitel 2) klicka på ett paket. Paketets innehål ritas sedan upp i panelen för realtidsdata. På så sätt är det möjligt att kunna se datan för varje enskilt paket.



I bilden ser man ett exempel på hur man klickat på ett paket i listan och sedan kan inspektera dess flaggor och värden till höger. I detta fall är det ett UT-paket.

6. Spara filer

Esab CAN Message Analyzer stödjer möjligheten att spara loggar. Loggarna kan sparas i CSV format och som PCAP format, det senare används med Wireshark. För att kunna spara i dessa format krävs det först att man har läst in antingen en loggfil först (Se avsnitt 2) eller att man har läst CAN trafik (Se avsnitt 3).



6.1 Spara som libpcap (*.pcap) fil

Under fliken "File" finns en knapp som heter "Save as pcap". Välj den och sedan var du vill spara filen. Filen kan nu öppnas och tolkas av Wireshark.

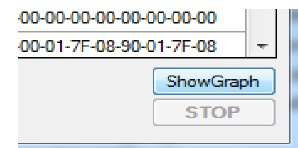
6.2 Spara som CSV

Under fliken "File" finns en knapp som heter "Save as csv". Välj

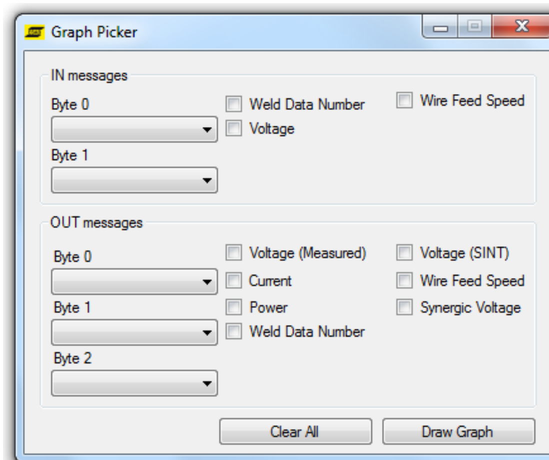
den och sedan var du vill spara filen. En logg-fil av formatet csv kan sedan återigen öppnas av Esab Can Message Analyzer ifall du vid ett senare tillfälle vill analysera den.

7. Rita grafer

Efter en inläsning av CAN-meddelanden eller öppning av en loggfil så ser man paketen i huvudfönstret och man kan också se dess innehåll. Dock säger detta inte användaren mycket och det blir problematiskt att försöka göra en analys härifrån. För närmare granskning kan du använda den inbyggda graffunktionen.

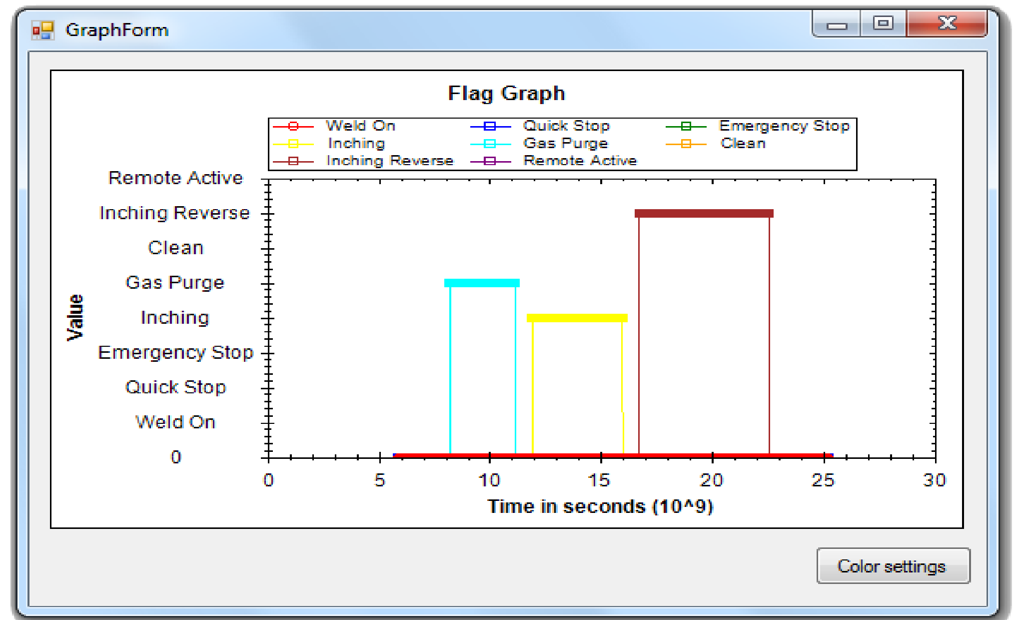


När du har inläst data kan du trycka på ”Show Graph” knappen (se bild till höger). En ny ruta öppnas nu som ber dig att specificera vilka värden och flaggor som du vill se i grafen (se bild nedan). Notera att man kan inte visa både flaggor (såsom weld on) tillsammans med värden (såsom voltage) i samma graf. Ifall du markerar både flaggor och värden kommer två grafer där den ena visar flaggor och den andra värden. När du valt vad du vill visa så trycker du på ”Draw Graph”.



OBS: Tänk på att det kan vara prestandakrävande att visa väldigt mycket värden i samma graf. Även under en kort session fångas stora mängder paket och att behandla många värden i samma graf kan ta mycket kraft och göra programmet långsammare. Det är alltså rekommenderat att man bara väljer i huvudsak de värdena som man anser är viktiga att kolla på och inte alla på en gång.

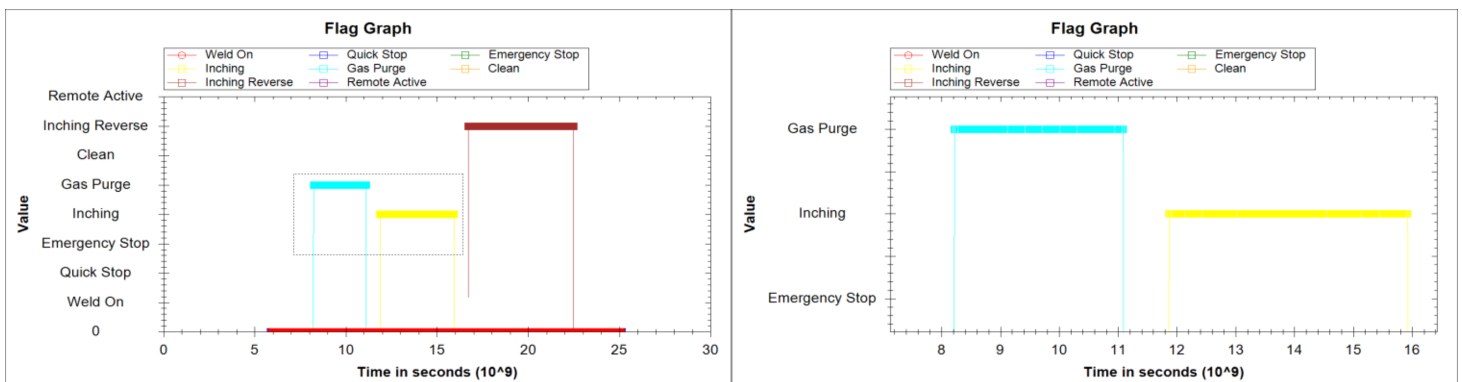
I bilden till höger visar grafen alla flaggor som finns i Byte 0 (för IN-paket).



Efter att ha valt vilka värden man vill visa och tryckt på ”Draw Graph” kommer graffönstret att visas (se bild ovan). Kom ihåg att man kan maximera fönstret för att se grafen noggrannare. Flaggor visas, som i bilden ovan, antingen som true eller false. Ifall en flagga är false kommer den alltid att ligga på 0 på y-axeln. Flaggor ligger på olika höjd på y-axeln för att göra det enklare att se vilka som är true samtidigt (så att dem inte ligger ovanpå varandra). X-axeln räknar på tiden i sekunder.

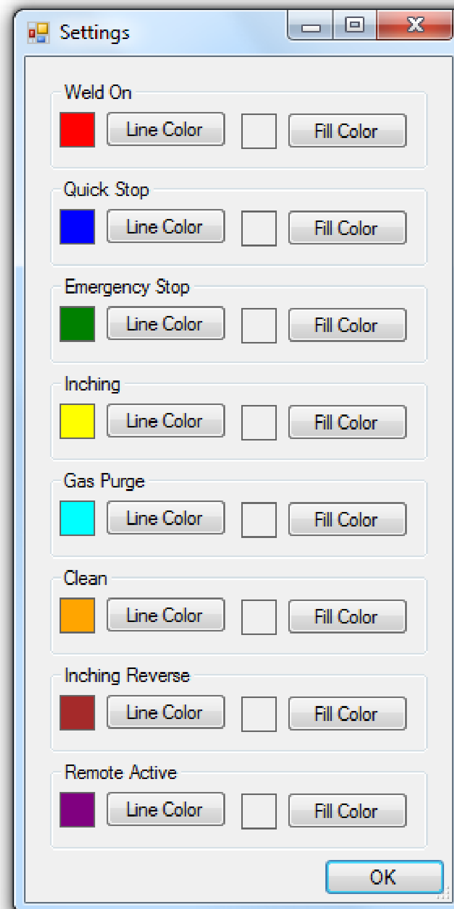
Genom att klicka och dra en ruta på en utvald del av grafen kommer grafen att zoomas in och fokusera på det valda området. På detta sätt går det att gå in närmare på en viss del och få en noggrannare inblick. För att ångra alla inzoomningar och återgå till början, högerklicka på grafen och välj ”Undo All Zoom/Pan”.

Då man har en graf som visar värden (t.ex. voltage och current) som alltså inte är flaggor som bara kan vara true och false går det även att se exakta värden vid vissa punkter. Håll bara muspekaren över en viss punkt på linjen så ser man koordinaten för den punkten.



Genom att dubbelklicka på en viss punkt i grafen kommer paketet för den punkten att läsas och representeras i högra panelen för realtidsdata i programmet. Det använder sig av samma funktionalitet för individuell paketdissekering som förklaras i kapitel 5.

Ifall man av någon anledning vill ändra färgerna på linjerna så finns även denna möjligheten. Tryck på ”Color settings” så kommer en ny ruta upp (se bild nedan). Där kan du ändra färger på linjer samt ”Fill colors”, en ifyllnadsfärg som syns under en funktions linje.



Planering

- Detaljerad sammanfattning av den ursprungliga planeringen

ALESANDRO SANCHEZ

MARTIN SONESSON

Innehåll

1. Överblick 3

2. Vecka 1 - Wireshark och Ethernet 4

3. Vecka 2 - CAN 4

4. Vecka 3 - CAN 4

5. Vecka 4 6

6. Resterande tid 6

1. Överblick

Den här skriften som är en bilaga till slutrapporten av examensarbetet med namnet ”Diagnostik verktyg för tolkning av kommunikation”. Denna bilaga kommer att redovisa om planeringen över arbetet som bestämdes och är mer detaljerad än planeringen i slutrapporten på kapitel 2.2.3.

	v 12	v 13	v 14	v 15	v 16	v 17	v 18	v 19	v 20	v 21	v 22
Planering/Efterforskning	■	■									
EtherNet/IP			■	■							
CAN				■	■						
Testning/Finslipningar						■					
Rapport						■	■	■	■	■	■
Slutredovisning								■	■	■	■

Bilden ovan sammanfattar löst den ursprungliga planeringen som bestämdes för hela examensarbetet.

2. Vecka 1 - Wireshark och Ethernet

Det första arbetsmomentet var att lösa Ethernet-delen. För att enkelt kunna läsa av och analysera Ethernet-trafik som sänds över EtherNet/IP-protokollet bestämdes det att skapa en Wireshark-dissector.

För att åstadkomma detta behövs arbetet utföras successivt i följande moment:

1. Göra så att man kan lyssna på EtherNet/IP (dess portar)
2. Beskriva delarna i paketet i dissectorn
3. Tillämpa ett sätt att skilja på IN och UT paket
4. Justera info-kolumnen så att den visar relevant data.
5. Tillämpa ett sätt att enkelt kunna identifiera när en förändring har skett. (Då väldigt många paket skickas succesivt som innehåller samma data).

Efter att dissectorn är färdig skall man utföra tester för att konstatera att den fungerar.

3. Vecka 2 - CAN

1. Skapa ett program i C# med ett användargränssnitt till en början
2. Tillämpa möjligheten i programmet att kunna öppna och förstå loggar av formatet csv, samt spara som csv.
3. Tillämpa möjligheten att spara inläst information som pcap-format
4. Skapa en dissector i Wireshark som förstår sig på paketen som kommer att återfinnas i pcap-filerna som CAN-programmet sparar. CAN dissectorn ska i sin tur använda sig av EtherNet/IP dissectorn (som skapades i arbetet som beskrivs i kapitel 2) för dissection av paketdelarna
5. På något sätt tillämpa IXXATs kod* för kommunikation med CAN-to-USB adaptern för använding i C# programmet. Eventuellt måste koden anpassas och eller konverteras till ett format som kan importeras i ett C# projekt då IXXAT koden är skriven i C

*: Under efterforskningstiden fick man tag på exempelkod skriven i C++ som redan hade en del funktioner såsom att fånga CAN-trafik.

4. Vecka 3 - CAN

1. Göra så att programmet kan visa vilka adaptrar som är inkopplade samt skapa en anslutning med en adapter
2. Ge programmet möjlighet att kunna lyssna på CAN trafik och fånga meddelanden

Dessa punkter ingick i originalplanen för CAN-delen. Dock fanns tidiga planer om eventuella tillägg för att göra programmet mer komplett och användbart. Detta planerades att utföras i mån av tid, dvs. om tid fanns över. Dessa är följande:

1. Tillämpa möjlighet att låta programmet bättre kunna visa upp datan minst lika detaljerat som Wireshark
2. Lägga till mer funktionalitet till programmet, vad för sorts funktionalitet kommer man fram till under arbetets gång beroende på vilka nya problem som kommer upp samt nya önskemål.

Tanken med att implementera utökad funktionalitet till programmet var att göra det så självstående som möjligt för att på så sätt göra det till ett komplett program som kan köras på egen hand utan att användas ihop med Wireshark eller något program.

Hela CAN-delen uppskattades ta runt två veckor.

5. Vecka 4 – Finslipningar och test

Tiden som återstod efter att Ethernet- och CAN-delen var färdiga planerades att läggas ner på dels finslipning.

Finslipning syftar på att ytterligare finputsa både Ethernet- samt CAN-delen. Till exempel snygga till saker och rätta till småfel. Det omfattar också att implementera utökade funktioner för bästa möjliga slutprodukt. En del av tiden planerades också att läggas ner på testning. Det uppskattades att det skulle krävas en del testning och viktigast av allt; att få testa på den riktiga hårdvaran för att verkligen veta att det fungerar som det ska. Ifall det under arbetets gång dyker upp nya krav och önskemål kan dessa sparas till finslipningsfasen. Denna fas beräknades ta ungefär en vecka.

6. Resterande tid

Efter att testning och eventuella finslipningar var färdiga planerades det att resten av tiden skulle läggas ner på att dokumentera examensarbetets slutrapport samt förbereda inför den slutliga examinationen och opposition. Denna fas uppskattades inte ta någon specifik tid, istället planerades det så att resten av tiden efter att utvecklandet var klart skulle läggas ner på detta. Ifall arbetet går som planeringen så här långt innebär då detta ungefär 5 veckor för denna fasen.