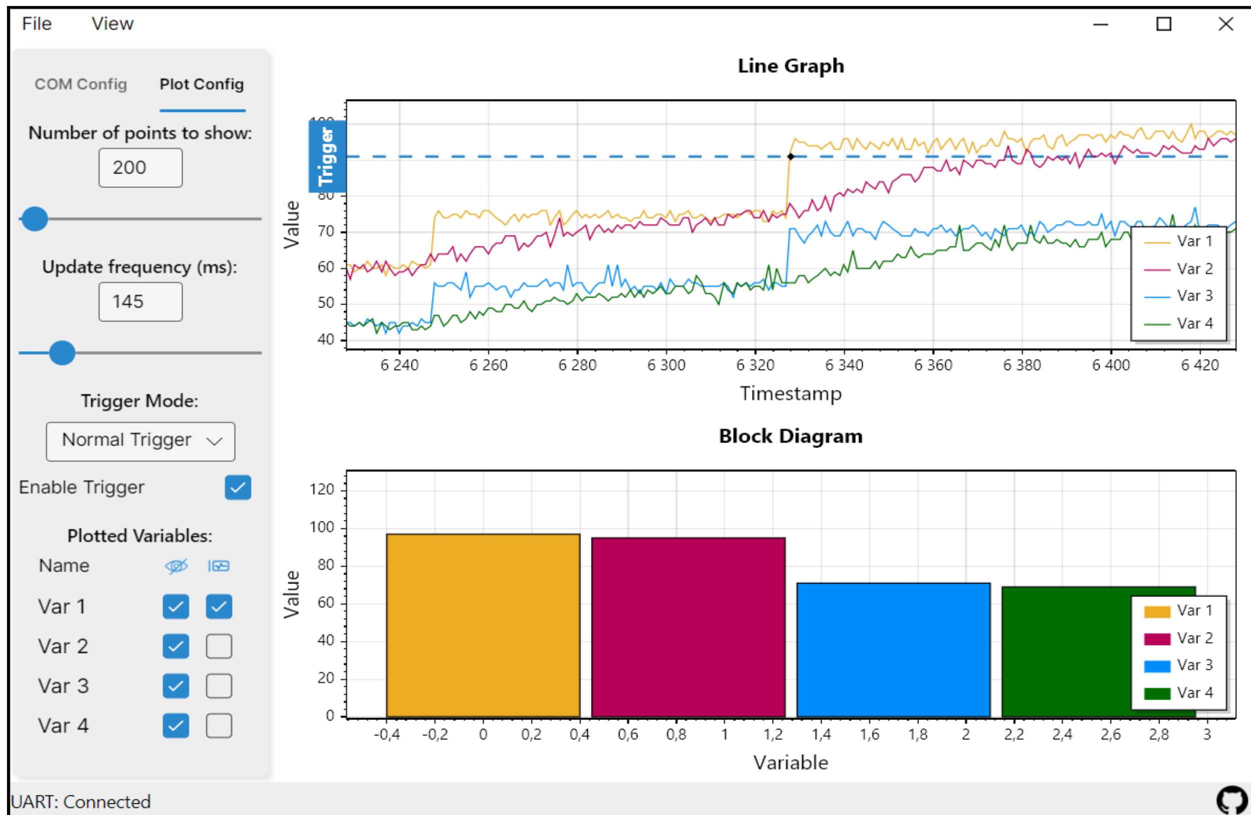
CHALMERS



Design av system för att extrahera och visualisera realtidsdata från inbyggda system under körning via CAN och UART

Flexibel programvara för mikrokontrolleranalys och visualisering

Examensarbete inom högskoleingenjörsprogrammet Datateknik

OSCAR BROBORN
JESPER VESANEN

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK

CHALMERS TEKNISKA HÖGSKOLA

Göteborg, Sverige 2025
www.chalmers.se

Examensarbete 2025

Design av system för att extrahera och visualisera realtidsdata från inbyggda system under körning via CAN och UART

Flexibel programvara för mikrokontrolleranalys och visualisering

OSCAR BROBORN
JESPER VESANEN

Data- och Informationsteknik
Chalmers Tekniska högskola
Göteborg, Sverige 2025

Design av system för att extrahera och visualisera realtidsdata från inbyggda system under körning via CAN och UART
Flexibel programvara för mikrokontrolleranalys och visualisering
Oscar Broborn , Jesper Vesanen

© Oscar Broborn, 2025.
© Jesper Vesanen, 2025.

Handledare: Sakib Sistek, Instutitionen för data- och informationsteknik
Examinator: Nicholas Smallbone, Instutitionen för data- och informationsteknik

Examensarbete 2025
Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Omslagsbild: PC-applikation utvecklad inom examensarbetet för realtidsvisualisering av mikrokontrollerdata

Skriven i LaTeX
Chalmers digitaltryck
Göteborg, Sverige 2025

Design av system för att extrahera och visualisera realtidsdata från inbyggda system under körning via CAN och UART
Flexibel programvara för mikrokontrolleranalys och visualisering
OSCAR BROBORN
JESPER VESANEN
Institutionen för Data- och Informationsteknik
Chalmers tekniska högskola

// Sammanfattning

Detta examensarbete undersöker möjligheten att utveckla ett C-bibliotek för tidsstämpling med hög precision och hög genomströmning vid dataöverföring över UART, avsett för användning i STM32-baserade mikrokontrollerprojekt, och att koppla ihop det med en användarvänlig visualiseringsapplikation. Projektets huvudsakliga mål är att underlätta realtidsutvinning, överföring och analys av inbyggda systemdata för att möjliggöra enkel åtkomst till felsökning och prestandaanalys under drift.

Projektet är uppdelat i två huvuddelar. För det första utformades och implementerades binära protokoll för ett UART-dataöverföringsbibliotek i C för att enkelt kunna integreras i befintliga STM32-projekt. Viktiga funktioner i biblioteket inkluderar konfigurerbar datapaketstorlek, tidsstämpling före överföring och bufferthantering. För det andra utvecklades en plattformsoberoende PC-applikation i C# med Avalonia UI-ramverket och ScottPlot för att rita grafer. Applikationen ansluter till befintlig CAN-trafik för lokal tidsmärkning och dataanalys, eller till UART C-biblioteket för högre precision med hårdvarutidsstämplar. Den stödjer avläsning och plottning av flera variabler samtidigt samt dynamisk aktivering eller inaktivering av deras synlighet. Applikationen erbjuder ett interaktivt linjediagram med zoom- och scrollfunktioner, möjlighet att visa historiskt mottagen data, en oscilloskopliknande triggermekanism och en stapeldiagramvy av de senaste värdena. Konfigurationsfiler kan också sparas och laddas för att minimera upprepade arbetsmoment.

Empirisk testning på ett STM32F407G-DISC1 utvecklingskort visade bibehållen användning av UART C-biblioteket vid överföringshastigheter på över 900 000 baud/s med en samplingsfrekvens på över 30 kHz vid avläsning av en 8-bitars datavariabel. Tester visade också att fler variabler med större datamängder var enkla att hantera vid lägre samplingsfrekvenser. PC-applikationen visade hög prestanda även den, genom att underhålla dataströmmar med över 50 000 datapaket per sekund och då den klarade av att visa tiotals miljoner punkter samtidigt i vyn för historisk data.

Projektets slutsatser visar på att det var möjligt att utveckla både ett effektivt C-bibliotek för dataöverföring via UART, och en PC-applikation för visualisering av mikrokontrollerdata. De mål som sattes för projektet uppnåddes väl, och kunde implementeras med hjälp av standardiserade principer och designmönster för objektorienterad programmering utan att kompromissa med kodkvalitet.

Rapporten är skriven på Svenska.

Keywords: inbyggda system, pc applikation, realtidsvisualisering, plattformsoberoende, uart, can, mvvm, c, csharp, dataanalys

Abstract

This thesis investigates the possibility of developing a C-library for precise timestamping and high throughput data transfer over UART, meant for use in STM32-based microcontroller projects, and coupling it with a user-friendly visualization application. The primary objective of the project is to facilitate real-time extraction, transmission, and analysis of embedded system data to enable easy access to debugging and performance analysis during operation.

The project is divided into two main parts. Firstly, binary protocols for a UART data transmission library in C was designed and implemented to seamlessly integrate into existing STM32 projects. Key features of the library include configurable data packet sizing, timestamping before transmission, and buffer management. Secondly, a cross-platform PC application was developed in C# using the Avalonia UI framework and ScottPlot for rendering plots. The application connects to existing CAN-traffic for local timestamping and data analysis, or to the UART C-library for higher precision using hardware timestamps. The application supports reading and plotting multiple variables at once, and enabling or disabling their visibility dynamically. It offers an interactive line graph with zoom and scroll capabilities, posibility to see historically received data, an oscilloscope-like trigger mechanism, and a block diagram view of the latest values. Configuration files may also be saved and loaded to minimize repeated tasks.

Empirical testing on a STM32F407G-DISC1 development board demonstrated sustained usage of the UART C-library at baud rates exceeding 900 000 baud/s at a sampling rate of over 30kHz reading one 8-bit data variable. The testing also showed more variables at larger data sizes was trivial at lower sampling rates. The PC-application showed results indicating high performance as well, managing to sustain data streams at rates of over 50 000 data packets per second, and display tens of millions of points at once in the historical data-view.

The project's conclusions show that it was possible to develop both an efficient C-library for data transfer via UART and a PC application for visualizing microcontroller data. The goals set for the project were well achieved and could be implemented using standardized principles and design patterns for object-oriented programming, without compromising code quality.

The report is written in Swedish.

Keywords: embedded systems, pc application, real-time visualization, cross-platform, uart, can, mvvm, c, csharp, data analysis

Förord

Arbetet i denna rapport är resultatet av en lärorik och utmanande utvecklingsprocess. Under arbetets gång har vi tagit oss an ett flertal komplexa problem, och utvecklat våran förmåga att tänka kritiskt och metodiskt utveckla den slutprodukt som framkom till följd av det utförda arbetet.

Vi vill även rikta ett stort tack till Fredrik Kensander från KraftPowercon Sweden AB och Sakib Sistek från Chalmers Tekniska Högskola, som med sina insikter och sitt engagemang har väglett och stöttat oss genom projektet.

Vi hoppas att denna rapport inte enbart speglar den tid och arbete vi lagt ned, utan att det även kan fungera som ett underlag för att väcka intresse för de ämnen som behandlas.

Oscar Broborn, Jesper Vesanen, Göteborg, Juni 2025

Akronymer

Nedan följer en lista över akronymer som har använts genom denna uppsats, listade i alfabetisk ordning:

ACK	Acknowledgement
API	Application Programming Interface
ARM	Advanced RISC Machines
BSD	Berkeley Software Distribution
CAN	Controller Area Network
CRC	Cyclic Redundancy Check
DIP	Dependency Inversion Principle
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
IDE	Integrated Development Environment
I/O	Input/Output
ISP	Interface Segregation Principle
JSON	JavaScript Object Notation
LL	Low Layer (Drivers)
LSP	Liskov Substitution Principle
MIT	Massachusetts Institute of Technology
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
OCP	Open-Closed Principle
PC	Personal Computer
RAM	Random Access Memory
ROM	Read-Only Memory
RX	Receive (Register)
SRP	Single Responsibility Principle
TX	Transmit (Register)
UART	Universal Asynchronous Receiver-Transmitter
USART	Universal Synchronous Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
UI	User Interface
XAML	Extensible Application Markup Language
XML	Extensible Markup Language

Figurer

2.1 CAN Frame med alla fält och deras storlekar. (Skapad med ChatGPT, 2025) . . . 8
2.2 UART koppling som överför data till en extern PC. (Skapad med ChatGPT, 2025) . . . 9

3.1 Initialt Gantt-schema för projektets tidsplanering . . . 24

5.1 Applikationens gränssnitt med linjegraf och stapeldiagram aktiverat, utan uppkoppling till mikrokontroller . . . 58
5.2 Lyckad uppkoppling till UART på Linux . . . 59
5.3 Lyckad uppkoppling till CAN på Linux . . . 59
5.4 Linjegraf vid mätning och visualisering av fyra variabler via UART . 60
5.5 Applikationens gränssnitt vid uppkoppling till mikrokontroller via UART, och aktiverad normal trigger på Y=10 för en variabel . . . 60
5.6 Applikationens gränssnitt vid uppkoppling till mikrokontroller via UART, med ett stapeldiagram aktiverat som visar senaste värdet för tre variabler . . . 61
5.7 Filhanterare för att spara och ladda fil på Windows då Save Config eller Load Config har klickats på . . . 62

Innehållsförteckning

Akronymer **xi**

Figurer **xiii**

1 Inledning **1**

1.1 Bakgrund och samarbete . . . 1
1.2 Syfte . . . 2
1.3 Mål . . . 2
1.4 Avgränsningar . . . 3

2 Teoretisk Bakgrund **5**

2.1 Inbyggda system . . . 5
2.1.1 Mikrokontrollers . . . 5
2.1.1.1 STM32F407G-DISC1 . . . 6
2.2 Controller Area Network . . . 7
2.3 Universal Asynchronous Receiver-Transmitter . . . 9
2.4 Versionshantering . . . 10
2.5 C - Programmeringsspråket . . . 10
2.6 CMake . . . 11
2.7 C# och .NET . . . 12
2.7.1 Avalonia UI . . . 12
2.7.2 ScottPlot . . . 13
2.8 API . . . 14
2.9 Programbibliotek . . . 14
2.10 SOLID Principer . . . 14
2.11 Arkitekturmönster . . . 16
2.11.1 Model–view–viewmodel . . . 16
2.12 Designmönster . . . 16
2.12.1 Observer mönstret . . . 17
2.13 Hållbarhet . . . 17
2.13.1 Ekonomisk hållbarhet . . . 17
2.13.2 Ekologisk hållbarhet . . . 18
2.13.3 Social hållbarhet . . . 18
2.14 Etik . . . 19
2.14.1 Konsekvensetik . . . 19
2.14.2 Pliktetik . . . 19

2.14.3 Yrkesetik 19

3 Metod **21**
3.1 Arbetsprocess 21
3.2 Verktyg 22
3.2.1 Obligatoriska verktyg 22
3.2.2 Valda verktyg och utvecklingsmiljöer 23
3.3 Tidsplan och initial planering 23
3.4 Genomförande 24
3.4.1 Undersökning av potentiella lösningar och verktyg 25
3.4.2 Initial kodgranskning och kompilation av given kod 26
3.4.3 Testning av CAN-kommunikation på hårdvarunivå 27
3.4.4 Utveckling av GUI för applikationen 27
3.4.5 Utveckling av hårdvarans UART protokoll 28
3.4.6 Koppling mellan UART och applikationen 30
3.4.7 Möjlighet att konfigurera ritning av variabler 31
3.4.8 Koppling mellan CAN och applikationen 31
3.4.9 Utveckling av trigger funktionanlitet 32
3.4.10 Omstrukturering av kod 33
3.4.11 Endimensionell representation av data 33
3.4.12 Möjligheten att spara och ladda konfigurationsfil 34
3.5 Informationsinsamling 35
3.6 Användning av AI 35

4 Systemkonstruktion **37**
4.1 UART-bibliotek för STM32 37
4.1.1 Initiering av biblioteket 37
4.1.2 Allokerad storlek av datapaket 38
4.1.3 Paketering och tidsstämpling av samplad data 38
4.1.4 Sändning av data från bufferten 39
4.1.5 Aktivering och avbrytning av samplingslogik 41
4.2 Applikationens Datamodeller 41
4.2.1 Lagring av grafdata 41
4.2.2 Definition av UART-datapaketstorlek 42
4.2.3 Tidsstämplad UART-data 43
4.2.4 Triggerlägen 43
4.2.5 Modell för variabelhantering 43
4.3 Applikationens Kommunikationstjänster 44
4.3.1 Mottagning av UART-data i applikationen 44
4.3.2 Mottagning av CAN-data i applikationen 45
4.3.2.1 SocketCanBus 46
4.3.2.2 PeakCanBus 47
4.4 Applikationens övriga tjänster 47
4.4.1 Konfigurationshantering 47
4.4.2 Konfigurationsavläsning 48
4.4.3 Datakanaler 48
4.4.4 Tjänster för hantering av linjegrafer 49

4.4.5 Tjänster för hantering av stapeldiagram 50
4.5 Applikationens Användargränssnitt 51
4.5.1 Huvudfönster 51
4.5.2 Statusfält 52
4.5.3 Grafvy 52
4.5.4 Verktygsfält 52
4.5.5 Sidopanel 53
4.6 Applikationens ViewModels 53
4.6.1 Hantering av statusfält 54
4.6.2 Hantering av grafvy 54
4.6.3 Hantering av verktygsfält 55
4.6.4 Hantering av sidopanel 55
4.7 Licensering 56
4.7.1 MIT license 56
4.7.2 BSD 3-Clause License 56
4.7.3 Anvädning av PEAKCAN drivrutiner 56

5 Resultat **57**
5.1 C-bibliotek för UART-kommunikation 57
5.1.1 Integration i STM32-baserade inbyggda system 57
5.1.2 Tidsstämpling av överförd data 57
5.1.3 Effektiv UART-kommunikation 58
5.2 Visualiseringsapplikation för mikrokontrollerdata 58
5.2.1 Skapande av en grafisk applikation 58
5.2.2 Uppkoppling till kommunikationsgränssnitt 59
5.2.3 Linjegraf för realtidsvisualisering över tidsaxel 59
5.2.4 Triggerfunktionalitet 60
5.2.5 Stapeldiagram för överblick av senaste värden 61
5.2.6 Hantering av konfigurationsfiler 62

6 Diskussion **63**
6.1 Användingsområde och målgrupp 63
6.2 Användning av MVVM arkitektur 64
6.3 Valet av teknologier 65
6.4 Användarvänlighet 66
6.5 Utvecklingsmöjligheter 68
6.6 Etikaspekter 70
6.7 Hållbarhetsaspekter 71

7 Slutsats **73**

1 Inledning

Detta avsnitt introducerar projektets bakgrund och samarbete, samt beskriver dess syfte, mål och slutligen avgränsningar.

1.1 Bakgrund och samarbete

Mikrokontrollers är centrala komponenter inom många tekniska system och industriella applikationer, där de fungerar som hjärnan i olika enheter och processer. Mikrokontrollers användningsområde sträcker sig allt ifrån styrsystem inom bilindustrin till reglering av kraftomvandlare [1]. Eftersom många av produkterna från dessa industrier kan anses vara essentiella för dagens samhälle, är det av stor vikt att dessa fungerar på ett förutsägbart och säkert sätt. I en kontrollerad labbmiljö finns det oftast möjligheten att stanna en processors exekvering av ett program för att kunna analysera de värden som programvaran producerar. För att underlätta analysen av mikrokontrollers existerar även ett par visualiseringsverktyg som exempelvis oscilloskop som tillåter användaren att mäta spänning [2], [3].

Exempelvis inom industrier som utvecklar kraftomvandlare kan det finnas säkerhetsrisker vid analys av mikrokontrollers värden, då denna verksamhet associeras med höga spänningar [4]. Dessa högspänningsindustrier kan därför vara livshotande för människor om man inte vidtar korrekta säkerhetsåtgärder [5]. Detta betyder att det inte alltid är möjligt att stanna processorn på ett säkert och förutsägbart sätt och samtidigt analysera de värden som mikrokontrollern genererar. Mer specifikt kan inte ett vanligt oscilloskop användas som ett visualiseringsverktyg för att underlätta analysen vid höga spänningar [6]. För att reducera kostnaden, samt få en generellt säkrare lösning, kan man istället analysera de inkommande värdena på distans med hjälp av informationsöverföringsprotokoll såsom CAN eller UART. Problematiken är att för närvarande saknar dessa protokoll ett standardiserat och lättillgängligt visualiseringsverktyg.

Denna problematik ligger till grund för ett samarbete med KraftPowercon, som är ett företag som innehar en lång erfarenhet inom kraftomvandlingsindustrin och är ledande i utvecklingen av teknologin samt även lösningar gällande kraftomvandling [7]. KraftPowercon föreslog att en lösning på detta problem skulle vara att tillverka ett effektivt UART-bibliotek för STM32 mikrokontrollers samt ett visualiseringsverktyg som tillsammans underlättar analysen av mikrokontrollers.

1.2 Syfte

Syftet med detta examensarbete är att undersöka möjligheten att utveckla ett C-bibliotek för effektiv tidsstämpling och dataöverföring via UART, anpassat för mikrokontrollersystem, i syfte att möjliggöra kommunikation med en visualiseringsapplikation i realtid. Arbetet syftar även till att utvärdera möjligheten att utveckla ett sådant visualiseringsverktyg som är modulärt, användarvänligt och högpresterande, och som kan användas för att underlätta analys av mikrokontrollerdata.

1.3 Mål

Målet med projektet är att utveckla ett verktyg för att extrahera och visualisera data från inbyggda system under körning. När projektmålen har uppnåtts ska verktyget underlätta analys och felsökning av mikrokontrollersystem i realtid. För att tydliggöra projektets inriktning har följande huvudsakliga delmål formulerats:

- Utveckla ett C-bibliotek för UART-kommunikation som kan integreras i befintlig kod för inbyggda system baserade på STM32-mikrokontrollers. Biblioteket ska stödja effektiv dataöverföring via UART med tidsstämpling.

- Utveckla ett plattformsoberoende PC-program som kan anslutas till en befintlig CAN-bus eller UART via det utvecklade C-biblioteket. Ett grafiskt gränssnitt ska göra det möjligt att välja ut specifik data i trafiken.

- Implementera funktionalitet för en linjegraf i PC-programmet, för att visualisera en eller flera variabler över tid, med möjligheter till zoomning och scrollning genom historisk data.

- Implementera grundläggande triggerfunktionalitet i PC-programmet, liknande den i ett oscilloskop.

- Implementera funktionalitet för ett stapeldiagram i PC-programmet, för att visa det senaste mottagna värdet för varje variabel.

- Implementera stöd för att spara och ladda konfigurationsfiler för enklare uppkoppling till kommunikationsprotokollen.

1.4 Avgränsningar

För att avgränsa projektets omfattning och underlätta utvecklingen av programvaran har följande avgränsningar fastställts:

- Utveckling och testning av C-biblioteket har enbart genomförts med `STM32F407G-DISC1` utvecklingskort, eftersom detta är den hårdvara som tillhandahållits för projektet. Mjukvaran har därför anpassats specifikt för denna plattform.

- Testning av PC-applikationen sker enbart på Windows 10 och Ubuntu, då dessa operativsystem har använts under utvecklingsprocessen. Applikationen är avgränsad till att vara plattformsoberoende inom Windows- och Linuxmiljöer.

- Visualiseringsapplikationen testas enbart med CAN-adaptrar från PEAK-System, då detta var den hårdvara som tillhandahölls under projektet. På Windows är adaptersupporten begränsad till PEAK-Systems egna drivrutiner, eftersom operativsystemet saknar ett standardiserat gränssnitt för CAN-kommunikation. På Linux används i stället SocketCAN, som är integrerat i Linux-kärnan och erbjuder ett generellt API för CAN-kommunikation, vilket möjliggör kompatibilitet med ett brett urval av CAN-adaptrar – även om testning i detta projekt uteslutande har avgränsats till adaptrar från PEAK-System.

- Triggerhanteringen är avgränsad till rising edge samt single- och normaltriggers. Utveckling av andra triggertyper för PC-applikationen ingår inte inom projektets omfattning.

- Fokus på implementering av avancerade felhanteringsmekanismer vid mottagning av mikrokontrollerdata via UART har avgränsats till en enkel handskakning mellan mikrokontroller och applikation, i förmån av högre dataöverföringshastighet.

2
Teoretisk Bakgrund

I detta kapitel presenteras de tekniska och teoretiska begrepp samt de teknologier som anses vara centrala för projektet. Begreppen och teknologierna används för att ge en grundläggande förståelse för de tekniker och metoder som är relevanta för att förstå projektets genomförande och resultat.

2.1 Inbyggda system

Inbyggda system är elektroniska komponenter som används för att utföra ett begränsat antal uppgifter inom ett specifikt användningsområde [8]. Dessa system integreras ofta i allt från vardagliga produkter som exempelvis hushållsapparater, till industriella system för reglering eller automation, oftast utan att användaren av produkten till synes kan se att en sådan komponent finns inbyggd [8].

En fundamental egenskap av inbyggda system är uppsättningen av externa enheter som kan kopplas till systemet. Dessa gör så att ett inbyggt system får möjligheten att interagera med världen omkring sig, fast med sidoeffekten att den inte enkelt kan återanvändas för något helt annat [8]. Exempelvis är det inte rimligt att omprogrammera ett inbyggt system för en termostat till att fungera som en spelkonsol, och tvärtom.

En annan central aspekt är att många inbäddade system måste fungera i realtid. Det betyder att de behöver reagera på händelser enligt givna prestandakrav samt inom förutbestämda tidsgränser för att inte riskera säkerheten av systemets användare [8]. Exempelvis kan ett styrsystem i en bil behöva justera motorregleringen inom millisekunder för att garantera en jämn och säker körning.

2.1.1 Mikrokontrollers

Som tidigare nämnt är inbyggda system en uppsättning av elektriska komponenter som används för att utföra uppgifter. Dessa elektriska komponenter består till störst del främst av en eller flera mikrokontrollers, som konfigureras med avseende på resten av systemet för att uppfylla ett gemensamt mål för det inbyggda systemet [8].

Mikrokontrollers är små datorer som innehåller nödvändiga komponenter för att utföra beräkningar på en enskild krets, och innehåller de komponenter som en vanlig dator har, som en processor, ett minne, och in- och utmatningsportar (I/O) [8]. Dessa mikrokontrollers är oftast hjärtat inom inbyggda system, då det är dessa som programmeras för att utföra specifika uppgifter, som exempelvis temperaturavmätningar eller reglering av externa strömkällor.

Som tidigare beskrivet för inbyggda system kan dessa innehålla enheter för att interagera med varandra eller sin omgivning. Dessa enheter kopplas oftast till mikrokontrollerns minnesmappade I/O-portar, vilket menar på att varje I/O-port i en mikrokontroller tilldelas en adress i samma minnesutrymme som det vanliga Random Access Memory (RAM) minnet, vilket är där temporär data då mikrokontrollern har strömkoppling [8]. På så sätt är det möjligt att läsa av eller skriva till en periferienhet som kopplats till en mikrokontroller med samma datainstruktioner som normalt används när data hanteras i minnet.

Utöver RAM-minnet som volatilt lagrar data enbart när det finns en koppling till ström, har mikrokontrollers ett permanent minne där programkod kan lagras, som ofta kallas för Read-Only Memory (ROM) [8]. En vanligt framkommande form av ROM-minne är Flash-ROM, där data måste raderas i stora block innan ny data kan skrivas, fast med fördelen att det går att uppdatera programvaran i en mikrokontroller utan att byta ut hela kretsen [8]. Detta möjliggör en flexibel och effektiv utveckling med en mikrokontroller, som i sin tur kan användas för bildandet av kompletta inbyggda system.

2.1.1.1 STM32F407G-DISC1

STM32F407G-DISC1 är en 32-bitars Advanced RISC Machine (ARM) Cortex-M4-baserad mikrokontroller utgiven av STMicroelectronics, och kan därmed programmeras med en ARM-verktygskedja för inbyggda system [9]. Mikrokontrollern är utgiven med 1 MB Flash-ROM, samt en inbyggd ST-LINK/V2-A debugger som kan användas för att både programmera mikrokontrollern och felsöka koden direkt via anslutning till en universell seriell buss (USB). Den medför även bland annat en accelerometer, mikrofon, digital-to-analog converter, samt ett flertal lysdioder inbyggda i mikrokontrollern [9]. Strömförsörjning för denna mikrokontroller kan ske via ett flertal kopplingar, som exempelvis direkt via ST-LINK eller USB, vilket resulterar i att den enkelt kan kopplas upp direkt till en persondator (PC) [9].

Mikrokontrollern medför även möjlighet till att kommunicera via ett flertal informationsöverföringsprotokoll, som exempelvis Universal Synchronous Asynchronous Receiver Transmitter (USART), Universal Asynchronous Receiver Transmitter (UART) och Controller Area Network (CAN), vilket gör den flexibel för olika typer av dataöverföring [10]. ST-LINK/V2-A debuggern medför även funktionalitet som underlättar vid informationsöverföring, som exempelvis en Virtuell COM-port [10]. Detta möjliggör för mikrokontrollern att skapa en virtuell port på en PC för att skicka och ta emot seriell kommunikation, som exempelvis kan ske över USB-till-UART som om det vore en traditionell seriell port.

För att underlätta utvecklingen av STM32-mikrokontrollers som STM32F407G-DISC1, har STMicroelectronics utgivit ett grafiskt verktyg vid namn STM32CubeMX, som möjliggör att konfigurera samt generera C-kod för initiering av Cortex-M-kärnan [11]. Detta verktyg underlättar vid grafisk initiering av allmänna in-/utgångar (GPIO), klockinställningar för systemet, samt för konfigurering och tilldelning av periferienheter till Cortex processorn. En viktig aspekt av STM32CubeMX kodgenereringen är att denna inkluderar skapandet av ett application programming interface (API) bibliotek som hardware abstraction layers (HAL) och low layers (LL) [12]. Dessa är mjukvarubibliotek som abstraherar åtkomsten till hårdvarans periferienheter på

olika nivåer, där HAL erbjuder hög-nivå API:er med fokus på portabilitet, medan LL erbjuder låg-nivå API:er som erbjuder direkt registeråtkomst, fast lägre portabilitet [12].

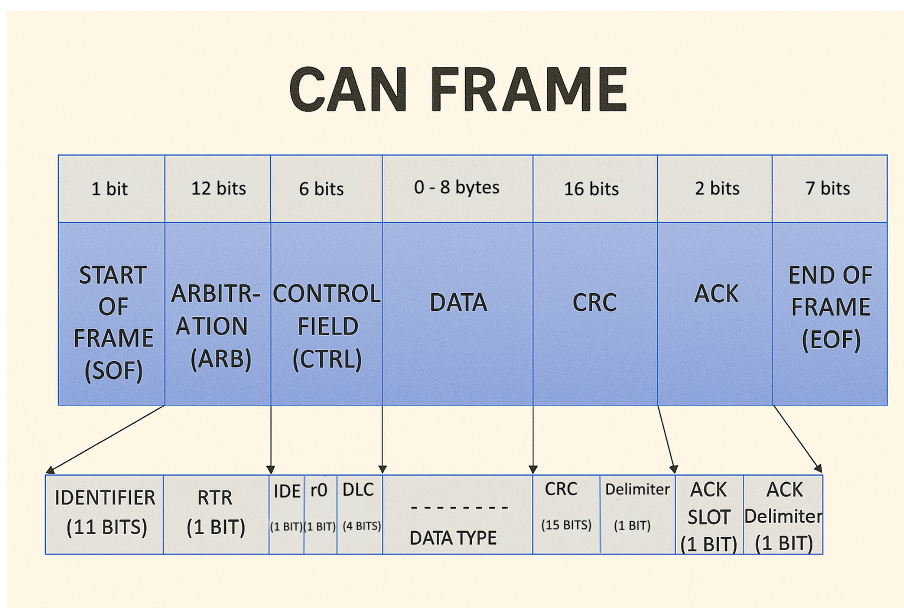
2.2 Controller Area Network

CAN är ett informationsöverföringsprotokoll som använder sig av seriell kommunikation för att överföra data [1]. CAN används exempelvis inom distribuerade system då den effektivt och säkert kan kontrollera och reglera inbyggda systems mikrokontroller i realtid [1]. Överföringshastigheten på CAN är flexibel och kan överföra data från 10 kb/s till en maximal hastighet på 1 Mb/s [1], [13], [14]. Utöver flexibiliteten gällande variation av överföringshastighet anses även CAN vara kostnadseffektiv, då den reducerar mängden kablar som krävs för att kommunicera mellan olika noder [14], [15]. Den reducerade mängden kablar leder även till minskad underhållskostnad, samt att upprätthållningen av kablarna blir simplifierad [15], [16]. CAN används därför ständigt inom bilindustrin [1], där den på låga överföringshastigheter kan användas för att kontrollera bilens fönster eller för att manipulera bilens sätesreglage [17]. På högre överföringshastigheter kan CAN användas för motorhantering eller bromskontroll såsom för låsningsfria bromsar [17].

CAN-protokollet använder sig av en buss-arkitektur för att reducera antalet kablar som behövs användas vid informationsöverföring [1]. Denna arkitektur är uppbyggd genom att varenda enhet är uppkopplad till en och samma buss som sedan skickar ut meddelanden till varenda enhet som är kopplad till bussen [1]. Fördelen är att man inte behöver en central kontroller, som dirigerar vilken rutt varje meddelande ska ta för att kunna kommunicera mellan de olika enheterna [1], [18]. Detta betyder även att CAN-protokollet är modulärt, det vill säga att man flexibelt kan medföra fler enheter in i nätverket utan att man behöver ändra på den struktur som använts tidigare. Varenda enhet som finns på nätverket kan antingen lyssna eller skicka data ut på bussen, som de andra enheterna sedan kan plocka upp [1]. Strukturen blir även oberoende av enheterna som är uppkopplade till nätverket, det vill säga att om en av de avlyssnade eller dataskickande enheterna fallerar, påverkar inte detta direkt hela systemet i sin helhet. Dessa egenskaper såsom modularitet och kostnadseffektivitet medför att CAN-protokollet kan anses vara väldigt skalbart, och därför passande för företagsindustrier.

För att överföra meddelanden mellan de olika enheterna på systemet, är CAN-protokollet uppbyggt av fyra olika datatyper, den första datatypen *data frame* representerar den data som skickas in på bussen, för att sedan läsas av [1]. Denna datatyp är uppbyggd bitvis, och består av en identifierare som används för att lösa konflikter om vilken enhet som ska få skicka data i realtid [1], [16]. Den har även ett kontrollfält som specificerar storleken på den data som kommer skickas [1], [16]. Sedan finns det ett fält där den faktiska datan ligger och den har en kapacitet på upp till 8 bytes per meddelande som skickas [1], [16]. För att öka chansen att bitar inte flippas finns även parity bitar med, vilket görs med ett Cyclic Redundancy Check (CRC) fält som medför möjligheten att avgöra ifall datafältet har blivit korrupt, vilket görs via cyclic redundancy. Slutligen finns även ett så kallat bekräftelse (ACK) fält, som består av två bitar, och används för att en mottagare av data ska kunna

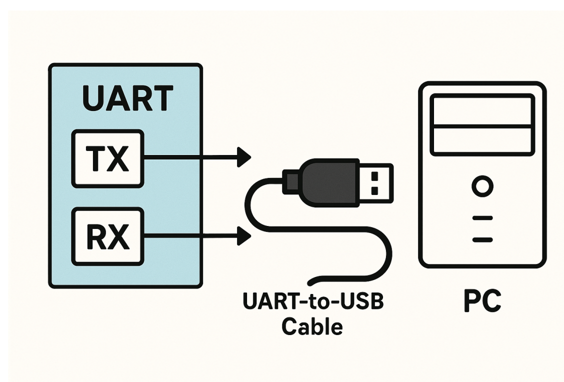
etablera bekräftelsekommunikation med sändaren. Det som skickas tillbaka är en bekräftelse att meddelandet har mottagits och att inga bitar har flippats enligt CRC fältet [1], [16]. Det finns även ett flertal fler fält, se Fig. 2.1.



Figur 2.1: CAN Frame med alla fält och deras storlekar. (Skapad med ChatGPT, 2025)

2.3 Universal Asynchronous Receiver-Transmitter

Universal Asynchronous Receiver-Transmitter (UART) är ett informationsöverförings-protokoll, som oftast framkommer som en fysisk periferienhet på mikrokontrollers [19]. Den används konstant inom industrikontroll men även inom medicinteknik på grund av hur effektivt UART-protokollet överför data mellan olika enheter både gällande överföringshastighet samt även energikostnad [20]. Enkelheten kommer från att det endast krävs två kablar för kommunikation mellan två olika enheter för att antingen överföra data från en enhet eller för att läsa av data [8], [19]. UART saknar också en inbyggd klocksignal för att synkronisera meddelanden [8]. Med andra ord är UART ett väldigt simpelt protokoll, som är uppbyggt av väldigt få komponenter [8]. En UART-enhet har två viktiga pinnar: Receive Register (RX), som tar emot data, och Transmit Register (TX), som skickar data [8], [19]. Värt att notera är att det inte är UART-enheten som genererar någon data, utan den överför data från en enhet till en annan enhet. Exempelvis kan UART användas för att överföra data från en lokal databuss till en annan extern databuss som sedan överför till en extern enhet, som exempelvis en extern PC [19]. Se Fig. 2.2.



Figur 2.2: UART koppling som överför data till en extern PC. (Skapad med ChatGPT, 2025)

Överföringen med UART sker bitvis, den hämtar datan parallellt från en lokal databuss, och sedan överför den datan seriellt från dess TX-pin [21]. Eftersom att datan överförs seriellt så är det även möjligt att överföra data till mer än bara en mottagande UART, exempelvis existerar UART-till-USB kablar. Själva UART-paketet är flexibelt och börjar med en startbit som indikerar att överföring av data kommer att börja [22]. Sedan börjar själva överföringen av den datan som ska läsas av vilket sedan avslutas med en avslutningsbit [8]. Antalet bitar som skickas per meddelande varierar mellan fem till nio bitar data som skickas varav ett par av dessa kan innehålla parity-bitar för att kontrollera att data kommit fram korrekt [8]. Mängden databitar per sekund som en UART kan överföra kallas *baudrate*, för att kommunikationen ska vara stabil måste detta värde vara ungefär samma på alla UART-enheter som ska kommunicera med varandra [8]. Detta är på grund av att den utför sin kommunikation asynkront.

2.4 Versionshantering

Versionshantering är en av de metoder som används för hantering av ett flertal versioner av samma projekt [23], [24]. För att hantera en version av ett projekt måste ändringar som bidragits via tillägg, redigering eller borttagning av filer spåras. Detta innebär att versionshantering registrerar alla ändringar som görs i en fil vilket ger upphov till möjligheten att ångra eller återställa ändringar vid behov. Att spåra vilka ändringar som har gjorts möjliggör även för flera individer att arbeta på samma kopia av projektet med reducerad risk att skriva över andras arbete.

Versionshantering är ett av de verktyg som används för hantering av ett flertal versioner av samma projekt [23]. Denna versionshantering går till på ett sådant sätt att ändringar som bidragits via tillägg, redigeringar, eller borttagningar av filer spåras. Versionshantering registrerar därmed alla ändringar som görs i en fil vilket möjliggör återställning av projektet till en tidigare version. Detta gör även att flera individer kan arbeta på samma version av ett projekt med reducerad risk att överskrida andras arbete [23]. En webbsida som tillåter enkel molnbaserad versionshantering är *GitHub*, som använder sig av versionshanteringsverktyget Git, som tillåter den funktionalitet som nämndes ovan [25].

2.5 C - Programmeringsspråket

C är ett generellt, kompilerat, lågnivå programmeringsspråk som kan användas för en bred uppsättning uppgifter [26]. Det skiljer sig från domänspecifika språk, som är designade för ett specifikt användningsområde [27]. Språket introducerades på 1970-talet, och karakteriseras som ett väldigt effektivt, portabelt och flexibelt språk då det bland annat är användbart för att programmera kompilatorer och operativsystem [26].

Jämfört med många moderna språk medföljer C även mycket karaktäristik från lågnivå språk som Assembly, vilket innefattar att C i sin grund är entrådigt, hanterar tecken, nummer, men även minnesadresser [26]. Följaktligen menar detta att C har möjlighet att komma åt hårdvaruspecifika minnesadresser för att på ett effektivt sätt lagra och manipulera en minnesadress och dess innehåll för fullständig kontroll av hårdvaran [26].

Egenskaperna ovan gör att C anses som ett starkt och effektivt språk för att utveckla inbyggda system. De främsta verktygen inom C som möjliggör detta är *pekare* och *bitmanipulation* [26]. En pekare är en variabel som håller en minnesadress och tillåter manipulering av både minnesadressen samt den data som existerar där [26]. Pekare gör det därmed möjligt att arbeta med minnesadresser istället för bara variabler, som i exemplet nedan, vilket medför en hög grad av kontroll över hårdvaran och dess externa enheter, såsom GPIO-portar och andra periferienheter [26]. Nedanstående kodblock visar initiering av en pekare och dess användning för åtkomst till en minnesaddress:

```
#include <stdint.h>

// Pekare till en specifik 32-bitars minnesaddress
uint32_t *ptr = (uint32_t *) 0x40020C00;

// Uppdatera datan vid minnesadressen
*ptr = 20;
```

Bitmanipulation är det verktyg som C medför för att med precision manipulera specifika bitar i ett minnesregister [26]. Bitvisa operationer som *AND*, *OR*, *XOR* tillåter en programmerare att manipulera specifika bitar i ett givet minnesregister, vilket gör det möjligt att kontrollera enskilda flaggor för att aktivera eller deaktivera specifika funktioner [26]. Nedanstående kodblock visar typiska bitmanipulationer:

```
// Pekare till en specifik 32-bitars minnesadress
uint32_t *ptr = (uint32_t *) 0x40020C00;

// AND: Rensa bit 3 (uppdatera biten till 0)
*ptr &= ~(1 << 3);

// OR: Uppdatera bit 7 till 1
*ptr |= (1 << 7);

// XOR: Invertera bit 15
*ptr ^= (1 << 15);
```

Genom att möjliggöra effektiv minneshantering och exakt kontroll av individuella bitar i minnesregister, är C passande i utvecklingen av allt ifrån operativsystem och drivrutiner till inbyggda system och realtidsapplikationer. Kombinationen av flexibilitet, prestanda och hårdvarunära egenskaper har bidragit till språkets långvariga relevans inom teknikutveckling.

2.6 CMake

CMake är en open-source, plattformsoberoende familj av verktyg som är utformade för att antingen bygga, testa eller paketera programvara [28]. Den använder sig av en generator som bygger upp mjukvaran, vilket möjliggör för utvecklare att de kan beskriva sin byggprocess på en plattforms- och även på en kompilatornivå [28]. Beskrivningen producerar inbyggda system som sedan blir skräddarsydda för specifika miljöer [28].

Det som skiljer CMake från mer traditionella byggsystem som direkt hanterar kompileringsprocessen, genererar CMake konfigurationsfiler [29]. På olika operativsystem kan den nämligen generera olika sorters konfigurationsfiler; på Windows kan den generera Visual Studio-projektfiler, på Linux kan den generera Makefiles, och på macOS kan den skapa Xcode-projekt [29]. Den här flexibiliteten gör det möjligt för utvecklare att underhålla en enda uppsättning av bygginstruktioner som kan

användas på olika plattformar och operativsystem. [29].

En av de viktigaste fördelarna med att använda CMake är att man kan bygga programvaran utanför dess källfil [30]. Alla genererade filer, till exempel objektfiler och körbara filer, kan då istället placeras i en separat mapp [30]. Som följd av detta kan man då även utföra flera byggen samtidigt från samma root-directory, vilket underlättar konfigurationen [30].

2.7 C# och .NET

C# är ett modernt objektorienterat programmeringsspråk utvecklat av Microsoft för *.NET-platformen*, där det även är det populäraste programmeringsspråket [31]. Språket har utvecklats till att bli ett passande språk för utveckling inom många fält, som exempelvis mobilapplikationer, desktopapplikationer, webbutveckling samt servrar [31]. Utöver att följa traditionella objektorienterade principer implementerar språket även andra paradigmer. Exempelvis har C# inslag av funktionell programmering som *pattern matching* och *language integrated query* [31]. Vidare innefattar språket låg-nivå verktyg som tillåter utveckling av högpresterande program med reducerad risk att skriva osäker kod [31]. Även fast språket inför funktionalitet från andra paradigmer, är C# fortfarande ett objektorienterat språk i sin grund. Detta menar att programmerare som är vana vid Java, C++, eller JavaScript direkt bör känna sig bekväma med språket och ha det enkelt att lära sig hur det fungerar [32].

.NET, även kallat *dotnet*, är en gratis utvecklingsplattform som är plattformsoberoende, vilket menar på att utvecklare kan bygga desktop-applikationer som fungerar på Windows, Linux och MacOS [33]. Utvecklingsplattformen innefattar språken C#, F# samt Visual Basic. [34]. .NET-Plattformen är utrustad med automatisk minneshantering via *garbage collection* för att bidra till skapandet av tillförlitliga och säkra program [33].

En central aspekt för utvecklingen av program inom .NET är möjligheten att dela och återanvända kod, vilket sker genom skapandet av ett *paket* som innehåller kompilerad kod i form av DLL-filer [35]. Det verktyget som möjliggör detta för .NET-plattformen är pakethanteraren NuGet, som tillåter användare att söka efter kodbibliotek som kan laddas ner och direkt kallas på i användarens kod [35]. För att underlätta användningen av versionshanteringsverktyg, medför NuGet även en referenslista över alla paket som laddas ner för ett projekt [35]. Detta resulterar i att en användare som laddar ner projektet eller överför det mellan datorer ej behöver medföra alla NuGet paket som har laddats ner, då dessa automatiskt hämtas och instansieras för projektet via referenslistan med kommandot *dotnet restore* [35].

2.7.1 Avalonia UI

Avalonia är ett gratis open-source ramverk för .NET-plattformen som med hög prestanda underlättar utvecklingen av plattformsoberoende desktop-applikationer [36]. Avalonia använder sig utav Extensible Application Markup Language (XAML), ett Extensible Markup Language (XML)-baserat språk som framkommer inom flera användargränssnitt (UI) ramverk, och stödjer även avancerad funktionalitet som databindning för att länka XAML egenskaper till underliggande C# objekt [37].

Denna sortens databindning nyttjas främst vid implementering av Model-View-ViewModel (MVVM) mönstret [38]. Nedan syns ett exempel på hur ett fönster med en text kan skrivas i Avalonias XAML. En simpel databindning som tillåter applikationen att automatiskt uppdatera textfältet då variabeln *Name* uppdateras i den underliggande C-koden har även tilldelats textblocket.

```
<!-- Application Window containing a single TextBlock -->
<Window

  <!-- XML processor namespaces -->
    xmlns="https://github.com/avaloniaui"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <!-- Start av grafiskt interface -->
    <TextBlock Text="{Binding Name}"/>

</Window>
```

Avalonia finns även tillgängligt som ett paket i .NETs pakethanterare NuGet, vilket möjliggör att enkelt ladda ner ramverket och inkludera det i projektet via NuGets referenslista [39].

2.7.2 ScottPlot

ScottPlot är ett gratis open-source bibliotek för .NET-plattformen, vars syfte är att på ett enkelt och effektivt sätt möjliggöra plottning av stora mängder data, som allt ifrån interaktiva linjediagram till stapeldiagram [40].

Kodbiblioteket medför även en stark dokumentation av de funktioner som ScottPlots API ger åtkomst till, vid namn *ScottPlott 5.0 Cookbook*, som även innehåller visuella exempel av alla huvudfunktioner som finns tillgängliga [41]. Dokumentationen visar även att ScottPlot medför hög nog prestanda via *Signal Plots* för att interaktivt kunna plotta och visa miljontals datapunkter i realtid [42].

ScottPlot medför även stöd för biblioteket i applikationer som använder Avalonia UI, via paketet ScottPlot.Avalonia, som möjliggör att lägga till en graf eller ett diagram i en Avalonia-applikations XAML [43]. Utvecklaren av ScottPlot har medvetet gjort valet att utesluta MVVM och databindning från kodbiblioteket för att uppnå en högre prestanda och bättre kontroll över hur nya bilder renderas [44]. Då Avalonia normalt uppmanar användandet av MVVM och databindning, resulterar detta i att undantag gällande databindning måste göras för just graferna om ett MVVM mönster används under utveckling med Avalonia.

Då ScottPlot biblioteket är byggt specifikt för .NET-plattformen, finns det även tillgängligt som ett paket i .NETs pakethanterare NuGet, vilket möjliggör att enkelt ladda ner ramverket och inkludera det i projektet via NuGets referenslista [45].

2.8 API

Ett Application Programming Interface (API) är en uppsättning regler och protokoll som gör det möjligt för olika mjukvaruapplikationer, och möjligtvis hårdvara, att kommunicera och interagera med varandra [46], [47], [48]. Det definierar metoder och dataformat som utvecklare kan använda för att begära och utbyta information mellan olika sorters system, till exempel en mikrokontroller och en extern applikation [49]. API:er används för att reducera komplexiteten hos underliggande mjukvara genom att abstrahera strukturen då man endast exponerar specifika funktioner för att underlätta processen för utvecklare att integrera nya funktioner eller koppla samman tjänster utan att behöva förstå systemets interna struktur [49].

2.9 Programbibliotek

Ett programbibliotek är en färdigskriven mängd av förkompilerad kod, färdigskrivna funktioner och även rutiner som kan antingen laddas ner, eller vara inbyggda i programmeringsspråk [50]. En utvecklare kan använda sig av ett programbibliotek genom att återanvända bibliotekets funktioner fast inom sin egen kod, vilket tillåter dem att inte behöva skriva om kod som redan finns färdig [50]. Ett programbibliotek hjälper därför utvecklare att spara tid och erbjuder ofta redan optimerade lösningar på vanliga förekommande problem [50].

2.10 SOLID Principer

Vid objektorienterad programmering är det av stor vikt att ett program har bra design, och att man undviker att skriva kod på ett sådant sätt som inte är flexibelt, underhållbart, eller återanvändningsbart [51]. Det finns ett antal tecken som tyder på att en objektorienterad design är dålig, som att koden exempelvis är svår att ändra, använda korrekt, återanvända, eller att koden innehåller mycket repetition [51].

För att undvika att designen av en objektorienterad applikation stöter på dessa problem, är det självklart att ett ramverk av principer kan underlätta. En uppsättning av principer för att undvika dålig design är **SOLID** principerna, som innefattar följande [51]:

S*ingle Responsibility Principle*	(SRP)
O*pen-Closed Principle*	(OCP)
L*iskov Substitution Principle*	(LSP)
I*nterface Segregation Principle*	(ISP)
D*ependency Inversion Principle*	(DIP)

Single Responsibility Principle säger att en klass enbart ska ha en anledning för att ändras [51]. Med detta så menas att en klass endast ska ha ett ansvar. Ett exempel på en klass som skulle bryta mot SRP är en klass `TextDocument` som ansvarar både för att skapa text för ett dokument, och att skriva ut det på papper.

Detta bör anses som två olika ansvar, och bör hanteras i olika klasser, exempelvis `TextDocumentWriter` och `TextDocumentPrinter`.

Open-Closed Principle säger att en klass ska vara öppen för utökning men stängd för förändring [51]. Detta menar på att man inte ska sätta samman olika klasser som utför samma funktioner på olika sorters objekt. Ett exempel som bryter mot OCP är om man har en klass `FileExporter`, där klassen behöver ändras varje gång ett nytt filformat ska stödjas. En korrekt lösning hade istället varit att skapa en abstrakt klass, eller ett interface `Exporter`, som implementeras av de nya filformaten, exempelvis `PdfExporter`. Detta resulterar i att klasser som implementerar andra filtyper ej behöver ändras.

Liskov Substitution Principle säger att subtyper ska kunna ersätta sina basklasser utan att förstöra funktionalitet [51]. Ett exempel som bryter mot LSP kan vara om vi har ett interface `FlyingInterface` med en metod *fly()*, och en klass `Penguin` som implementerar detta interface. Eftersom pingviner inte kan flyga, skulle de rimligtvis inte kunna implementera *fly()*. Om en metod förväntar sig ett objekt som implementerar `FlyingInterface`, skulle en `Penguin` inte vara ett lämpligt objekt att använda, då den inte har möjlighet att bidra med en meningsfull implementation av interfacets alla metoder.

Interface Segregation Principle säger att klasser inte ska tvingas implementera metoder de inte behöver [51]. Denna princip hävdar att interfaces inte bör vara för omfattande, utan att det är bättre med ett flertal mindre interfaces som implementeras vid behov. Ett exempel som bryter mot ISP skulle kunna vara ett interface `Animal`, som definierar en metod *Eat()* och *Fly()*. Detta bryter mot ISP då icke-flygande djur hade varit tvungna att implementera metoden *Fly()*, även fast de inte kan flyga. Den korrekta lösningen hade varit att ha två stycken interfaces istället, för `Animal` och `FlyingAnimal`.

Dependency Inversion Principle säger att hög-nivå moduler inte ska bero på låg-nivå moduler, utan att båda ska bero på abstraktioner [51]. Detta innebär att det är bättre att instansiera klasser via sina interfaces, då koden som resultat beror mindre på implementationen och mer på de metoddefinitioner som exponeras av interfacet. Ett exempel som bryter mot DIP är om en klass SQLDatabase implementerar DatabaseInterface, men används direkt i koden som:

```
// Fel enligt DIP
SQLDatabase db = new SQLDatabase();
```

Detta skapar en direkt beroendekoppling till den specifika implementationen. En bättre lösning är att deklarera databasen via sitt interface:

```
// Korrekt enligt DIP
DatabaseInterface db = new SQLDatabase();
```

Resultatet blir ett minskat beroende av en specifik klass, vilket gör koden mer flexibel och utbytbar, då man enkelt kan byta ut databasen till något annat direkt vid deklarationen.

2.11 Arkitekturmönster

Arkitekturmönster är etablerade designlösningar som definierar en strukturerad grund för en applikations uppbyggnad, samt definierar regler och riktlinjer för hur uppdelning av systemets komponenter bör ske [52]. Olika arkitekturmönster, samt varianter av dessa, finns för att uppfylla specifika mål, som exempelvis Model-View-Controller (MVC) för interaktiva system, där UI separeras från datalogiken. Likväl finns *Broker* för distribuerade system, där en central koordinator möjliggör kommunikation mellan tjänster [52].

De flesta mjukvarusystem kan dock inte byggas utifrån ett enda arkitekturmönster. För att uppfylla olika systemkrav krävs ofta en kombination av flera arkitekturmönster, eller att undantag görs där det är lämpligt för att inte introducera onödig komplexitet där det annars inte behövs [52].

2.11.1 Model–view–viewmodel

Model-View-ViewModel, som ofta förkortas som MVVM, är ett vanligt framkommande arkitekturmönster som används för att strukturera applikationer med användargränssnitt [53]. Då detta används för användargränssnitt, kan det jämföras med MVC, som är ett arkitekturmönster för liknande applikationer, fast utan databindning som nämns nedan. Arkitekturmönstrets huvudsakliga syfte är att separera applikationens data- och presentationslogik från användargränssnittet, sådant att applikationen kan utveckla logiken bakom datan separat från hur denna skall uppvisas [54].

Model-klasserna i ett MVVM-strukturerat program är klasser som kapslar in applikationens data, tillsammans med funktioner och valideringslogik som möjliggör användning och interaktion med datan [54].

View-klasserna är ansvariga för att skapa strukturen av applikationens UI. Dessa klasser kan medföra en *code-behind* fil där man med kod kan definiera visuella beteenden för view-klassen utan att gå in i någon extern logik [54].

ViewModel-klasserna är de mellanliggande klasserna som ansvarar för att koppla samman och presentera datan som innefattas av Model-klasserna till relevanta View-klasser [54]. Processen som tillåter en ViewModel att exponera sitt lokala programtillstånd eller innehållet av en Model-klass till en View-klass kallas databindning [53]. Databindning tillåter att en View binder till data som exponeras i ViewModel klassen, och möjliggör även att View-klassen kan få notiser när denna data uppdateras [54].

2.12 Designmönster

Designmönster är etablerade lösningar för frekvent framkommande problem inom mjukvarudesign, som definierar en struktur för hur dessa kan lösas på ett passande sätt [55]. Vid definitionen av ett designmönster beskrivs alltid ett problem som det ämnar att lösa, och även under vilka villkor som mönstret bör appliceras. Exempelvis kan detta handla om hur man på ett enkelt sätt kan notifiera ett bestämt antal objekt om förändringar inom ett annat, fast samtidigt möjliggöra stor flexibilitet och möjlighet för vidareutveckling av datastrukturen [55].

Utöver att beskriva ett problem som designmönstret löser, definierar mönstret även en lösning för vad för element som bör innefattas i den kod som implementerar mönstret. Detta inkluderar relationer, ansvar, och kopplingar mellan objekt [55]. Designmönstret beskriver med andra ord en abstrakt lösning för det givna problemet, som sedan kan modifieras för att på bästa sätt appliceras i utvecklarens kod [55].

2.12.1 Observer mönstret

Observer mönstret är ett designmönster som används ofta inom olika industrier. Observer är av subtypen beteendemönster som skapar en en-till-många relation mellan ett objekt som ska observeras och objekt som observerar [56]. Objekten som är observerade notifierar sina observatörer om förändringar i deras tillstånd och används därför ofta i händelsedrivna system, som ett grafiskt användargränssnitt (GUI) eller realtidsuppdateringar [56].

I mönstret har en så kallad observerare en lista över objekt som observerar som registrerar sig för att få notifieringar [56]. När en förändring sker i de som observeras skickas en uppdatering till alla observerare vilket tillåter de observerare att exekvera någon form av kod [56]. Fördelen med detta är att man får en lös koppling mellan de olika komponenterna, vilket möjliggör att systemet blir mer flexibelt och enklare att utöka [56].

2.13 Hållbarhet

Hållbar utveckling är ett begrepp som enligt Brundtland-definitionen lyder: "*En hållbar utveckling är en utveckling som tillfredsställer dagens behov utan att äventyra kommande generationers möjligheter att tillfredsställa sina behov*"[57]. Detta begrepp är därefter uppdelat i tre dimensioner: ekonomisk hållbarhet, ekologisk hållbarhet och social hållbarhet.

2.13.1 Ekonomisk hållbarhet

Ekonomisk hållbarhet handlar om att bevara och överföra kapital av olika former till framtida generationer. Mer specifikt innebär det att människor inte ska förbruka mer resurser än de naturliga resurser som kan återskapas eller ersättas av andra resurser. Kapital i denna bemärkelse utgår ifrån två former av kapital, där den första formen av kapital benämns ändliga naturresurser och den andra formen av kapital kallas mänskligt skapat kapital. Ändliga naturresurser definieras som de tillgångar som finns i naturen som människor kan använda sig av men som inte är förnybara. Ett exempel på detta är fossila bränslen eller metaller [57]. Utöver ändliga naturresurser finns det även den andra formen av kapital, det vill säga mänskligt skapat kapital som kan definieras som de tillgångar som människor själva har byggt upp [57]. Mänskligt skapat kapital kan därefter delas in i två subtyper, fysiskt kapital och humankapital. Fysiskt kapital är fysiska tillgångar som är skapade av människor och som ligger kvar även om skaparen försvinner, exempelvis saker såsom byggnader, maskiner eller infrastruktur [57]. Humankapital syftar på människors kompetens, utbildning, erfarenhet och arbetsförmåga [57].

2.13.2 Ekologisk hållbarhet

Den andra dimensionen är Ekologisk hållbarhet som handlar om naturens förmåga att producera kapital, samt naturens förmåga att självreglera sina ekosystem [57]. Att vara ekologiskt hållbar innebär egentligen att man inte använder mer resurser från naturen än vad naturen själv kan producera. För att mäta detta finns två centrala begrepp för att mäta hur hållbart något är. Dessa två begrepp benämns naturens produktionsförmåga och naturens assimilationsförmåga. Naturens produktionsförmåga syftar på naturens kapacitet och den tid det tar för naturen att producera förnybara resurser av sig själv [57]. Ett exempel på detta är skog, fisk, vatten och jordbruksprodukter. Naturens assimilationsförmåga handlar om hur naturen reglerar sig själv från potentiellt skadliga ämnen, det vill säga naturens egna förmåga att rengöra sig själv [57]. Ett exempel på detta är naturens förmåga att reglera hur mycket koldioxid som finns i atmosfären.

2.13.3 Social hållbarhet

Hållbarhet handlar inte bara om rent kapital, utan även om sociala regler och rättigheter. Det är därför det finns en tredje dimension som heter social hållbarhet, som handlar mycket om den samhällsstruktur som förs vidare till kommande generationer [57]. Med samhällsstruktur menas mer abstrakta koncept som lagar eller ideologier istället för fysiska objekt. För att få en hög social hållbarhet anses det vara attraktivt att samhället är uppbyggt sådant att grundläggande mänskliga rättigheter respekteras, samt att fördelning av resurser och fördelning av makt sker på ett rättvist sätt. Precis som i de andra två dimensionerna finns det centrala begrepp; för social hållbarhet heter de två centrala begreppen vertikala sociala relationer och horisontella sociala relationer. Det första begreppet är vertikala sociala relationer som beskriver förhållandet mellan enstaka individer och de olika formerna av maktstrukturer som finns i samhället [57]. Dessa maktstrukturer skulle kunna vara saker såsom staten, myndigheter, arbetsgivare eller andra former av institutioner. Det andra begreppet är horisontella sociala relationer vilket handlar mer om relationerna individer emellan [57]. Som ett exempel kan det vara relationerna mellan grannar, vänner, kollegor eller medborgare.

2.14 Etik

Etik är ett begrepp som beskriver på vilket sätt en människa bör bete sig i olika situationer [58]. Inom etiken finns det ett flertal ramverk till hur man mäter etik, det vill säga olika som avgör om ett beteende eller handling är etiskt korrekt eller inte. Beteenden eller handlingar i vissa av dessa ramverk kan anses vara etiskt korrekta, medan samma beteende eller handling kan anses som oetiskt i ett annat ramverk.

2.14.1 Konsekvensetik

Det första ramverket inom etik som kommer förklaras är konsekvensetik, som handlar om direkta och indirekta konsekvenser till både beteende och handlingar [58]. Inom detta ramverk är det endast konsekvenser som spelar roll, inte motiv eller personens avsikter med handlingen. Exempelvis om en individ släpper en produkt som sedan används för att skada människor, är detta en dålig handling enligt konsekvensetiken, även om individen inte menade att direkt skada andra människor. På samma sätt kan det anses vara en god handling att rädda människor enligt konsekvensetiken, även om personen som räddade människorna inte menade denna handling.

2.14.2 Pliktetik

Det andra ramverket inom etik som kommer förklaras är pliktetik, vars etiska teori utgår ifrån att vissa handlingar är moraliskt rätta eller fel i sig själva oberoende av de konsekvenser som medförs med handlingen [58]. Med andra ord handlar det mer om att undvika direkt dåliga handlingar, och att man har en moralisk plikt att bete sig på ett visst sätt, även om konsekvenserna skulle kunna bli stora. Pliktetiken har därför mycket mer fokus på avsikter och principer snarare än på resultat eller konsekvenser [58]. Poängen med pliktetiken handlar om att individer ska betona sitt eget ansvar att alltid göra det rätta"valet. Ett typiskt exempel på pliktetik är att man aldrig ska skada en annan människa. Detta gäller även ifall skadande av denna person hade gynnat miljontals andra personer.

2.14.3 Yrkesetik

Det tredje ramverket är yrkesetik som handlar om så kallade etiska principer och normer, som kan finnas inom specifika yrken och hur personer som utövar dessa yrken bör agera för att ta ansvar gentemot både individer och samhället [58]. Ofta finns en så kallad hederskodex som innehåller särskilda riktlinjer till vad som anses vara etiskt eller inte. Ett exempel på detta är Sveriges Ingenjörers hederskodex, som många anser gälla för ingenjörer. En viktig distinktion mellan pliktetik och yrkesetik är att yrkesetik är mycket mer kontextbaserad. Yrkesetik är därför mer specifik och handlar mycket om hur beteende ska se ut i konkreta situationer samt vilka särskilda moraliska förväntningar som gäller vid dessa situationer. Där pliktetiken säger "*du ska inte ljuga*", kan yrkesetiken för en läkare säga "*du får undanhålla viss information om det gagnar patientens hälsa*". I detta fallet är läkaren oetisk enligt pliktetik, men etisk berättigad inom yrkesetiken [58].

3 Metod

Denna sektion är uppdelad i tre huvuddelar. Den första delen, arbetsprocessen, behandlar de utvecklingsmetoder och projektledningsramverk som använts för att driva projektet framåt. Här förklaras och motiveras de metoder som använts för att föra projektet framåt. Dessutom kommer handledningen under projektet att beskrivas. Efter arbetsprocessen följer verktygssektionen, där de teknologier och resurser som använts i projektet beskrivs. Verktygssektionen är uppdelad i två mindre delsektioner: den första behandlar den givna hårdvaran samt de verktyg som anses obligatoriska för utförandet enligt den specifikation given av KraftPowercon, medan den andra delsektionen behandlar projektgruppens valda mjukvaruverktyg och utvecklingsmiljöer för resterande aspekter av projektet. Avslutningsvis presenteras genomförandet, där de centrala stegen i projektets utveckling beskrivs och motiveras. Här redogörs för hur de olika delarna av systemet har implementerats och integrerats för att uppnå projektets mål.

3.1 Arbetsprocess

Eftersom att projektet ansågs vara oförutsägbart när det kommer till vilka teknologier som är mest lämpliga att tillämpa, var det rimligt att använda sig av någon slags agil projektledning. Det vill säga att kontinuerligt anpassa utfört arbete utifrån nya insikter och behov. Bland annat har gruppen vid flera tillfällen diskuterat olika lösningsförslag, som har testats för att sedan ersättas av en annan lösning som ansågs vara antingen enklare eller mer effektiv. Utöver detta har båda gruppmedlemmarna haft en nära och löpande kommunikation, där nästa steg i projektet regelbundet har diskuterats och även korrigerats efter vad gruppen ansåg vara mest passande för stunden. Till skillnad från en mer traditionell agil struktur, där projektets mål ofta klargörs successivt under arbetets gång, har detta projekt haft tydliga krav och en klar slutvision redan från första början. Det som däremot har varit osäkert är vilka teknologier och lösningar som bäst uppfyller kraven. Därför har detta istället utforskats och anpassats längs vägen och successivt blivit mer tydligt under projektets gång.

För att underlätta arbetet på projektet har gruppen blivit erbjudna vägledning och feedback i form av två handledare som översett projektet. En företagshandledare från företaget KraftPowercon samt även en mer administrativ handledare från Chalmers tekniska högskola. Handledaren från KraftPowercon, som ämnar att tillföra den mer praktiska handledningen, har stöttat gruppen genom att ha konstanta diskussioner som handlar om allt utifrån hur hårdvaran fungerar, till lösningar som

skulle kunna användas för att vidareutveckla projektet. Företagshandledaren har även varit med om frekventa möten, där vi har diskuterat projektets framsteg och har fått värdefulla insikter för att vidareutveckla programvaran på ett företagsvänligt sätt. Parallellt har vi haft kontakt med handledaren från Chalmers, som har bistått med akademisk vägledning och expertis. Bland annat har denna handledare haft en mer administrativ roll, det vill säga att han har väglett gruppen sådant att Chalmers riktlinjer följs, samt att han specificerar vad som är relevant för projektet på en mer administrativ nivå. Det vill säga frågor gällande rapportskrivning eller liknande delmoment. Dessa möten med båda handledarna har varit avgörande för att säkerställa att projektet uppnår både akademiska och praktiska krav.

3.2 Verktyg

Här presenteras alla relevanta verktyg och hårdvaror, både de som gruppen själva valt att använda sig av, men även de obligatoriska verktygen som gruppen fått under projektets gång.

3.2.1 Obligatoriska verktyg

Vid projektets start tilldelades ett antal verktyg och teknologier som var nödvändiga för utvecklingen. Dessa verktyg har använts för både den inbyggda programmeringen och kommunikationen med övriga systemkomponenter.

- **STM32F407G-DISC1** - Ett utvecklingskit med en STM32F407 mikrokontroller, baserad på ARM Cortex M4 arkitekturen.
- **C** – Programmeringsspråket som används för att utveckla mjukvaran till STM32-mikrokontrollern. C valdes eftersom resterande kod för det tilldelade STM32-projektet nyttjade detta, samt att det är standard för inbyggda system då det ger direkt hårdvaruåtkomst samt god prestanda.
- **ARM Toolchain** – Ett utvecklingsverktyg som används för att kompilera och länka C-koden för STM32-mikrokontrollern. ARM Toolchain inkluderar GNU Compiler Collection för ARM.
- **CAN-kabel med en PEAK System CAN-till-USB adapter** – Används för kommunikation mellan en dator och mikrokontrollern. Denna kabel och adapter används för dataöverföring via CAN.
- **GitHub** – Versionshanteringsplattformen som används för att lagra och hantera källkoden under projektets gång.
- **STM32CubeMX** - Grafiskt verktyg för inspektering och konfigurering av STM32 projekt.
- **STM32 ST-LINK Utility** - Verktyg för programmering av mikrokontrollerns Flash-ROM.

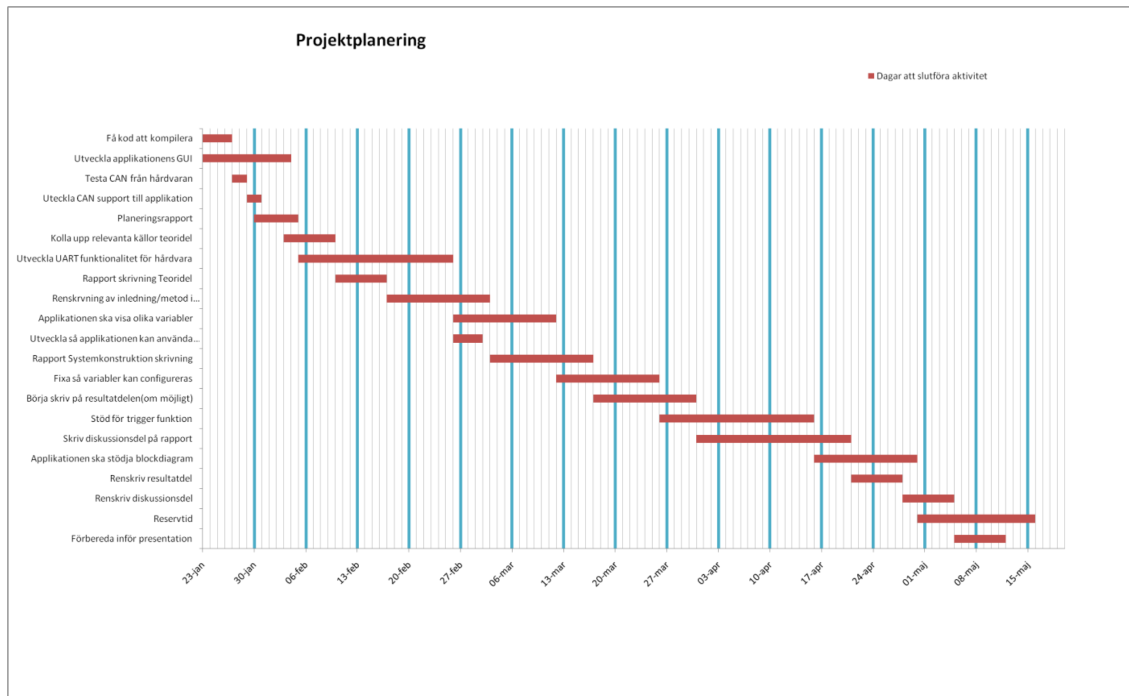
3.2.2 Valda verktyg och utvecklingsmiljöer

Utvecklingen av projektet har genomförts med hjälp av flera centrala teknologier och verktyg, i syfte att skapa en effektiv och plattformsoberoende applikation som fungerar både på Windows och Linux. Nedan framförs de verktyg och utvecklingsmiljöer som har använts:

- **.NET** - Ett framework som inkluderar programmeringspråket C#. .NET används för att tillhandahålla ett brett utbud av bibliotek och verktyg som underlättar utvecklingsprocessen. Ett tydligt exempel på ett bibliotek är AvaloniaUI som nämnts ovan.
- **AvaloniaUI** - Används för att skapa användargränssnittet och säkerställa kompatibilitet med flera operativsystem. AvaloniaUI är ett XAML-baserat UI-ramverk som liknar Windows Presentation Foundation ramverket, men erbjuder plattformsoberoende funktionalitet.
- **C#** - Utgör det primära programmeringsspråket för att implementera applikationens funktionalitet. C# erbjuder stark typning, god prestanda och omfattande stöd inom .NET-plattformen.
- **CLion** - integrerad utvecklingsmiljö (IDE) för C/C++
- **Peak PCANBasic.NET** - .NET Assembly för Peak Systems CAN-adapters
- **Putty** - Terminalemulator för mottagning av seriell data
- **ScottPlot** - Plottningsbibliotek för C#.
- **SocketCanSharp** - Bibliotek för objektorienterad abstraktion till SocketCAN på Linux
- **ReactiveUI** - MVVM ramverk för Avalonia
- **Rider** - IDE för .NET
- **Unix Makefiles** - Unix Makefiles är en generator som kan användas för CMake. Den har använts för att CMake ska kunna bygga upp konfigurationsfilerna inom projektet.

3.3 Tidsplan och initial planering

Innan själva genomförandet av projektet kunde börja, lade gruppen vikt vid att få fram en initial tidsplanering. Denna initiala tidsplanering skapades för att underlätta strukturen på projektet genom att ge en tydlig och konkret struktur på projektet. En Excel-fil skapades för att hålla på flexibel information som kunde uppdateras under projektets gång. Excel-filen fylldes med information för viktiga aktiviteter som identifierades, samt att tidsuppskattningar gjordes för dessa. Utifrån detta material genererades sedan ett Gantt-schema, se Fig. 3.1, som visualiserade vilka aktiviteter i projektet som skulle genomföras och även vilken tidsram aktiviteterna skulle genomföras.



Figur 3.1: Initialt Gantt-schema för projektets tidsplanering

3.4 Genomförande

Projektets genomförande följer en iterativ och flexibel arbetsprocess där olika delar utvecklas och testas parallellt. Nedanstående punkter beskriver de huvudsakliga delarna av utvecklingsarbetet på en hög nivå, i den ordningen som de utfördes. Efter punktlistan kommer varenda delmoment att beskrivas i mer detalj. Delmomenten inom projektet är:

- **Undersökning av potentiella lösningar och verktyg**:
  Projektbeskrivningen togs fram av handledaren från företaget. Beskrivningen analyserades och olika programvaror och verktyg diskuterades inom gruppen, och ett par potentiella lösningar togs fram.
- **Initial kodgranskning och kompilation av given kod**:
  Första steget var att först analysera koden samt att säkerställa att koden kan kompileras på ett korrekt sätt med den givna utvecklingsmiljön.
- **Testning av CAN-kommunikation på hårdvarunivå**:
  För att verifiera att mikrokontrollerns CAN-protokoll fungerade som förväntat, utfördes ett test genom att skapa ett preliminärt CAN-bibliotek för Windows och Linux i C#, då detta var något den fullständiga applikationen krävde. En CAN-bus kopplades sedan till en USB-port på en PC för avläsning av CAN-bussens trafik.
- **Utveckling av GUI för applikationen**:
  Ett grafisk användargränssnittsapplikation som har utvecklas med C# via AvaloniaUI för att möjliggöra plattformsoberoende funktionalitet som kan visualisera input data.

- **Utveckling av hårdvarans UART protokoll**:
Seriell kommunikationsprotokoll via UART implementeras på hårdvaran för att tillåta effektiv dataöverföring.
- **Koppling mellan UART och applikationen**:
UART-protokollet kopplades till den externa applikationen, som tar emot den överförda data som sedan visualiseras i applikationen.
- **Möjlighet att konfigurera ritning av variabler**:
Möjligheten att välja specifika variabler utvecklades, vilket tillåter användaren att specifiera hur många variabler som ska visualiseras och även när de ska ritas ut.
- **Koppling mellan CAN och applikationen**:
CAN-protokollet kopplades till applikationen för att möjliggöra avläsning av datatrafiken för vidare visualisering i applikationen.
- **Utveckling av trigger funktionanlitet**:
Funktionalitet för oscilloskop-liknande triggersystem utvecklades för att möjliggöra händelsestyrd analys av CAN eller UART-datan.
- **Omstrukturering av kod**:
Fokus skiftades till omstrukturering av kod för att vidareutveckla projektets arkitektur för att enklare implementera nya funktionaliteter utan att påverka de redan existerande funktionaliteterna.
- **Endimensionell representation av data**:
En endimensionell representation av data utvecklades i form av stapeldiagram, för att ge en enklare översikt över datatrafikens senaste mottagna värden.
- **Möjligheten att spara och ladda konfigurationsfil**:
Applikationen utökades med stöd för skapandet av konfigurationsfiler, där användarens inställningar för anslutning av ett kommunikationsprotokoll kan lagras.

3.4.1 Undersökning av potentiella lösningar och verktyg

Då uppgiftsbeskrivningen var given och efterfrågade en PC-applikation som med hög prestanda kan infoga data som är överförd via både UART och CAN. Därför påbörjades en undersökning om relevanta teknologier eller verktyg som skulle kunna användas för att utveckla ett visualiseringsverktyg som ska kunna avläsa data via både CAN och UART och rita ut dessa värden på en graf.

Den initiala undersökningen av relevanta verktyg påbörjades med en undersökning om det var möjligt att tillverka programvaran i programmeringsspråket Python. Python har många inbyggda bibliotek och är ett väldigt populärt programmeringsspråk som är enkelt att utveckla produkter med [59]. Två stycken relevanta Python bibliotek hittades, den första vid namn Matplotlib, som är ett open-source bibliotek som kan användas för att rita ut interaktiva grafer [60]. Det andra biblioteket heter PyQtGraph som också är ett open-source bibliotek, med fokus på att användas inom vetenskapliga sammanhang för att rita ut mer komplexa ritningar med plattformsoberoende funktionalitet [61]. Båda biblioteken utsattes för stresstest genom att rita ut mellan ett par hundra punkter i sekunden till några tusentals i sekunden. Under dessa stresstester observerades hur väl dessa bibliotek presterade vid plottning

i realtid, där resultatet av dessa test visade att biblioteken inte var högpresterande nog för ändamålet.

Yttligare ett annat alternativ som undersöktes är att tillverka en webbapplikation via ElectricUI. ElectricUI är ett ramverk som erbjuder bibliotek som tillåter användaren att på ett enkelt sätt tillverka en webbapplikation som använder sig av gränsnitt som kan uppdateras i realtid [62]. Att använda ramverket kostar $80 om året för privatpersoner [63]. Tillsammans med ElectricUI kan man använda Electron, en open-source plattform som tillåter att göra om en webbapplikation till en skrivbordsapplikation [64].

Programmeringsspråket C++ ansågs också vara ett alternativ. Specifikt undersöktes biblioteket QCustomPlot på en ytlig nivå. QCustomPlot är ett bibliotek som tillåter utveckling av 2D grafer som har hög prestanda och är specialdesignad för realtidssystem [65]. Innan projektgruppen utförde tester på detta, undersöktes C# som ett alternativ då projektgruppen har erfarenhet i Java, som är ett väldigt liknande objektorienterat språk vilket hade resulterat i en enklare inlärningskurva.

Det ramverk som bestämdes att användas under projektets gång är plattformen .NET med programmeringsspråket C#. Biblioteken AvaloniaUI och ScottPlot undersöktes och det bestämdes att dessa ska användas för att tillverka gränssnitten samt bygga upp de grafer som applikationen kommer att använda. Dessa bibliotek testades under stress genom att upp till tiotusentals punkter ritades ut per sekund. Utifrån dessa empiriska tester observerades det att dessa två bibliotek är högpresterande och även fungerade på såväl Windows som Linux.

3.4.2 Initial kodgranskning och kompilation av given kod

När gruppen anledde till företaget KraftPowercon tilldelades ett redan existerande STM32-baserat mikrokontrollerprojekt som skulle användas under utveckling, men även den relevanta hårdvaran och kopplingssladdar. Den givna koden analyserades, där det efter en längre analys var tydligt att koden innehåller kod som är en blandning av företagets implementationer av olika funktioner till mikrokontrollern, men även färdiga drivrutiner från STM32 kortet. Utefter detta riktades mer fokus på STM32 drivrutinerna, då dessa kunde användas för att implementera ett mer generellt C-bibliotek.

Den givna koden innehöll även en CMake-fil för att kunna bygga projektet. Problematik dök upp med denna CMake-fil, då det vid försök av kompilering uppstod en massa error, där det stod att '*nmake*' saknades. Efter undersökning insågs det att det saknades en generator. Gruppen laddade då ner UNIX makefiles, för att komplettera detta, och ställde in denna som den generator som skall användas av CMake. Därefter kunde koden kompilera och genererade en hex fil. Denna hex fil kunde sedan flashas in i STM32-kortet via programvaran ST-Link Utility. Därefter kunde utvecklingen påbörjas.

3.4.3 Testning av CAN-kommunikation på hårdvarunivå

När koden väl kompilerat och en hexfil genererades, flashades hårdvaran med den givna koden via mikrokontrollerns ST-LINK/V2-A, för att sedan testa hur CAN-kommunikationen fungerar. Den givna C-koden skummades igenom för att verifiera att CAN-trafik redan existerade och skickades, sådant att avläsning av CAN-trafik är möjlig via en PC. Det identifierades att ström-värden skickades på CAN-bussen via en 100 Hz avbrotts-timer. Peak Systems officiella C# kodbibliotek för Windows laddades ner för att koppla till den befintliga CAN-bussen, då en Peak Systems CAN till USB-adapter användes. Efter att metoder för grundläggande avläsning på en separat tråd implementerades var det möjligt att läsa CAN-trafiken genom utskrivning i en terminal. Eftersom att avläsningen lyckades abstraherades därefter koden via interfaces, och utökades till att även tillåta koppling till CAN-bussen via Linux genom Linux-kärnans inbyggda SocketCAN driverbibliotek. Därefter var det även möjligt att använda det preliminära CAN-biblioteket för att läsa av mikrokontrollerns CAN-trafik på en terminal i Linux.

3.4.4 Utveckling av GUI för applikationen

Efter att tester visar på att hårdvaran och dess kommunikation fungerade som förväntat, var det nästa steget att börja utveckla den visualiseringsapplikation som är till grund för visualiseringen av den data som kommer att avläsas. Första steget var att designa den UI som applikationen skulle använda sig av. Denna process gjordes muntligt mellan de två gruppmedlemmarna, där det bestämdes att XAML ska användas för skapandet av gränssnitt till följd av den inbyggda supporten inom AvaloniaUI. Strukturen som togs fram är att det skulle finnas tre huvudkomponenter i applikationen. Där dessa tre komponenter skulle vara modulära och oberoende av varandra.

Den första komponenten var att det skulle finnas en header som skulle finnas längst upp i applikationen, som representerar ett verktygsfält. Verktygsfältet skulle ansvara för funktioner som angår applikationens struktur. Exempelvis skulle programvaran inkludera ett fält i verktygsfältet som ändrar vilket slags diagram som ska visualiseras. Det bestämdes även att verktygsfältet skulle ha ansvaret att kunna spara och ladda konfigurationsfiler. Detta gjordes genom att verktygsfältet innehåller två stycken knappar som har varsin dropdown meny när man klickar på en av knapparna. Den första knappen ska innefatta texten *File*, som vid ett klick ska fälla ner en rullgardinsmeny som visar två ytterligare knappar med texten *Save config* och *Load config*. På ett liknande sätt ska den andra knappen implementeras, där det står *View*, och som vid ett klick ska fälla ner en rullgardinsmeny där det ska finnas alternativ för hur det grafiska området ska representeras. Bland annat ska detta innefatta knappar för *Toggle Sidebar*, *Toggle Line Graph*, och *Toggle Block Diagram*.

Efter att verktygsfältets design var färdig designad, designades själva huvudområdet för applikationen. Det första gruppen kom fram till var att huvudområdet bör vara uppdelat i två mindre komponenter, där den första delkomponenten är ett sidofält för konfigurationer och den andra delkomponenten består av diverse grafer som applikationen skulle använda. Sidofältet designades sådant att det har ett flertal

sidor som man kan navigera mellan. Den första sidan som skapades för sidofältet var gränssnittet för kommunikationsprotokollen, där det först finns ett fält där man kan välja antingen UART eller CAN. Det designades sådant att om användaren klickar på UART dyker de fält som är relevanta för kopplingen med UART upp. Detta inkluderar fält för vilken port UART-kopplingen ligger på, vilken baud rate avläsning ska ske på, samt även en rullgardinsmeny för färdigställda datastorlekar. Liknande designades det för CAN, men med andra fält som passade CAN-protokollet bättre. Den andra delkomponenten designades för att hålla två stycken grafer, som skapades via ScottPlot biblioteket. Dessa menades vara den destination där inkommande data ska tas emot och grafiskt ritas upp på lämpligt sätt.

Den sista komponenten som designades och implementerades var en footer som ligger längst ner i applikationen. Denna footer designades sådant att den kan hålla på meddelanden, och bestämdes därmed för att kallas applikationens statusfält. Denna komponent bestämdes vara där potentiella felmeddelanden kommer att visas upp, eller annars vad för protokoll som applikationen är kopplad till.

3.4.5 Utveckling av hårdvarans UART protokoll

När det väl hade testats att det var möjligt att kommunicera mellan mikrokontrollern och en PC via CAN samt att ett grundläggande gränssnitt fanns, var det följande steget att även göra ett likadant terminaltest för UART. Detta gjordes genom att konstruera en simpel buffert och att med en loop tillföra ett och samma värde in i bufferten om och om igen. LL-drivrutinerna användes sedan för att skicka ut värdena från bufferten ut via UART. Datorn kopplades till mikrokontrollern via en UART-till-USB adapter. Därefter observerades de inkommande värdena med programvaran Putty, där samma ASCII-värden som inmatades i bufferten printades ut upprepade gånger.

Efter det initiala testet påbörjades utvecklingen av UART-protokollet. Det första steget var att bestämma hur UART-paketen skulle se ut. Bland annat skulle paketen innehålla både data och tidstämpel, frågan var mest hur många bitar data och hur många bitar tidstämpel paketen skulle ha. Den första iterationen av storleken för UART-paket som designades var att varje frame skulle innehålla 12 bitar data och 12 bitar tidstämpel. Efter en analys av alla LL-drivrutiners inbyggda funktioner, hittades *LL_USART_SetDataWidth* som används för att sätta storleken på UART-paketen. Denna funktion kan ställas in med antingen makrot *LL_USART_DATAWIDTH_9B* för 9 bitar och även *LL_USART_DATAWIDTH_8B* för 8 bitar per paket. Det gick även att specificera varken en parity bit faktiskt ska skickas eller inte vid transmission av data.

Den initiala designen för UART-protokollet bestod av 12 bitar data och 12 bitar tidstämpel. Det insågs snabbt att detta var en dålig lösning, då högsta optimeringen kräver att värden från både datafältet och tidsstämpeln paketeras i en och samma byte då storlekarna ej tar upp fullständiga bytes. Gruppen bestämde att använda 8 bitar per paket utan parity bit, för att maximera prestanda, och designade istället protokollet sådant att det är möjligt att flexibelt kunna använda 8-, 16- eller 32-bitars värde av data, men alltid 16-bitars tidstämpel. Emot slutet av arbetet vidareutvecklades detta sådant att tidsstämpeln enbart är 8 bitar, och att applikationen skulle

hantera de frekventa överflöden som sker i de mottagna tidsstämplarna.

Efter att ha implementerat storleken på datastrukturen, insågs det snabbt att UART:en saknar en inbyggd transmit buffer i mikrokontrollern. För att garantera att ny data inte skriver över gammal data, var nästa steg att implementera logiken för att kunna lagra värden lokalt på en extern buffert. Först designades buffertens hantering av data där ett flertal alternativ föreslogs inom gruppen; bland annat föreslogs att man kan använda två stycken buffertar, där den ena bufferten tar emot ny information, och den andra bufferten skickar ut information. Sedan, när den första bufferten skickat färdigt, byter man vilken buffert som skickar eller tar emot data. En av fördelarna som diskuterades med denna implementation är att den kan hantera en stor mängd data under en kort mängd tid. En nackdel som insågs med denna implementation är att det krävs en överväldigande mängd minne för att ha två stycken buffertar. Ett annat alternativ som är mer minneseffektivt var att implementera en cirkulär buffert som skulle kunna implementeras med en array. Arrayen har två stycken index, head och tail, där head skriver in data, och tail läser av data. Skulle antingen head eller tail hamna utanför arrayen, ska den ställas om till början av arrayen. Bufferten skulle anses vara full ifall både head och tail-indexen har samma värde. En av fördelarna som hittades med denna lösning är att den är minneseffektiv, samt att läsning och skrivning kan ske parallellt utan att data måste bearbetas.

Med ett fungerande buffertsystem var det möjligt att börja implementera logiken för att kunna skicka ut data från en mikrokontroller. För att åstadkomma detta bestämdes det att funktionerna *UART_StoreData*, *UART_FlushOne* och *UART_-FlushBuffer* ska implementeras. *UART_StoreData* implementerades som en funktion som samplar både relevant data och tidsstämpel till den implementerade cirkulära bufferten. Sedan implementerades *UART_FlushOne* som en funktion som kollar om samplad data finns, och skickar denna i först in, först ut-ordning över UART. Till sist implementerades även *UART_FlushBuffer* som är en funktion som liknar *UART_FlushOne*, men som istället för att enbart skicka ett värde från bufferten, skickar hela buffertens innehåll över UART.

3.4.6 Koppling mellan UART och applikationen

Efter att UART-protokollet var implementerat var det nästkommande steget att få en direkt koppling mellan hårdvaran och applikationen för att möjliggöra utritning av data. Systemet konfigurerades för att använda UART-kommunikation baserat på användarens inmatade konfiguration. Vid initiering sattes UART upp enligt konfigurationen, och inkommande data hanterades genom ett eventdrivet system, där underliggande tjänster väntade på att ta emot data från ett kopplat kommunikationsprotokoll. Vid mottagandet av data registrerades den mottagna datan i en datamodell vars uppgift är att hålla ordning på grafdata.

För kontinuerliga UI-uppdateringar implementerades det i grafernas viewmodel en timer, vars uppgift var att hantera uppdaterings-loopar för de enskilda graferna. Interna metoder för varje grafs uppdateringssekvens registrerades till denna timer, för att sedan kallas på periodiskt. Dessa brukade först hjälpmetoder för att åstadkomma grafisk ritning, fast byttes senare ut mot hantering av separata tjänstklasser istället då detta var en mer flexibel lösning.

3.4.7 Möjlighet att konfigurera ritning av variabler

Efter att grafisk ritning av variabler som skickas över UART möjliggjordes, insåg projektgruppen att det vid många variabler är svårt att urskilja dem under ritning. Ett möte utfördes med handledaren, där det bestämdes att det bör vara möjligt att växla synligheten av specifika variabler i grafen. Till följd av detta beslutet utvecklades en ny flik i sidopanelen vid namn *Plot Config* med ändamålet att innehålla verktyg för konfiguration av den grafiska utritningen. En scrollbar lista implementerades inom denna flik där det efter uppkoppling till ett kommunikationsprotokoll går att se alla enskilda variabler som ritas ut. För UART bestämdes antalet variabler att listas upp direkt via användarens inmatning, medan det för CAN räknades ut efter att användaren har markerat vilka delar av datan som ska avläsas för olika variabler. Funktionalitet lades till för att klicka på namnen i listan för manuell namngivning av variabler, samt en kryssruta för att växla synligheten.

För att vidareutveckla användarens möjlighet att justera den grafiska ritningen av variabler lades det även till ett skjutreglage för justering av grafens uppdateringsfrekvens i millisekunder. Tanken med detta var att användaren vid stora mängder data ska kunna ställa in grafen till att uppdateras mer frekvent. För att göra detta mer användarvänligt visas den aktuella uppdateringsfrekvensen i millisekunder som en redigerbar numerisk text, sådant att det är möjligt att manuellt ändra den.

Vidare möjliggjordes justering även för antalet variabler som visas på grafen under ritning. Precis som tidigare implementerades detta med ett skjutreglage med en redigerbar numerisk text för manuell sifferinmatning. Detta implementerades på ett sätt sådant att x-axeln automatiskt anpassas efter antalet valda variabler, genom att bredda fönstret långt nog bakåt i tiden för att visa alla de punkter som befinner sig inom det inställda intervallet.

3.4.8 Koppling mellan CAN och applikationen

Då fullt stöd för UART fanns tillgängligt inom applikationen påbörjades arbete med att färdigställa stödet för CAN i applikationen. CAN-biblioteket som skapades tidigare med stöd för både Windows och Linux utvecklades vidare efter korta diskussioner med företagets handledare, sådant att tidsstämplar skapas i applikationen vid mottagning av data jämfört med tidigare. Detta möjliggjorde att CAN-trafiken kunde läsas av oberoende av om den innefattade en tidsstämpel i datafältet eller ej. Vidare spenderades lite tid på optimering av biblioteket för att drastiskt minska skapandet av C# objekt som behöver hanteras av dess garbage collector.

Efter detta arbete var det möjligt att koppla samman CAN med applikationens UI. Det säkerställdes att alla relevanta konfigurationsfält fanns i applikationens sidebar då CAN valdes som kommunikationsgränssnitt. Efteråt skapades databindningar som möjliggjorde att de angivna konfigurationsvärdena kunde skickas till grafens viewmodel för initiering av CAN-biblioteket.

Enligt den design som projektgruppen tidigare kom överens om, skedde initiering av CAN-biblioteket på ett väldigt likvärdigt sätt som för UART. Det vill säga att kommunikationsprotokollet initieras och tillåter prenumeration till ett *event* som tillåter viss kod att utföras när ny data anländer. I fallet av CAN, designades koden som utfördes via detta event sådant att när data togs emot, jämfördes den med en konfiguration som sparades inom programmet vid initiering, för att sedan föras in i grafens lista av punkter att rita ut. Utritning av dessa punkter som avläses sker på exakt samma sätt som för UART.

3.4.9 Utveckling av trigger funktionanlitet

Efter implementering av CAN och UART påbörjades arbetet med att förbättra förmågan att analysera data genom triggerfunktionalitet, liknande ett oscilloskop. Applikationens UI vidareutvecklades sådant att en kryssruta existerar för att aktivera och inaktivera en triggernivå inom *Plot Config* fliken i sidopanelen. Användarupplevelsen prioriterades vid implementation av triggerfunktionaliteten, och därför implementerades den på ett sådant sätt att när kryssrutan markeras av användaren dyker en tydlig horisontell linje upp i grafen. Justering av triggernivån implementerades genom en interaktiv metod där användaren skulle kunna dra triggernivån vertikalt i grafen med muspekaren. Noggrann justering av triggerpunkten möjliggjordes då en numerisk indikator placerades bredvid triggernivån i grafen.

När gränssnittet väl var implementerat för triggerfunktionalitet, var det rimligt att börja implementera logiken bakom triggerfunktionaliteten, det vill säga hur applikationen skulle reagera ifall en trigger går igenom. Gruppmedlemmarna började därför att diskutera vilka sorters triggers som skulle implementeras. Flera alternativ övervägdes, men gruppen beslutade att inledningsvis fokusera på att applikationen initialt ska ha stöd för single triggers. Tid spenderades på att informationssöka om definitionen av en single trigger. Det som framkom var att ifall ett värde överstiger triggernivån, så avslutas inhämtningen av ny data, och användaren får istället en stillbild för att analysera vad som skett. Det bestämdes att single triggers bör implementeras sådant att applikationen väntar på en trigger, och om en trigger sker centreras triggerpunkten, och applikationen fortsätter att läsa in data i 2 sekunder, och därefter fryser applikationen sådant att användaren kan se data både innan och efter triggermomentet.

Efter en diskussion med handledaren bestämde sig gruppen för att vidareutveckla triggerfunktionaliteten för stöd av normal triggers. Denna sorts trigger handlar om att applikationen endast ska uppdatera grafen om ett värde passerar triggernivån. Vid diskussionen med handledaren diskuterades även om normal trigger bör ske vid positiv eller negativ flank vid passering av triggernivån, där det bestämdes att positiv flank var av störst vikt, och som sedan implementerades.

Det insågs att det fanns en stor brist i implementeringen av triggerfunktionaliteten, då det skulle vara möjligt för användarna att kunna specificera vilka variabler som ska kunna orsaka triggers. För att åstadkomma detta lades det till en till kryssruta till höger om varje variabel i variabellistan. Markeras kryssrutan tilläggs variabeln i en triggerkanal, vilket indikerar att den har möjlighet att orsaka triggers.

3.4.10 Omstrukturering av kod

Vid planering av ytterligare grafiska representationer av data diskuterades det om det skulle vara av någon nytta att först omstrukturera en del av den redan existerande koden. Det bestämdes att det bästa valet var att skifta fokus till att omstrukturera koden i syfte att förbättra struktur, ansvarsfördelning och underhållbarhet innan nästkommande moment. Arbetet utgick från etablerade designprinciper, särskilt SOLID-principerna, för att strukturera upp den existerande koden i en mer modulär och utbyggbar arkitektur.

Ändringar som gjordes var främst att minska de uppgifter som hanterades utav enstaka ViewModel klasser. Detta ledde i sin tur främst till att konfigurationstolkning, kommunikationskanaler och UI-logik som datautvinning, triggeravläsning, och grafuppdatering separerades till enskilda tjänster, som även abstraherades via gränssnitt.

Syftet med detta moment var att undvika framtida problem med avseende på tätt kopplade komponenter i applikationen. Främst gällde detta inför hanteringen av två olika grafrepresentationer av samma data, där respektive graf med tidigare implementation hade varit beroende av varandras klassvariabler. Genom omstrukturering av koden säkerställdes det att dessa komponenter konfigureras separat och oberoende av varandra. Liknande principer applicerades för alla de tidigare nämnda tjänsterna som skapades, vilket möjliggjorde en mer modulär struktur där nya funktionaliteter kan läggas till utan onödiga justeringar av andra delar i kodbasen.

3.4.11 Endimensionell representation av data

Efter omformateratering och renskrivning av koden, var det nästkommande steget att implementera stapeldiagram. Stapeldiagram är en funktionalitet som var efterfrågad av handledaren sedan början av projektet. Därför började gruppmedlemmarna med att diskutera vad exakt stapeldiagrammen ska användas till inom applikationen. För att underlätta förståelsen kontaktades handledaren på företaget för en tydligare beskrivning av vad som efterfrågades gällande stapeldiagram. Efter en kort diskussion kom gruppen fram till att stapeldiagrammen kan användas för att utöka programvaran med funktionaliteten att visualisera variablers nuvarande värde på ett endimensionellt plan.

Efter att gruppen kommit överens om vad stapeldiagrammen ska användas till var det första steget att designa hur variablerna ska läggas på stapeldiagrammet, mer specifikt om staplarna skulle ändra storlek dynamiskt, där fönsterstorleken avgör staplarnas storlek, eller om staplarna skulle vara konstant i storlek och att det istället skulle vara möjligt att scrolla igenom stapeldiagrammet. Det bestämdes snabbt att staplarnas storlek ska variera med fönstrets storlek samt att den även ska vara beroende av antalet variabler som ska visualiseras.

När storleken på staplarna bestämts var det nästkommande steget att faktiskt implementera stapeldiagrammet. Detta gjordes på liknande sätt som implementationen för linjegrafen, fast att istället för att rita ut punkter ritas det istället ut staplar som tar in data och varje stapel representerar en variabel, och höjden för varje stapel är värdet av respektive variabel. Värt att notera är att data i linjegrafen och stapeldiagrammet extraherar värden ur samma underliggande datamodell, med

skillnader i representationen, då stapeldiagrammet enbart extraherar de senaste punkterna för respektive variabel.

När stapeldiagrammet var färdigimplementerat insågs det ett möjligt dilemma, vilket var ifall datan som extraheras skulle påverkas av den redan existerande trigger-funktionaliteten, precis som för linjegrafen. Efter en diskussion mellan gruppmedlemmarna bestämdes det att trigger-funktionaliteten inte skulle påverka utritningen av staplarna alls, utan att staplarna alltid uppdateras för att visa de senaste värdena som vid vanlig dataavläsning även om en trigger skulle ske.

3.4.12 Möjligheten att spara och ladda konfigurationsfil

Efter att den endimensionella representationen var implementerad genom stapeldiagram diskuterades det sista steget inom projektet, vilket var möjligheten att spara och ladda konfigurationsfiler. Först bestämdes det vilka delar av applikationen som faktiskt skulle kunna sparas och laddas. Efter en diskussion bestämdes det att de fält som måste fyllas i för att starta inläsningen av data är de fält som bör sparas i konfigurationsfilen.

När gruppen hade bestämt vilka fält som ska sparas i konfigurationsfilen skulle filformatet som konfigurationerna skulle sparas i bestämmas, vilket bestämdes vara JavaScript-objektnotation (JSON) format. Detta gjordes genom att de relevanta fälten fick varsitt nyckel i JSON, som länkas med ett datavärde. När användaren då skulle spara konfigurationen som exempelvis baudrate för UART, hämtas värdet ifrån baudrate-fältet, och den läggs i baudrate-nyckelns dataplats i JSON-filen. Detta upprepas sedan för varenda nyckel som har ett värde. När filen laddas in, hämtas istället värdena från nyckelplatserna och fylls in i de respektive fälten.

Efter att implementationen var färdig, fokuserade gruppen även sitt fokus på att öka säkerheten av applikationen. Detta genom att säkerställa att JSON-filer som har felaktiga datatyper inte kraschar applikationen eller inmatar felaktiga värden. Detta delades upp i tre stycken fall, som skulle kunna vara säkerhetsrisker. Det första fallet är ifall JSON-filen innehåller en nyckel som applikationen inte kan hantera, det andra fallet är vid laddning av JSON-fil ifall en av nycklarna har felaktig datatyp, och det sista fallet är ifall användaren skulle försöka spara en konfigurationsfil med felaktiga datatyper.

Det första fallet löstes genom att laddningen av JSON-filen letar efter specifika nycklar som är förinställda och laddar endast värdena från dessa förinställda nycklar. Andra problemet som handlar om att förinställda nyckeltyper innehåller fel datatyper i JSON-filen, löstes genom att de förinställda nyckeltyperna även ställs in vilken typ som är förväntad att fås in, och sedan när en fil laddas jämförs typen från filen med den förinställda typen; om värdet har felaktig typ ändras värdet till ett standardvärde istället och kastas bort. På liknande sätt löstes även det tredje problemet, där om användaren manuellt försöker redigera ett fält med felaktig datatyp, byts värdet istället ut med ett standardvärde och kastas sedan bort vid inladdning.

3.5 Informationsinsamling

Med tanke på att projektet har varit uppbyggt på ett flertal externa verktyg, teknologier och bibliotek, har projektets gruppmedlemmar behövt införskaffa olika sorters information. Då projektet ansågs vara väldigt praktiskt, baseras rapporten till stor del på officiell dokumentation från de verktyg som gruppen har använt sig av under projektets gång. Därför är majoriteten av källorna av denna karaktär, det vill säga mindre källor som varit väldigt relevanta referenser då de är tagna direkt från förstahandskällor, då inget mer relevant substitut finns. Ett exempel på detta är referenser i koppling till programmeringsspråket C#, då ansågs det väldigt relevant att referera till Microsofts egna dokumentation, istället för en vetenskaplig artikel då C# ägs och utvecklas av Microsoft.

Värt att notera är att vetenskapliga artiklar har använts där det passar. Mer specifikt där området ansågs vara mer teoretiskt. Bland annat de delar som handlat om hårdvaran ansågs vara av en mer teoretisk karaktär, vilket inkluderar inbyggda system, CAN-protokollet och UART. Dessa områden har därför fått en mer formell referenslista, som inkluderade vetenskapliga artiklar, böcker och andra viktiga dokument.

3.6 Användning av AI

AI-verktyg har använts under projektets gång. Specifikt för att underlätta utvecklingen av C# applikationen. Gruppen har inte använt sig direkt utav AI-genererad kod, utan har istället använt AI för att underlätta inlärningsprocessen av C# koncept och syntax, samt för sökningen av potentiella kodbibliotek som kan användas för att underlätta utvecklingen. AI-verktyg som använts är ChatGPT, som enbart användes för ovanstående sökning av information och konceptapplicering i koppling till C#. AI-verktyget användes även för generering av ett par bilder för illustration i den tekniska bakgrunden, fast granskades manuellt och redigerades för att reflektera korrekt information vid behov.

4
Systemkonstruktion

I detta kapitel presenteras den tekniska konstruktionen av det utvecklade systemet, med fokus på både relevanta mjukvarukomponenter. Arbetet är uppdelat i två centrala delar. Den första delen av arbetet innefattar utvecklingen av ett effektivt UART-bibliotek i C för STM32 mikrokontrollers, som möjliggör högpresterande kommunikation till en PC. Den andra delen av arbetet innefattar en plottningsapplikation utvecklad i C#, som har möjlighet att kopplas upp till UART-biblioteket eller existerande CAN-trafik för att avläsa och visualisera data på olika format i realtid.

4.1 UART-bibliotek för STM32

Denna delsektion fokuserar på att ge en detaljerad beskrivning av konstruktionen av det högpresterande UART-bibliotek som utvecklades. Specifikt behandlas designen av datapaketen och hur dataöverföringen går till inom det kommunikationsprotokoll som implementerades. Biblioteket, skrivet i C för STM32-mikrokontrollers, använder headerfilen `stm32f4xx_ll_usart.h`, som tillhandahålls separat av STMicroelectronics under Berkeley Software Distribution (BSD) 3-Clause-licens, för att ge låg-nivå åtkomst till USART-periferienheterna på en STM32F4 mikrokontroller. Denna användes för att etablera en asynkron koppling till en ledig USART-port på STM32F407G-DISC1 utvecklingskortet, vilket användes för utvecklingen.

4.1.1 Initiering av biblioteket

Initiering av biblioteket görs via ett anrop till funktionen *UART_Init*, som tar emot tre parametrar: en pekare till den specifika USART-instansen, en pekare till en 32-bitars tidsstämpel som inkrementeras externt, samt ett val av datastorlek som anger hur mycket av datafältet i varje paket som skall användas. Giltiga datastorlekar är 8, 16 samt 32 bitar vilket definieras av en enum datatyp `UART_PayloadSize`. Vid anrop av funktionen sparas dessa parametrar internt för enkel åtkomst. Vidare nollställs även två indexpekare *uartTxHead* och *uartTxTail*, som representerar skriv- och läspositionen för nästa datapaket i den underliggande cirkulära bufferten som används, vars storlek ställs in i bibliotekets headerfil `uart.h`. Slutligen ställs den givna USART-instansen in till en datastorlek på 8 bitar, och paritetbitar stängs av, varefter USART-instansen aktiveras.

4.1.2 Allokerad storlek av datapaket

Implementationen av nedanstående datamodell utgår från att ett datapaket, benämnt `UART_TimestampedData`, är 40 bitar långt, varav 32 bitar allokeras för ett datafält. Kvarstående 8 bitar av datapaketet representerar en tidsstämpel som förväntas avläsas tillsammans med lagringen av ett datapaket som skall skickas. Då den del av tidsstämpeln som kan representeras i ett datapaket enbart är 8 bitar utav de 32 som tidsstämpeln maximalt kan representera, förväntas detta leda till frekventa överflöden. Exempelvis sker ett överflöde på en period av 2,56 sekunder vid en uppdateringsfrekvens på 100 Hz, och behöver därmed hanteras på mottagarapplikationens sida. Designvalet är ett resultat av behovet att minimera det antal bitar som behöver skickas för att representera ett tidsstämplat datapaket, för att maximera den mängd data som kan överföras inom en viss tidsperiod.

```
typedef struct {
  uint32_t data;        //32 bitars data
  uint8_t timestamp;    //8  bitars timestamp
}UART_TimestampedData;
```

4.1.3 Paketering och tidsstämpling av samplad data

Funktionen *UART_StoreData* ansvarar för att avläsa, tidsstämpla och lagra datavärden i en intern cirkulär buffert för en mikrokontroller, innan de kan överföras till en PC. Funktionen tar som argument en lista av pekare till `uint32_t`-värden samt ett numeriskt värde `n`, som anger antalet element i listan.

Om biblioteket ej har initierats, eller det inte finns tillräckligt många lediga platser i den lokala bufferten för att avläsa och lagra ett datavärde för varje pekare som avläses, ignoreras denna operation och funktionen returnerar det numeriska värdet 0. Om detta ej är fallet, skapas ett paket genom att först allokera en pekare till nästa lediga plats i bufferten, som pekar på en datatyp av typen `UART_TimestampedData`. Därefter kopieras datafältet från den datapekare som avläses till paketets *data*-fält, där endast de lägre 8, 16 eller 32 bitarna används beroende på den payloadstorlek som angavs vid initiering av biblioteket. Likvärdigt maskeras och kopieras de 8 minst signifikanta bitarna av tidsstämpeln som angavs vid initiering till datapaketets *timestamp*-fält. Efter detta flyttas *uartTxHead* fram ett steg med hjälp av en define-macro som ger tillbaka nästa position i bufferten, vilket tillåter att denna process utförs återigen för resterande pekare som inmatades till funktionen vid anrop. Efter att alla paket har skapats returnerar funktionen det numeriska värdet 1. Den centrala loopen i funktionen som utför denna packetering ser ut som i följande kodblock:

```
// Loop through each pointer in dataArray to create packets.
for(size_t i = 0; i < n; i++) {
  // Have dataToAdd point to the next free buffer slot
  UART_TimestampedData *dataToAdd = &uartTxBuffer[uartTxHead];
  uint32_t dataValue = *dataArray[i];

  // Mask the data according to selected payload size
  switch (uartPayloadSize) {
    case UART_PAYLOAD_8:
      dataToAdd->data = dataValue & 0xFF;
      break;
    case UART_PAYLOAD_16:
      dataToAdd->data = dataValue & 0xFFFF;
      break;
    default: // UART_PAYLOAD_32, don't mask.
      dataToAdd->data = dataValue;
      break;
  }

  // Use 8 bits of a provided "timestamp variable".
  dataToAdd->timestamp = (uint8_t)(*timeValue & 0xFF);

  // Data added, move head forward:
  uartTxHead = BUFFER_NEXT(uartTxHead);
}
```

4.1.4 Sändning av data från bufferten

För att överföra packeterad och tidsstämplad data från mikrokontrollern till en PC används funktionerna *UART_FlushOne* och *UART_FlushBuffer*. Dessa ansvarar för att extrahera datapaketen från den cirkulära bufferten och skicka dessa via mikrokontrollerns UART-gränssnitt. Dessa två funktioner utför samma funktion, med den distinkta skillnaden att *UART_FlushOne* skickar ett datapaket, medan *UART_FlushBuffer* genom en while-loop skickar alla. Värt att notera är att den sistnämnda ej kallar på den förstnämnda i sin loop, för att undvika den extra tid som används för funktionsanrop då dessa funktioner utförs på en väldigt låg nivå.

Funktionerna kollar först om data existerar genom att jämföra *uartTxHead* och *uartTxTail*. Om data finns tillgänglig, hämtas det datapaket som finns på nuvarande *uartTxTail*-index för att sedan skickas över UART. Processen går till sådan att en lista av de bytes som skall skickas skapas och fylls på utifrån det datafält och den tidsstämpel som datapaketet innefattar. Längden på denna lista är ekvivalent till den datastorlek som ställdes in vid initiering, plus en byte för tidsstämpel. Detta resulterar till att biblioteket vid transmission av data enbart skickar de bytes som krävs för att representera det maskerade värdet i det 32-bitars datafält som används för intern minnesallokering. En loop går sedan igenom denna lista för att skicka

varje enskild byte över UART:s dataregister. Inför varje sändning inväntas det att `Transmit Data Register Empty (TXE)`-flaggan indikerar att registret är redo att skicka nästa byte. En timeout på 5000 loop cyklar används även för att se till att denna loop inte fastnar i väntan på TXE-flaggan om något skulle gå fel. Värt att notera är att den ordningen som bytes i ett datapaket skickas i är `big-endian`-ordning, vilket menar på att den mest signifikanta byten i datafältet skickas först, som kan ses i följande kod, vartefter respektive byte i *bytesToTransmit*-listan skickas från lågt till högt index:

```
// Get the next packet in line to send
UART_TimestampedData packet = uartTxBuffer[uartTxTail];

// 1, 2 or 4 bytes of data + 1 byte timestamp.
uint8_t bytesToTransmit[uartPayloadSize + 1];
uint8_t byteCount;

// Pack data to send based on payload size
switch(uartPayloadSize){
  case UART_PAYLOAD_8:
    bytesToTransmit[0] = packet.data & 0xFF;
    byteCount = 1;
    break;
  case UART_PAYLOAD_16:
    bytesToTransmit[0] = (packet.data >> 8) & 0xFF;
    bytesToTransmit[1] = packet.data & 0xFF;
    byteCount = 2;
    break;
  default: // UART_PAYLOAD_32
    bytesToTransmit[0] = (packet.data >> 24) & 0xFF;
    bytesToTransmit[1] = (packet.data >> 16) & 0xFF;
    bytesToTransmit[2] = (packet.data >> 8) & 0xFF;
    bytesToTransmit[3] = packet.data & 0xFF;
    byteCount = 4;
    break;
}

// Add the least significant byte of the timestamp
bytesToTransmit[byteCount++] = packet.timestamp & 0xFF;
```

4.1.5 Aktivering och avbrytning av samplingslogik

En funktion *UART_ProcessCommand* ansvarar för att tolka inkommande byte-meddelanden från en PC och fungerar som ett gränssnitt för att kontrollera bibliotekets möjlighet till att delvis sampla- men även skicka data. Funktionen inspekterar `Receive Data Register Not Empty` flaggan för USART-instansen, som indikerar att en byte har mottagits av mikrokontrollern via UART-kopplingen. Om detta är fallet, hämtas denna byte med hjälp av en LL-metod för att sedan inspekteras. Om byte-värdet representerar tecknet `'S'`, sätts en flagga *uartTransmitEnabled* i kodbiblioteket till värdet 1, för att indikera att biblioteket nu får sampla och skicka data. I fallet att byte-värdet istället representerar tecknet `'R'`, sätts denna flagga, *uartTxHead* och *uartTxTail* till värdet noll för att avsluta datakommunikationen och nollställa bufferten. Övriga värden på en byte ignoreras.

Denna funktion kallas på i början av alla tidigare nämnda bibliotek-funktioner, och används för att minimera risken att en PC kopplar in sig till en pågående dataström och återskapar inkorrekta värden. Med ovanstående implementation, krävs det att en PC först kopplar upp sig till den mikrokontroller som skall analyseras, för att sedan skicka en `'S'`-byte som indikerar att PC-applikationen är redo att ta emot det första datapaketet. Vidare krävs det enbart att en `'R'`-byte skickas, och sedan en ny `'S'`-byte igen för att starta om transmissionen om avläsningsfel uppstår.

4.2 Applikationens Datamodeller

C# applikationens datamodeller ansvarar för att hantera och lagra data i ett strukturerat format enligt MVVM-arkitekturen. Dessa modeller kapslar in data och tillhörande logik för manipulering av denna, utan att påverkas av hur applikationen väljer att interagera med datamodellerna. Varje modell finns i `Models`-mappen som en C# fil, och definierar en egen form av data med ett enskilt och tydligt syfte för att separera datahanteringen med övriga delar av systemet.

De olika modellerna är designade för att på egen hand utföra den logik som krävs för att utföra sin uppgift. I denna sektionen beskrivs de centrala datamodellerna i applikationen samt deras syften, utan att framföra hur dessa brukas inom applikationens resterande klasser.

4.2.1 Lagring av grafdata

Lagring av grafdata hanteras utav en klass vid namn `GraphDataModel`, som innehåller en privat lista för tidsstämplar på X-axeln, och en för värden på Y-axeln. Vidare innehåller den även variabler som används för detektion av överflöden, för att dynamiskt återhämta från dessa sådan att modellen alltid representerar en ritbar graf.

Klassen exponerar två offentliga metoder; *AddPoint* och *Clear*. Den första metoden tar osignerade heltal x och y som argument, och lägger till dessa värden i respektive listor. Innan datapunkterna läggs in i listorna jämförs det givna x-värdet med det tidigare värdet som lades in i listan. Om fallet är att det tidigare värdet är större, måste det vara så att ett överflöde har skett. Överflöden är något som

förväntas ske då data läses in via UART, då antalet bytes för tidsstämpeln minimeras för att tillåta en större mängd data att överföras på en given baud rate. Detta leder till mer frekventa överflöden, då färre bitar skickas med för att representera sådan data. För att motverka att den felaktiga data som representerar tidsstämplar läggs in, adderas det värde som överflödet beräknas ha skett vid, som senare kan adderas på alla nykomna värden:

```
if (x < _lastRawTimestamp){
    // Add the value lost during overflow in received data
    _overflowAdd += (uint)Math.Pow(2,_xValBitSize);
}
_lastRawTimestamp = x;
uint adjustedX = x + _overflowAdd;
_XData.Add(adjustedX);
_YData.Add(y);
```

Den andra metoden, *Clear*, tar bort allt innehåll i listorna, samt nollställer alla numeriska variabler som används för att hantera överflöden av inkommande tidsstämplar.

Denna klassen innehåller enbart en lista för X- respektive Y-axeln, även fast den i projektet har möjlighet att innefatta ett flertal variabler. Detta är ett designval som förenklar åtkomsten till all historisk data på en och samma plats. För att möjliggöra lagring av flera variabler används dessa listor i applikationen med interfoliering, sådant att en modulo-jämförelse kan filtera bort alla irrelevanta index för en given variabel. Exempelvis ger `index % numberOfVariables == 0` sant enbart för de datapunkter som interfolieras med variabeln vars första värde är placerat på index 0.

4.2.2 Definition av UART-datapaketstorlek

Vid initiering av UART krävs det att användaren markerar datastorleken för datapaketen som förväntas anlända, så att det är möjligt att räkna ut när ett komplett datapaket med tillhörande tidsstämpel har anlänt. En uppräkningstyp vid namn `UARTDataPayloadSize` har skapats för detta ändamålet, och ser ut som nedan:

```
public enum UARTDataPayloadSize{
  UART_PAYLOAD_8,   //  8-bit data
  UART_PAYLOAD_16,  // 16-bit data
  UART_PAYLOAD_32,  // 32-bit data
}
```

Denna används för att på ett strukturerat sätt definiera vilka datastorlekar som får skickas in till initieringsmetoden för en UART-klass, då detta möjliggör enkel felhantering och strikta alternativ som hårdvarubiblioteket som applikationen skapats i hänsyn till kan hantera.

4.2.3 Tidsstämplad UART-data

`UARTTimestampedData` är en datastruktur som har som syfte att representera ett komplett rekonstruerat paket bestående av data och tidsstämpel som mottagits av en seriell buss. Den innehåller ett 32-bitars datavärde då detta är den maximala storleken som förväntas tas emot, samt en 8-bitars tidsstämpel. Modellen används huvudsakligen för att abstrahera mottagna bytes som läsbara variabler, så att dessa kan skickas som event och enkelt hanteras av mottagaren.

4.2.4 Triggerlägen

`TriggerMode` är en datamodell som definierar de triggerlägen som applikationen stödjer och kan använda sig utav, och ser ut som nedan:

```
public enum TriggerMode{
  Single_Trigger ,
  Normal_Trigger ,
}
```

Denna används för att förhindra beroende på strängbaserade inmatningar av triggerlägen, då detta kan resultera i att buggar uppstår till följd av felskrivningar, eller att icke-hanterade värden matas in.

4.2.5 Modell för variabelhantering

I applikationen används ett interface `IVariableModel` som vidare implementerar ett interface `INotifyPropertyChanged`, vilket möjliggör att ViewModels som har en instans av IVariableModel kan reagera på förändringar av dess egenskaper. IVariableModel innehåller en sträng som representerar ett namn av en variabel, samt en boolsk variabel som indikerar om denna variabel för tillfället skall vara synlig på grafen eller ej.

En konkret implementation av IVariableModel är klassen `VariableModel`, som med sin implementation hanterar egenskaperna ovan med hjälp av ReactiveUI:s *RaiseAndSetIfChanged* metoder för att automatiskt notifiera applikationens UI om någon ändring har skett. Detta möjliggör en objektorienterad enkapsulering av de egenskaper som behöver synkroniseras mellan modellen och användargränssnittet i applikationen. Resultatet är att det dynamiskt går att ändra namn på de variabler som ritas ut på grafen, samt växla deras synlighet medan utritning är pågående. Då implementationen följer alla SOLID-principer, är det även enkelt att byta ut eller utöka implementationen sådan att dessa egenskaper fungerar annorlunda vid behov.

4.3 Applikationens Kommunikationstjänster

För att möjliggöra mottagning av data från externa enheter hanterar PC-applikationen kommunikation via CAN och UART genom isolerade tjänster. Dessa tjänster är helt frånkopplade från applikationens övriga logik och ansvarar enbart för att möjliggöra plattformsoberoende dataöverföring.

Kommunikationstjänsterna är organiserade i `Services`-mappen och innehåller abstraherade implementationer av CAN och UART tjänsterna för att implementeras i enlighet med SOLID-principerna. I denna sektion beskrivs hur dessa tjänster är implementerade, utan koppling till resterande delar av applikationen.

4.3.1 Mottagning av UART-data i applikationen

För att göra mottagningen av inkommande UART-data mer beroende av abstraktioner finns ett gränssnitt `ISerialReader`, som används för att hantera seriell kommunikation. Det definierar funktionalitet för att starta och stoppa kommunikationen samt hantera inkommande tidsstämplad data. I detta interface finns två stycken metoder, *StartSerial* som används för att påbörja den seriella kommunikationen. Detta görs genom att skicka in en sträng som innehåller den port som användaren vill kommunicera med, samt även den baudrate som kommunikationen förväntas ske på och till sist även hur stora datapaketen ska vara. Utöver *StartSerial* existerar även metoden *StopSerial* som används för att avsluta en pågående asynkron läsprocess, denna metoden behöver därför inga inputs. För att möjliggöra hantering av inkommande data innehåller interfacet även en händelse, `TimestampedDataReceived` som används för att likt ett Observer-pattern tillåta prenumerationer till data som eventet anropas med i klassen.

Klassen `UARTSerialReader` implementerar interfacet `ISerialReader` och tvingas därför att implementera de relevanta metoderna. Metoden *StartSerial* är implementerad genom att den öppnar en seriell anslutning på den angivna COM-porten med en specificerad baudrate och datastorlek. Om anslutningen redan är aktiv returnerar metoden direkt utan att den begår några uträkningar eller öppnande av portar. Dessutom för att säkerställa att anslutningen avslutas vid en tvångsavslutning såsom att stänga ner applikationfönstret, registreras en metod *ForceStopCleanup* som automatiskt stänger porten när applikationen avslutas. Den skickar även ett slutkommando `'R'` för att avsluta mikrokontrollerns sändning av data, och för att nollställa dess buffert.

Ett argument till *StartSerial* benämner även hur många bytes som varje datapaket förväntas innehålla. Alternativen för detta är 8, 16 eller 32 bitar, vilket bestäms via ett switch case som tar in en instans av `UARTDataPayloadSize`. Därefter skapas en SerialPort-instans med specificerade buffertstorlekar och timeout-inställningar. På Linux utförs en extra konfiguration där baudraten först sätts till ett 'dummyvärde' för att undvika en problematik med hanteringen av seriella portar på Linux operativsystem som kan hindra rätt baudrate från att ställas in.

När porten sedan har öppnats, allokeras en buffert som håller på inkommande data, samt att en asynkron loop startas genom en privat hjälpfunktion. Denna hjälpfunktion, *StartReadingLoop*, baserar sig på `BaseStream` objektet som är under-

liggande till C#:s SerialPort bibliotek som används för seriell kommunikation. Detta objekt är platformsoberoende och kan användas för avläsning av både Linux och Windows seriella portar. Slutligen skickas ett startkommando `'S'` seriellt via den seriella porten, för att signalera till den uppkopplade mikrokontrollern att transmission av data kan påbörjas.

Den andra metoden som `UARTSerialReader` implementerar är metoden *StopSerial* som avslutar den seriella kommunikationen mellan applikation och mikrokontrollern. Detta görs genom att först kontrollera om en nuvarande läsningsprocess är pågående. Om ingen aktiv läsning är igång, returnerar metoden direkt utan vidare handling. Detta görs genom att kolla ifall om den lokala boolska variablen *isReading* är sann. Om porten är öppen skickar applikationen istället ett stoppkommando `'R'` till den anslutna enheten för att signalera att datamottagningen ska upphöra. Därefter markeras läsningen som inaktiv och läsningsprocessen avslutas, vilket sker genom att den boolska variabeln *isReading* sätts till falskt. Slutligen stängs även applikationens koppling till den seriella porten.

4.3.2 Mottagning av CAN-data i applikationen

CAN-biblioteket är en av applikationens kommunikationstjänster som är utformad för att hantera kommunikation via CAN-bussen, med ett primärt fokus på att högpresterande mottagning av data. Data som läses av från en implementation av en CAN-buss skall även tidsstämplas sådant att generell CAN-data kan användas för plottning oavsett om denna inkluderar en tidsstämpel i sitt datafält.

CAN använder, likt UART, ett interface för att se till sådant att programvaran är beroende på abstraktioner och inte implantationer. Detta interface heter `ICanBus` och används för att abstrahera olika implementationer av CAN-kommunikation över en CAN-buss. För att möjliggöra detta innehåller interfacet ett par metoder, som med ISP i åtanke anses vara relevanta för alla möjliga sorters implementationer av ett CAN-kommunikationsprotokoll. Den första metoden är *Connect* som tar in två stycken strängar, där den första strängen representerar vilken virtuell eller fysisk port som kommer att användas, och den andra är frivillig och representerar vilken bitrate som ska användas, det vill säga vilken överföringshastighet. Den andra metoden heter *Disconnect* och den tar inte in några värden, utan dess funktion är att avsluta kopplingen mellan applikation och hårdvara när anslutningen inte längre behövs.

SendMessage som tar in två parametrar, en uint som representerar ett CAN-ID, och även en byte-array som innehåller datan som ska överföras. Metoden ska därefter returnera mängden bitar som skickats. Precis som vid UART finns det även en händelse i interfacet. Denna händelse heter `MessageReceived` och den fungerar som ett observer pattern där användaren får ett meddelande vid anropning, det vill säga om det skickats något från CAN-bussen.

För att kunna hantera olika implementationer till olika operativsystem konstruerades en statisk klass vid namn `ControllerAreaNetwork`. Denna klass skapar instanser av CAN-busskommunikation, där den väljer automatiskt den mest lämpliga implementationen baserat på vilket operativsystem applikationen exekveras på, såsom exempelvis `SocketCanBus` implementationen för Linux system och `PeakCanBus` implementationen för Windows system.

4.3.2.1 SocketCanBus

`SocketCanBus` är en implementation av `ICanBus`-interfacet som används för att hantera kommunikation med en CAN-buss på Linux system. Den bygger på Linux-kärnans inbyggda CAN-drivrutin, och använder sig utav SocketCAN-Sharp biblioteket som finns tillgängligt i pakethanteraren NuGet för att interagera med denna drivrutin i koden. Vid initiering av klassen skapas de instansvariabler och objekt som är nödvändiga för effektiv meddelandehantering. Viktigast skapas en `RawCanSocket` för koppling till en CAN-port, samt ett antal bakgrundstrådar som antingen lyssnar på inkommande data, eller hanterar mottagen data. En trådsäker kö skapas även för att hålla på mottagna meddelanden innan de hanteras, för att reducera risken att meddelanden inte kan hanteras i tid. För att optimera skapandet av byte-arrays vid mottagande av data används även en variant av trådpool-designmönstret, där det istället skapas ett bestämt antal byte-arrays i en trådsäker kö. Dessa kan mottagartråden använda sig utav för att drastiskt minimera antalet objekt som skapas.

Vid anslutning till en `SocketCanBus` används argumentet *interfaceName* för att identifiera den CAN-port som användaren matar in i applikationens UI. Den frivilliga bitrate strängen används ej för denna implementationen, då bitrate för den eftertraktade porten ställs in direkt i Linux-kärnans SocketCAN drivrutin i en terminal, exempelvis som följande:

```
sudo modprobe <driver_name>
sudo ip link set <port_name> type can bitrate <bitrate>
sudo ip link set <port_name> up
```

Bakgrundstrådarna för avläsning och meddelandehantering startas tillsammans med anslutningen, samt även en *Stopwatch* som används för tidstämpling av den tråd som hanterar mottagna meddelanden och anropar att ny data har anlänt på klassens `MessageReceived` händelse.

Den första tråden som hanterar avläsning av meddelanden gör detta via en metod ReceiveMessages() som, så länge en flagga _running är sann, läser av CAN-trafik. Detta görs genom ett blockerande anrop som väntar på anländande data. När data har anlänt försöker metoden att ta en byte-array från den trådsäkra kön, som data kan kopieras till och sedan läggas över till den andra trådsäkra kön som håller i mottagna meddelanden. När `SocketCanBus` klassen kopplas ifrån blir _running falsk vilket tillåter att tråden termineras.

Den andra tråden som hanterar dem mottagna meddelandena gör detta via en metod *ProcessMessages* som likt den tidigare enbart är aktiv så länge en flagga _running är sann. Den fungerar sådan att så länge det finns något i den trådsäkra kön av mottagna meddelanden, extraherar den meddelanden därifrån, skapar en tidsstämpel via *Stopwatch* objektet, och anropar `MessageReceived` händelsen med CAN-meddelandets id, data-array, och tidsstämpeln. Slutligen lämnas byte-arrayen tillbaks till byte-array poolen, och hanterar nästa meddelande.

4.3.2.2 PeakCanBus

`PeakCanBus`, likt `SocketCanBus`, är en implementation av `ICanBus`-interfacet, med avseende att användas inom en Windows-miljö med CAN-adapters som använder sig utav PEAK-Systems drivrutiner. Denna klass är implementerad med samma logik som SocketCanBus, då den använder sig av två stycken bakgrundstrådar för inläsning samt hantering av meddelanden, en Stopwatch för tidsstämpling, och trådsäkra köer för buffring av meddelanden och förallokerade byte-arrays.

Det som skiljer de två klasserna är det underliggande bibliotek som användes för kopplingen till CAN-bussen. I fallet av `PeakCanBus` klassen, användes ej SocketCan-Sharp då det är exklusivt till Linux, utan PCANBasic.NET som finns tillgängligt i pakethanteraren NuGet. Även då båda bibliotek uppfyller liknande funktionalitet, leder detta till skillnader vid initiering. För initiering av `PeakCanBus`, förväntas en bitrate sträng, då denna krävs för att initiera PCANBasic med rätt överförings-hastighet. Det vill säga att programmet på Windows låter användaren mata in en överföringshastighet utifrån ett förutbestämt antal alternativ vid konfigurering av kopplingen, medan användaren på Linux hänvisas till att ställa in denna genom SocketCAN. Skulle detta inte göras, kastas ett undantagsfall som resterande delar av applikationen kan ta emot för att hantera eller visa som felmeddelande i diverse statusfält.

4.4 Applikationens övriga tjänster

Utöver kommunikationstjänster innefattar projektet ett flertal tjänster för att separera funktionalitet till separata moduler, vilka kan användas vid behov för att komma åt specifik funktionalitet inom applikationen. Utöver kommunikationstjänster hanteras även följande som egna tjänster:

- **Konfigurationshantering** - Hanterar skrivning och inläsning av konfigurationsfiler.
- **Konfigurationsavläsning** - Hanterar avläsning av konfigurationer för kommunikationskanaler och utvinner konfigurationsvärden.
- **Datakanaler** - Hanterar abstraktion och skapandet av gemensamma gränssnitt för anslutning av kommunikationskanaler.
- **Linjegraf** - Hanterar utvinning av presenterbar data, triggerhantering, samt presentationslogik av en linjegraf.
- **Stapeldiagram** - Hanterar utvinning av presenterbar data samt presentationslogik för ett stapeldiagram.

4.4.1 Konfigurationshantering

För att möjliggöra sparande och laddande av konfigurationsfiler för applikationen implementerades ett konfigurationshanteringssystem i form av en tjänst, som finns i projektet under `Services/ConfigHandler`. Det består av ett gränssnitt samt en implementation av dess API.

`IConfigService` är det gränssnitt som definierar API:n för konfigurationshanteringen. Det innehåller en metod *AddToConfig* för att lägga till konfigurationspara-

metrar, som sedan kan sparas till en fil via ett anrop av *ExportConfig*. Vidare finns även en metod *LoadConfig*, som tar in den datatyp som förväntas läsas av, samt ett nyckelvärde för att identifiera det värde som eftertraktas. En *FilePath* exponeras även, som måste ställas in för att kunna kalla på *ExportConfig* och *LoadConfig*.

En konkret implementation av ovanstående gränssnitt, `ConfigService`, tillämpar Singleton-mönstret för att säkerställa att enbart en instans av konfigurationshanteraren existerar under programmets livslängd. Detta är ett vanligt mönster som i detta fallet simplifierar processen att spara konfigurationsvariabler som är utspridda i hela projektet till en och samma konfigurationsfil. Klassen hanterar lagring av konfigurationen i ett JsonArray objekt, som finns i .NET:s standardbibliotek. Det tillåter att flexibelt representera data som en koppling mellan nyckelord till dess värden, vilket även är hur exporterad data representeras i sin sparfil.

4.4.2 Konfigurationsavläsning

För att ansluta till kommunikationsgränssnitt som UART och CAN, behövs ett antal parametrar för dess konfigurering. Dessa konfigurationer skickas mellan olika viewmodel klasser som textsträngar, och behöver utifrån dessa konverteras tillbaka till värden av rätt datatyp. Detta hanteras utav klasserna under `Services/ConfigParsers` i projektet. Dessa tjänsterna och dess API definieras under ett gemensamt gränssnitt `IConfigParser`, som exponerar metoderna *ParseUartConfig*, *ParseCanConfig* samt *ParseCanDataMask*.

En konkret implementation av detta gränssnitt är `ConfigParser` klassen, som utför avläsning av UART och CAN konfiguration med hjälp av reguljära uttryck för att validera formatet av den sträng som inmatas. Vid lyckad avläsning av datan, returneras de värden som textsträngen innefattar. Vid misslyckande returneras standardvärden istället.

Klassens implementation av *ParseCanDataMask* bygger på att den tar emot en maskeringssträng i formatet `"__:__:__:__:__:__:__:__"` tillsammans med bytearray som representerar den data som mottagits via CAN-protokollet. Varje segment i maskeringssträngen motsvarar en byte i den åtta byte stora datalasten. Inom varje segment kan en eller flera variabelidentiferare anges, exempelvis `"11"` för att markera att hela byten tillhör variabel 1, eller `"1_"` respektive `"_1"` för att markera att den övre undre halvan av byten tillhör variabel 1. Skulle ett segment lämnas blankt, ignoreras denna byte. Metoden separerar maskeringssträngen till 8 olika sektioner, och itererar över varje position i masken och återskapar de variabler som identifieras med användning av bitoperationer, fram tills dess att nästa variabel ej har definierats. Detta möjliggör väldigt hög flexibilitet för användaren vad gäller hur datavariabler ska tolkas med avseende på den datalasten som avläses.

4.4.3 Datakanaler

För att förenkla användandet av kommunikationstjänster i applikationen har dessa abstraherats ytterligare under ett gemensamt gränssnitt `IDataChannel` under `Services/DataChannels`. Detta gränssnitt exponerar de två metoderna *Connect* och *Disconnect*, vilka ansvarar för att etablera en anslutning samt avveckla denna

för en underliggande datakälla. Denna abstraktion existerar sådan att olika kommunikationstjänster kan användas utbytbart i applikationen utan behov av att tänka på vad för kommunikationskanal som är kopplad.

En konkret implementation är `UartDataChannel`, som hanterar den seriella kommunikationen via UART. I dess konstruktor injiceras en `ISerialReader`-instans tillsammans med de parametrar som krävs för att ansluta till den seriella porten, samt en referens till den `GraphDataModel` där inkommande datapunkter skall sparas. Här registreras även en metod *OnUartDataReceived*, som åkallas då ett nytt datapaket tas emot, och som omvandlar detta till presenterbar data i `GraphDataModel`-objektet. När *Connect* anropas, skapas en koppling till den seriella porten, och avläsning påbörjas. Vid anrop av *Disconnect*, avslutas kopplingen, och *OnUartDataReceived* avregistreras från den händelsen som åkallas vid mottagning av datapaket.

Likvärdigt har en klass `CanDataChannel` implementerats som istället tar emot en `ICanBus`-instans, tillsammans med de parametrar som krävs för att ansluta till CAN-trafiken. *Connect* och *Disconnect* fungerar för CAN precis som för UART. Registrering av en metod *OnCanDataReceived* sker även i konstruktorn, där CAN-trafik filtreras efter det eftersökta CAN ID nummret, samt omvandlas till enskilda variabelvärden enligt datamaskeringen via en `ConfigParser`.

Slutligen finns klassen `DataGeneratorChannel` för att generera slumpgenererad testdata utan fysisk hårdvara med hjälp av en klass `DataGenerator`. Denna fungerar likvärdigt de tidigare klasserna, och används enbart för att testa applikationens övriga funktioner under utvecklingsprocessen vid tillfällen då hårdvara ej är tillgänglig.

För att utvidga applikationen för stöd av fler kommunikationskanaler behöver enbart en ny *IDataChannel* implementation skapas för dessa kommunikationskanaler.

4.4.4 Tjänster för hantering av linjegrafer

För effektiv och flexibel realtidsvisualisering av inkommande data som en linjegraf används ett antal separata tjänster under `Services/Plotting/LineGraph`. Dessa tjänster ansvarar för datautvinning, triggerfunktionalitet, samt grafisk presentation av data.

Ett flertal gränssnitt har definierats för dessa tjänster, nämligen `IGraphDataService`, `ITriggerService` samt `IPlotUiService`. För respektive gränssnitt medföljer en konkret implementation, `GraphDataService`, `TriggerService` och `PlotUiService`.

Gränssnittet `IGraphDataService` och den medföljande konkreta implementationen `GraphDataService` definierar den logik som avgör vilken delmängd av den totala historiska data som skall utvinnas för senare hantering från andra tjänster. Implementationen hanterar buffring av X- och Y-data beroende på delvis fönsterstorlek, triggertillstånd samt om *History mode* har aktiverats, i vilket fall all historisk data extraheras. Om *History mode* ej är aktiverat, extraheras den senast mottagna delmängd data som behövs för att fylla ut grafens fönster enligt den inställda fönsterstorleken. Om en *normal trigger* har aktiverats, centreras denna delmängd på senaste triggerpunkt i stället. Hantering för dubletter på tidsaxeln hanteras även, i de fall då upplösningen på en tidsstämpel ej är hög nog för att ge ett unikt värde för varje enskild datapunkt för en variabel. Detta görs genom att hitta den tidigaste framkomsten av en tidsstämpel genom binärsökning, för att markera denna punkt

som startpunkten för den exekverade delmängdens omfång. *GetSubData* är den metod som utför ovanstående logik, för att slutligen returnera dels de data som ska visas, men även via nedanstående tjänst ett triggerindex relaterat till delmängden som, om positivt, representerar den triggerpunkt som existerar i den extraherade delmängden av data.

Logiken för triggers hanteras av `ITriggerService` och den tillhörande implementationen `TriggerService`, som håller reda på om och när en trigger senast inträffade, om triggers har aktiverats av användaren. Vid aktivering eller flyttning av en triggernivå, sparas indexet för senast avlästa datapunkten, som därefter blir startpunkten för sökningen av framtida triggerpunkter. Vid sökning av nya triggerpunkter letas det bara efter de som tillhör en variabel vars *isTriggerable*-fält har aktiverats av användaren i *Plot Config* fönstret på applikationens UI. En positiv flank avläses genom att förgående datapunkt till den som inspekteras är under, medan den aktuella punkten ligger över den satta triggernivån - samt att värdet mellan de två punkterna faktiskt ökar. Vid inträffande av en trigger sparas dess globala index, relativt till den fullständiga historiska listan i `GraphDataModel` som kan användas för att i konstant tidsexekvering hitta det intervall som senaste triggern orsakades på relativt till alla historiska datapunkter. Vid en *single trigger* avläses enbart en triggerpunkt, för att sedan avsluta kopplingen till den aktiva kommunikationskanalen efter två sekunder för att tillåta en liten delmängd data att läsas in efter triggerpunkten. *History Mode* aktiveras efter dessa två sekunder, vilket tillåter användaren att fritt inspektera all historisk data upp till- och kort efter triggerpunkten. I *normal trigger*-läge avslutas ej kopplingen, utan istället återställs vid varje triggerhändelse startindexet för att leta efter nästa trigger.

Slutligen ansvarar `IPlotUiService` och den tillhörande implementationen `PlotUiService` för visuell presentation av en linjegraf. Den innefattar en referens till en samling av `IVariableModels` som styr variabellinjernas namn och synlighet, samt en sättningsfunktion som används för att tilldela en unik ScottPlot graf för linjegrafen. Gränssnittet definierar även en metod *UpdateGraphUI*, som uppdaterar samtliga variablers linjer, eventuella triggermarkörer på skärpunkten av en positiv flank, och legendinformation utifrån den extraherade data och de triggerindex som utvinns av ovanstående två tjänster. Gränssnittet definierar även den metod som används för att rita ut den horisontella triggernivån och justera graffönstret.

4.4.5 Tjänster för hantering av stapeldiagram

För att visualisera de senaste värdena för varje variabel som ett stapeldiagram implementerades tjänster för detta under `Services/Plotting/BlockDiagram`. Ett gränssnitt `IBlockDataService` definierar huvudsakligen en metod *ExtractVariableValues*. Den konkreta implementationen av gränssnittet, `BlockDataService`, injiceras med en referens till en `GraphDataModel`, och en inställning för antalet unika variabler. Implementeringen av *ExtractVariableValues* utgår från att iterera baklänges i `GraphDataModel`s datalistor. Jämförelse med varje variabelindex, beräknat `index % numberOfVariables`, görs för att finna det senaste värdet som tillhör den eftertraktade variabeln, som sedan sparas i resultatlistan. Denna process görs utefter variablernas numrering, och repeteras fram tills dess att alla variablers senaste värde

har hittats, vid vilket resultatet returneras.

Gränssnittet `IBlockUiService` definierar de metoder som ansvarar för den faktiska ritningen av stapeldiagrammet i en ScottPlot graf. Grafen ställs in via en av klassens sättningsfunktioner, och den graf som används är skiljt från den som används för linjegrafen. Implementationen i `BlockUiService` innehåller en samling av `IVariableModel`s som styr staplarnas namn och synlighet, likt för linjediagram. Vid anrop till gränssnittets `UpdateBlockUI` anropas en asynkron uppdatering på Avalonia:s UI-tråd, där alla tidigare staplar först rensas, sedan skapas på nytt med de värden som inmatas i metoden efter att ha extraherats via `IBlockDataService`. En intern metod sätter dynamiskt Y-axelns övre gräns till 10% över högsta stapelvärdet som avlästs under nuvarande anslutning, då detta förhindrar konstant justering av Y-axeln vid mycket volatila värden, då detta annars avsevärt försvårar avläsning av stapeldiagrammen.

4.5 Applikationens Användargränssnitt

Applikationens UI är uppbygt av flera vyer, vilka definieras i `View`-mappen och som tillsammans skapar en strukturerad och intuitiv layout för applikationen. Varje vy är skriven i Avalonia XAML, och ansvarar för en specifik del av applikationens funktionalitet. Varje vy definierar sitt eget utseende och innehåll fullt fristående från andra vyer. Dem definierar även databindningar till viewmodel klasser som hanterar anhörig datalogik, samt en *code-behind* fil där interaktionslogik definieras för en vy.

I denna sektion beskrivs de huvudsakliga vyerna som används i applikationen, tillsammans med deras ansvarsområden och hur de ämnar att förbättra användarupplevelsen.

4.5.1 Huvudfönster

Användargränssnitten har en högsta container som innehåller de resterande komponenterna av gränsnittet. Denna container heter MainWindow och använder sig av ett gridsystem i from av rader. Dessa rader består av tre delar: *header*-delen, *main content*-delen och även *footer*-delen. Dessa komponenter är upplagda vertikalt, sådan att *headern* ligger överst i gränssnittet, med *main content* i mitten, och *footern* längst ner. Storleken på *headern* är inställd på auto, med andra ord är den så stor som alla fält i *headern* kräver. Samma gäller för *footern*, som också är inställd på automatiskt storlek, sådant att den blir lika stor som alla komponenter som finns inuti själva *footern*. *Main content* är mer dynamisk, den tar upp den resterande storlek av fönstret som *headern* och *footern* inte tar upp. *Main content* är även en container, och innehåller två stycken andra komponenter i en grid layout beståendes av två stycken kolumner. I den första kolumnen finns Sidebar komponenten, som är inställd till att enbart ta upp den yta den kräver. I den andra kolumnen finns Graph/Diagram komponenten, som tar upp resterande yta för att visa upp diverse grafer.

4.5.2 Statusfält

FooterView komponenten ligger längst ner i applikationen, och är uppdelad i tre sektioner som är uppradade horisontellt bredvid varandra. Den vänstra delen visar en text som visar statusmeddelanden för kommunikationsgränssnitt. Den andra delen är en mittensektion som existerar för att få ett flexibelt mellanrum mellan de två andra sektionerna. På den högra sidan finns en rund knapp med en tillåten GitHub-logotyp som kan klickas för att öppna en extern webblänk till applikationens projektfiler. Allt detta är gjort med en gridlayout, som är uppdelad i kolumner, där den första och tredje kolumnen är inställda på automatisk storlek, det vill säga att de kommer ta upp så mycket plats de behöver, och den andra är inställd på dynamisk storlek, vilket uppdateras efter hur stort fönstret är.

4.5.3 Grafvy

Inuti grid-komponenten som vi kallar *Main Content* i MainWindow, finns det två stycken andra komponenter, varav en av dessa komponenter är GraphDiagramView. Det är den delen av gränssnittet som innehåller ScottPlot-grafer. Det är i dessa grafer som all grafisk utritning sker. Komponenten är uppdelad i två stycken delar, vilket representerar två olika sorters grafer. Dessa två delar är designade sådant att deras närvaro i gränssnittet ska kunna växlas och justera sin storlek utefter varandras närvaro. Exempelvis, om en av graferna är inställd att vara igång, kommer den ta upp platsen för hela komponenten; är båda inställda på att vara igång, kommer de ta upp hälften av utrymmet vardera. På dessa grafer kan det även finnas en triggerlinje som användare kan interagera med. För att tillåta att användaren ska kunna använda sig av musdragning för att flytta denna triggerlinjen har en implementation baserad på ScottPlots kodexempel implementerats i vyns *code-behind* fil, sådan att den fungerar för AvaloniaUI. Följaktligen möjliggörs en intuitiv kontroll av triggerfunktionaliteten längs grafens y-axel.

4.5.4 Verktygsfält

Den header-komponenten som ligger längst upp i MainWindow är HeaderView, som ansvarar för organisering av grafiska verktygsfält. Komponenten består främst av ett grid-system, som definierar kolumner för placering av XAML komponenter. Denna brukas sådant att det längst till vänster i headern finns en dockningspanel som innehåller två rullgardinsmenyer. Den första av dessa visar texten *File*, och visar vid ett klick en knapp för *Save Config* och *Load Config*. Dessa existerar för att tillåta en användare att spara den nuvarande programkonfigurationen, sådan att det går att ladda in den automatiskt via en sparfil vid senare tillfälle. Den andra rullgardinsmenyn innehåller texten *View*, och visar vid ett klick ett antal alternativ för att manipulera upplägget av applikationen, som *Toggle Sidebar*, *Toggle Line Graph*, och *Toggle Block Diagram*. Menyn möjliggör med nuvarande konstruktion enbart knappar för att växla synlighet av andra komponenter. Mer specifikt berör detta sidopanelen och gränssnittets två grafer. Konstruktionen är utformad så att användare som anser att sidopanelen eller andra grafiska komponenter blockerar innehållet de fokuserar på, har möjlighet att växla dessa komponenters synlighet.

Detta görs för att optimera användarens visningsområde och underlätta interaktionen med de element som är mest relevanta för den aktuella uppgiften.

4.5.5 Sidopanel

Inuti MainContent komponenten, finns det till vänster en sidopanels komponent. Denna sidopanel har två flikar som kan bytas mellan, vars uppgift är att hålla på inställningar som användaren kan integrera med. I den första fliken, *COM Config*, presenteras användaren för konfigurationsalternativ beroende på vald kommunikationstyp CAN eller UART, som väljs via en nedrullningsbar meny. Om användaren har ställt in kommunikationstypen till CAN visas ett fält där användaren kan mata in vilket CAN-interface man ska använda sig av, vilken bitrate man ska använda, samt ett CAN ID-filter som avgör vilket ID man vill avlyssna. Dessutom finns det en speciell maskerad textsträng för att definiera vilka byte av CAN-data som representerar olika variabler.

Om användaren valt UART i den nedrullningsbara menyn visas istället ett fält som representerar vilken COM-port användaren vill använda sig av, samt vilken baudrate som ska användas under kopplingen, men antal unika variabler samt storleken på varje datapaket i bitar. Längst ner i denna flik finns en knapp där det står 'Connect' på för att antingen ansluta eller koppla från gränssnittet beroende på aktuell status, detta gäller för både CAN och UART.

I den andra fliken, Plot Config", kan användaren ställa in plottens utseende och uppdateringsbeteende. Det finns reglage för att justera hur många datapunkter som ska visas i realtid samt hur ofta uppdatering ska ske i millisekunder. Nedanför visas även en lista med alla tillgängliga variabler som kan plottas på grafen, där varje variabel har ett textfält för namn och en kryssruta för att aktivera eller avaktivera plottningen, samt att redigera variablens namn.

4.6 Applikationens ViewModels

I anknytning till MVVM-arkitekturen fungerar viewmodel som en brygga mellan användargränssnittet och applikationens underliggande datamodeller och tjänster. De definieras i `ViewModels`-mappen som C# filer, och ansvarar för att hantera all logik och datamanipulation som behövs för att på ett smidigt sätt presentera information för gränssnittet, utan att själv innehålla UI-relaterad kod.

Varje viewmodel är knuten till en specifik vy, utan att känna till dennas existens, och exponerar offentliga variabler och metoder för vyn att avläsa via databindningar. Detta tillåter Avalonia gränssnittet att uppdateras dynamiskt då underliggande data ändras, och även att interaktionslogiken mellan olika vyer är separerade från varandra. Alla viewmodels är även subklasser till en klass `ViewModelBase`, som ärver från ReactiveObject, då detta enkelt centraliserar funktionalitet som alla viewmodels använder eller får tillgång till utan ändringar. Vidare skapar denna klass även en gemensam Singleton `IConfigService` klass, som ger åtkomst för alla viewmodels att spara eller ladda parametrar från en och samma gemensamma konfigurationsfil.

Denna sektion beskriver de viewmodels som används inom applikationen, inklusive ansvarsområden, interaktioner mellan vyer, andra viewmodels och datamodeller.

4.6.1 Hantering av statusfält

`FooterViewModel` är den klass som representerar applikationens statusfält och som ansvarar för att presentera aktuell status för kommunikationsgränssnitt såsom CAN och UART. Denna viewmodel exponerar en textsträng CommInterfaceStatus, vilken är databunden till gränssnittets statusfält och som även uppdateras dynamiskt utefter applikationens tillstånd. Via ReactiveUI:s MessageBus klass lyssnar denna klass på strängbaserade meddelanden som publiceras globalt inom applikationen samtidigt som den undviker någon högre nivå av koppling mellan klasserna. Genom detta möjliggörs att andra viewmodel klasser skickar statusinformation som kan plockas upp av FooterViewModel, för att sedan visa errormeddelanden för användaren i statusfältet. Utformningen av denna klass är gjord på ett sådant sätt att det enbart behöver läggas till en ny prenumeration till ett givet MessageBus meddelande som ställer in nya statusmeddelanden vid mottagande för att utveckla dess möjligheter att uppvisa information till användaren.

Vidare innehåller modellen även ett databundet kommando, som exekverar en metod för att öppna projektets GitHub projektsida där projektfilerna finns tillgängliga. Detta kommando kopplas till den knapp som utmärks av en GitHub-logotyp.

4.6.2 Hantering av grafvy

`GraphDiagramViewModel` är den mest centrala viewmodel klassen inom applikationen, och ansvarar för att hantera all logik som har att göra med realtidsuppdatering och presentation av grafdata i form av linjediagram och blockdiagram. Klassen fungerar som en samordnare av ett flertal tjänst-klasser, som används tillsammans via sina gränssnitt för att möjliggöra koordinerad konfigurationshantering, datainsamling, triggerlogik och grafisk presentation. Dessa tjänster kallas via en intern timer som via mottagna meddelanden från andra viewmodels kan justera tidsintervallet för sin åkallan fritt. Den huvudsakliga uppgiften av klassen är att skapa en modulär och reaktiv visualisering genom att ta emot meddelanden från andra viewmodel klasser, och uppdatera relevanta tjänster beroende på den mottagna datan.

Då tjänsterna som används i denna klass är i behov av en referens till den data som skall visas, initieras även datamodeller för grafdata i `GraphDiagramViewModel`:s konstruktor. För att komplettera denna modellens närvaro, initieras även en instans av det gemensamma gränssnittet IDataChannel, till vilken olika kommunikationskanaler kan kopplas oberoende på hur gränssnittet hanterar insamling av data.

Denna klass exponerar ett flertal egenskaper, vilket främst innefattar instanser av de två grafer som tilldelas direkt via GraphDiagramView:s bakomliggande kod fil. Detta medför lite högre koppling än om dem hade tilldelats via databindning, och är ett resultat av att ScottPlot biblioteket ej stödjer detta i förmån av högre prestanda, vilket var att prioritera i detta projekt. Referenser till graferna tilldelas till nödvändiga tjänster vid initiering, sådan att de kan kapsla in all logik med avseende på en given graf. Vidare exponerar klassen även databundna egenskaper för att hantera synligheten av olika grafer för att möjliggöra för användaren att manipulera applikationens upplägg vid behov för att utöka användarvänligheten. Även dessa egenskaper kan via MessageBus meddelanden skifta sin synlighet och reaktivt uppta den yta som finns tillgänglig för grafen.

4.6.3 Hantering av verktygsfält

`HeaderViewModel` klassen fungerar som en centraliserad kontrollpanel som exponerar applikationsövergripande funktioner som sparande och inläsning av konfigurationsfiler, samt möjlighet att växla synligheten av specifika komponenter i applikationens fönster. Denna klass initierar via sin konstruktor ett antal kommandon som exponeras för databindning samt ansluts till klassens privata metoder, sådant att dessa kallas när motsvarande knappar trycks på i användargränssnittet.

Klassen har även en sekundär konstruktor, som motsvarande view:s underliggande kod försöker kalla på om programmet lyckas hämta huvudfönstrets StorageProvider, vilket möjliggör åtkomst till att öppna filutforskaren på datorn. Om detta lyckas, dyker denna upp då SaveConfig och LoadConfig metoden kallas på. Annars sparas konfigurationer till en och samma fil utan användarens inmatning av en filsökväg.

Metoder för att växla synlighet på sidopanelen, linjegrafen samt stapeldiagrammet exponeras även, och skickar enbart ett meddelande över applikationens MessageBus som låter berörda viewmodels ta emot meddelandet för att hantera begäran att ändra sitt lokala tillstånd.

4.6.4 Hantering av sidopanel

`SidebarViewModel` är den viewmodel som ansvarar för att hantera all logik kring användarens val av kommunikationsgränssnitt och tillhörande inställningar i sidopanelen. Utöver att ta emot inmatning av användarens konfigurationer, hanterar den även dynamisk validering av inmatade parametrar och publicerar färdiga konfigurationer via applikationens MessageBus vid försök att ansluta till ett kommunikationsprotokoll. Vidare hanterar den även konfigurering av parametrar som skickas vidare till GraphDiagramViewModel via applikationens MessageBus för att ändra grafernas parametrar. Detta innefattar uppdateringsfrekvens, antalet variabler som skall visas i linjegrafen, status gällande vilka variabler som bör vara synliga i grafen samt vilka variabler som är en del av triggerkanalen.

Klassen är uppbyggd sådan att varje tillgängligt fält i SidebarView har en motsvarande variabel i `SidebarViewModel`, till vilken en databindning existerar för att minimera koppling. Alla dessa håller det värde som respektive fält representerar, och uppdateras dynamiskt då användaren redigerar värden. Vid redigering av sådant som har med grafens konfiguration att göra, notifieras prenumererande viewmodels via MessageBus meddelanden med de uppdaterade värdena. Då en uppkoppling sker, kallas en privat metod ConnectButtonClicked, som skickar ett meddelande om att koppla till den kommunikationskanal som användaren har ställt in. `SidebarViewModel` tar emot ett meddelande för att indikera att en koppling lyckas, i vilket fall den byter ut uppkopplings-knappen till en frånkopplings-knapp, som vid ett klick i stället skickar ett meddelande för att frånkoppla kommunikationskannalen till prenumererande viewmodels.

Då klassen håller instanser för alla konfigurationsparametrar, prenumererar denna även på SaveConfig och LoadConfig MessageBus meddelanden från HeaderViewModel, för att möjliggöra att kunna spara alla dessa parametrar i en konfigurationsfil. Skapande eller avläsning av konfigurationsfiler görs via den IConfigService instans som tillhandahålls av ViewModelBase klassen.

4.7 Licensering

I denna delsektion kommer licenseringen av produkten att förklaras. Det vill säga vad som krävs av företag eller privatpersoner för att få använda eller utöka denna programvara. Början av denna sektion kommer att förklara de olika sorters licenser som är relevanta för projektets utförande, och sedan presenteras en lista med alla bibliotek och vilka licenser som används för dessa bibliotek.

4.7.1 MIT license

Inom projektet används ett flertal externa bibliotek som följer Massachusetts Institute of Technology (MIT) licensering, det är även denna licens som projektets framtagna projektfiler är licensierade under. MIT-licensen betyder att produkten kan användas både privat och kommersiellt. De huvudsakliga krav som ställs är att den ursprungliga licenstexten och upphovsrättsinformationen ska finnas kvar i projektets filer. Vidare anger licensen att programvaran tillhandahålls såsom den är, vilket innebär att användaren bär ansvaret för alla konsekvenser som användningen av programvaran kan medföra.

4.7.2 BSD 3-Clause License

Det används även externa bibliotek som följer BSD 3-Clause Licensen, vilket i sin tur är en öppen källkodslicens likvärdigt MIT. Licensen kräver precis som MIT-licensen att upphovsrätten och licensvillkoren behålls inom projektet, men den förbjuder även användning av upphovsmannens namn i reklam utan tillstånd.

4.7.3 Anvädning av PEAKCAN drivrutiner

Den tredje sortens licens som används är licensen för att använda sig av PEAKCANs drivrutiner. Denna licens är mer komplex och blir därmed förenklad sådant att endast de essentiella delarna för användningen av programvaran nämns. För att få använda sig av PEAKCANS drivrutiner krävs främst godkännande till följande:

- Använda den med PEAK:s hårdvara (eller få bekräftelse om OEM-stöd).
- Inte ta betalt för åtkomst till applikationen.
- Lämna kvar licenstext och copyright.
- Inte ändra dokumentationen.
- Följa avtalet i sin helhet.

5
Resultat

I detta kapitel presenteras resultat för de huvudsakliga målsättningar som har formulerats för projektarbetet. Detta innefattar utvecklingen av ett C-bibliotek för UART-kommunikation och tidsstämpling, samt skapandet av ett PC-baserat visualiseringsverktyg för mikrokontrollerdata i realtid. Vidare behandlas även implementeringen av applikationsspecifika funktionaliteter såsom implementationen av linjediagram, stapeldiagram, triggerfunktionalitet, samt möjlighet att spara och ladda konfigurationsfiler. Resultaten redovisas i relation till de mål som formulerades vid projektarbetets start, baserat på de kravspecifikationer som definierades av KraftPowercon.

5.1 C-bibliotek för UART-kommunikation

Detta avsnitt presenterar resultaten av arbetet med att ta fram ett C-bibliotek för STM32-baserade system, avsett för att möjliggöra effektiv tidsstämplad dataöverföring via UART.

5.1.1 Integration i STM32-baserade inbyggda system

C-biblioteket har utvecklats med fokus på att enkelt integreras i befintliga projekt för STM32-baserade mikrokontrollersystem, i enlighet med det första utav projektets delmål. Biblioteket bygger ej på externa beroenden utöver de låg-nivå standardbibliotek som utges för STM32 mikrokontrollers utav STMicroelectronics. Gränssnittet för biblioteket har utformats för att vara modulärt och enkelt att konfigurera när och hur sampling och transmission av data skall ske. Detta delmål anses vara uppfyllt, då integrering av C-biblioteket i ett givet STM32-projekt som använder sig utav ett STM32F407G-DISC1 utvecklingskort har lyckats utan några omständigheter.

5.1.2 Tidsstämpling av överförd data

För att möjliggöra korrekt analys av mikrokontrollers beteende i realtid ska C-biblioteket enligt det första delmålet även utföra tidsstämpling av varje datapaket som skickas via UART. Denna funktionalitet har implementerats till fullo. Tidsstämpling lyckades implementeras med principer att en global variabel som representerar en tidsstämpel skall bli försedd vid initiering, och att denna skall ökas via periodiska avbrott. Dess nuvarande värde vid en sampling används för att identifiera den tid som en dataavläsning har skett.

5.1.3 Effektiv UART-kommunikation

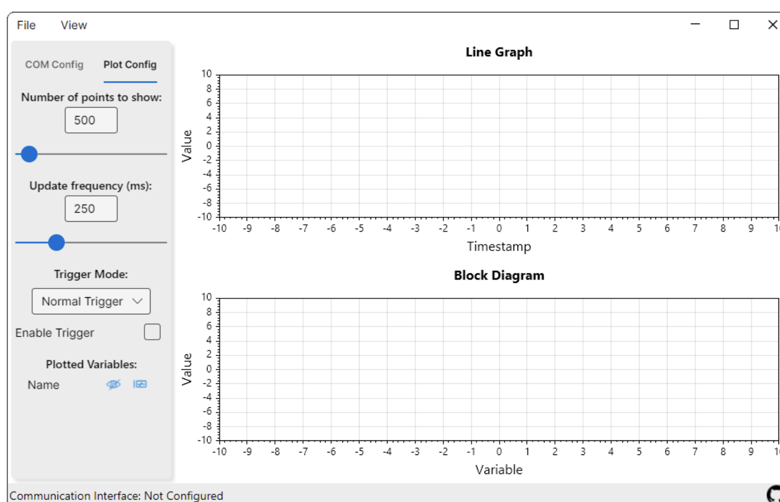
Delmålet som behandlar utvecklingen av C-biblioteket menar även att effektivitet är ett mål som ska prioriteras under utveckling. Biblioteket har optimerats för snabb dataöverföring genom det skräddarsydda binära protokoll som används vid transmission av data. Enbart de bytes som representerar en del av ett datavärde eller en tidsstämpel skickas, utan behov av övriga bytes för att markera start eller slut av ett datapaket. Enbart en byte skickas även för tidsstämpeln oavsett konfiguration av biblioteket för att maximera dataflödet under en given period. Eventuella överflöden av tidsstämplar hanteras i applikationen, för att tillåta C-biblioteket att vara så högpresterande som möjligt. Empirisk testning visade i projektets slutskede att C-biblioteket klarade av att sampla en 8-bitars variabel på över 30 kHz med 900 kBd/s överföringshastighet. Därmed anses utvecklingen av C-biblioteket vara lyckad och alla dess delmål vara uppfyllda.

5.2 Visualiseringsapplikation för mikrokontroller-data

Detta avsnitt presenterar resultaten av utvecklingen av det plattformsoberoende visualiseringsverktyg som utvecklades för realtidsanalys av mikrokontrollerdata.

5.2.1 Skapande av en grafisk applikation

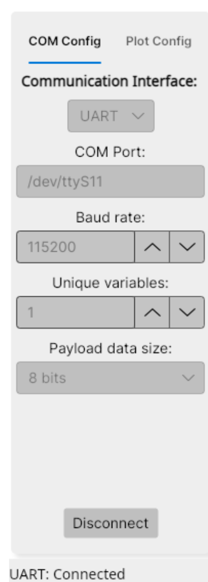
Målet om att utveckla ett plattformsoberoende PC-program för anslutning till befintlig CAN-bus eller UART nämner även att ett grafiskt gränssnitt skall användas. För detta ändamål har alla applikationens funktioner skapats sådant att dessa kan interageras med genom detta grafiska gränssnitt. Ett uniformt och användarvänligt gränssnitt har därmed skapats, se Fig. 5.1, och används därmed som en brygga för användaren när det kommer till interaktion av resterande mål.



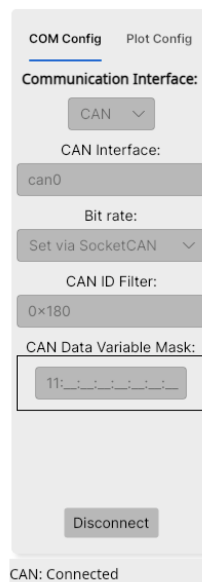
Figur 5.1: Applikationens gränssnitt med linjegraf och stapeldiagram aktiverat, utan uppkoppling till mikrokontroller

5.2.2 Uppkoppling till kommunikationsgränssnitt

Det tidigare nämnda målet gällande utveckling av ett plattformsoberoende PC-program anger även att denna skall kunna kopplas upp till en befintlig CAN-trafik, eller UART via det utvecklade C-biblioteket. Båda dessa funktioner har implementerats och är funktionella för både Windows och Linux-baserade system. I applikationens sidopanel finns det under *COM Config* en rullgardinsmeny där antingen CAN eller UART kan väljas. Relevanta konfigurationsfält för dessa kommunikationskanaler visas då upp, och vid korrekt konfiguration av en existerande kanal är det även möjligt att koppla till denna, som visas i applikationens statusfält, se Fig. 5.2 och Fig. 5.3.



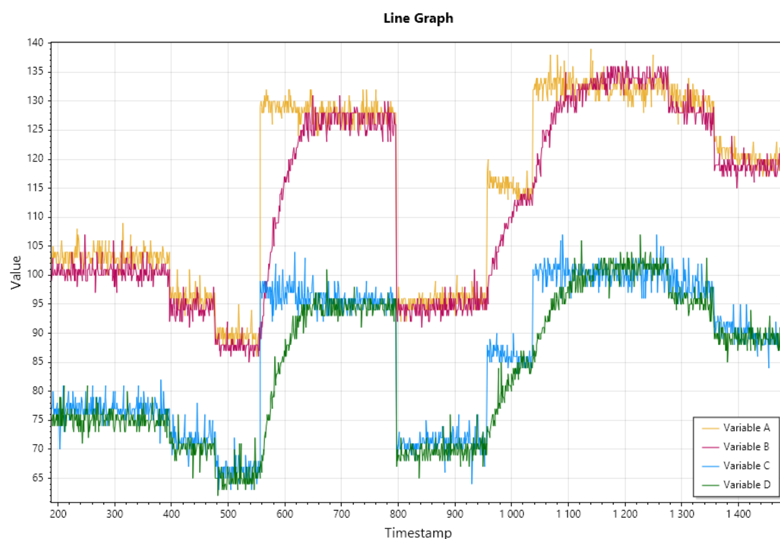
Figur 5.2: Lyckad uppkoppling till UART på Linux



Figur 5.3: Lyckad uppkoppling till CAN på Linux

5.2.3 Linjegraf för realtidsvisualisering över tidsaxel

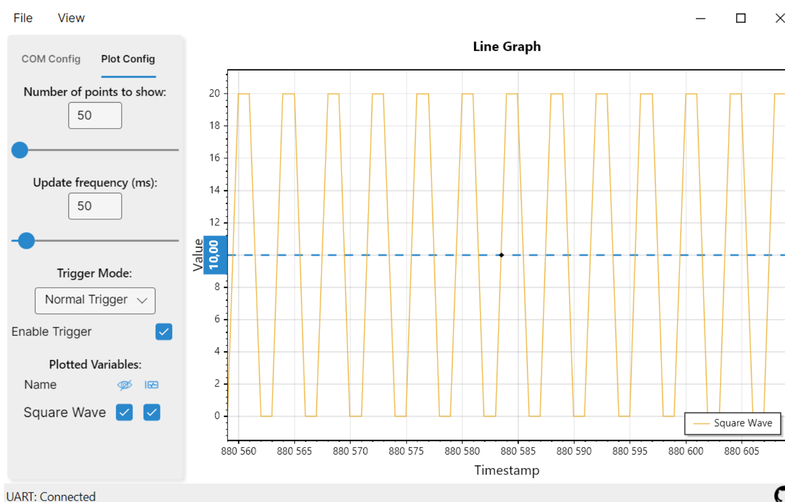
Ett annat mål för projektet var att implementera funktionalitet för en linjegraf i PC-programmet för visualisering av en eller flera variabler över tid, med funktionalitet för zoom och scroll. Detta har implementerats fullständigt. Fig. 5.4 presenterar en mätning som gjordes på en STM32 mikrokontroller. Mer specifikt, visar figuren det så kallade *history-mode*, som aktiverar möjlighet till zoom och scroll efter frånkoppling av en kommunikationskanal. Vid en aktiv koppling till CAN eller UART, uppdateras denna linjegraf automatiskt i den period som är inställd av användaren för att visa nykommen data. Variablerna kan även döpas om, vars namn syns i legenden i nedre högra hörnet av grafen. Empirisk stresstestning via en datagenererande klass visade på att linjegrafen klarar av dataströmmar på över 50 000 paket/s, och att i *history-mode* visa flera tiotals miljoner punkter samtidigt. Resultatet visar tydligt att detta mål har uppnåtts.



Figur 5.4: Linjegraf vid mätning och visualisering av fyra variabler via UART

5.2.4 Triggerfunktionalitet

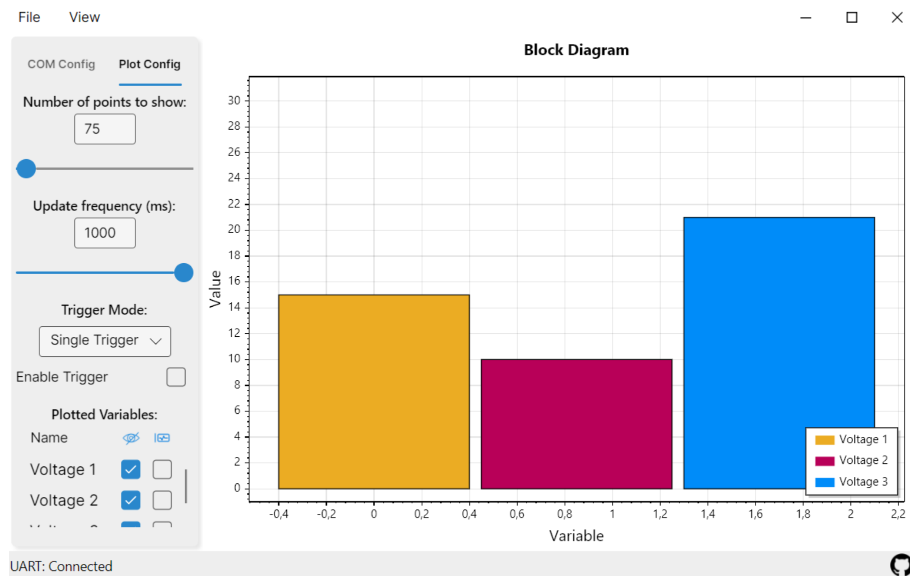
Som en vidareutveckling till linjegrafen skapades även ett separat delmål för implementationen av grundläggande trigger-funktionalitet i PC-programmet liknande den i ett oscilloskop. Även detta har implementerats fullständigt. En triggernivå kan aktiveras för linjegrafen via sidopanelen, vilket visas i Fig. 5.1. Vid aktivering syns en triggernivå på grafens gränssnitt, som användaren sedan kan flytta på med muspekaren. En variabel läggs till i triggerkanalen via en kryssruta på sidopanelen, vartefter triggers kan ske för det inställda läget, som synes i Fig. 5.5.



Figur 5.5: Applikationens gränssnitt vid uppkoppling till mikrokontroller via UART, och aktiverad normal trigger på Y=10 för en variabel

5.2.5 Stapeldiagram för överblick av senaste värden

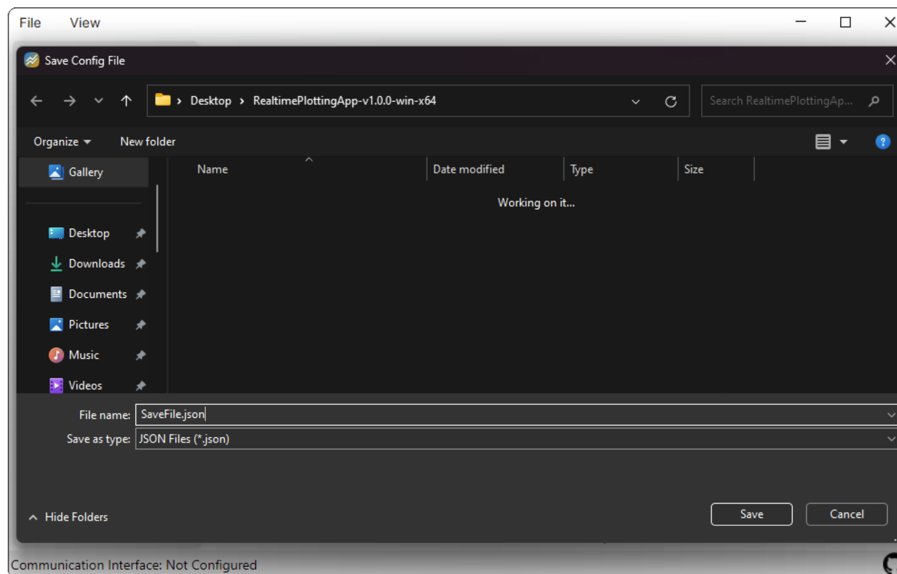
Likt målet för utvecklandet av en linjegraf sattes även mål gällande visualisering av variablers senaste värden som stapeldiagram. Detta målet har uppfyllts, och ett stapeldiagram kan aktiveras från *View*-menyn i applikationens verktygsfält, och ser under datavisualisering ut som visas i Fig. 5.6. Samma namngivning som för linjegrafen används, och det är även möjligt att visa detta stapeldiagram samtidigt som en linjegraf.



Figur 5.6: Applikationens gränssnitt vid uppkoppling till mikrokontroller via UART, med ett stapeldiagram aktiverat som visar senaste värdet för tre variabler

5.2.6 Hantering av konfigurationsfiler

Slutligen sattes även ett mål gällande stöd för att spara och ladda konfigurationsfiler för enklare uppkoppling till kommunikationsprotokollen CAN och UART. Även detta har implementerats fullständigt, och finns tillgängligt genom alternativen *Save Config* och *Load Config* under *File*-menyn i applikationens verktygsfält. Dessa öppnar operativsystemets standard filhanterare, se Fig. 5.7, och dess funktion har verifierats för både Windows och Ubuntu. Felhantering för potentiellt korrupt data har även implementerats, och därmed anses detta mål vara uppfyllt.



Figur 5.7: Filhanterare för att spara och ladda fil på Windows då Save Config eller Load Config har klickats på

6 Diskussion

I följande kapitel kommer projektets resultat och utfall att diskuteras. Huvudsakligen så kommer diskussioner kring användningsområde, design- och teknologival, användarvänlighet, utvecklingsmöjligheter, etik- och hållbarhetsaspekter att behandlas.

6.1 Användingsområde och målgrupp

I detta delavsnitt kommer det utvecklade produktens användningsområde att diskuteras. Mer specifikt diskuteras vilken målgrupp programvaran har, samt de funktioner som användarna kommer att använda produkten till. Målgruppen för produkten är att den ska kunna användas i en företagsmiljö, mer specifikt inom fordonsindustrin eller företag som tillverkar inbyggda system.

Ett visualiseringsverktyg för UART- och CAN-kommunikation är särskilt användbart för en rad olika typer av användare som arbetar med teknik där realtidsdata är centralt. Ett tydligt exempel på detta är fordonsindustrin där utvecklingsingenjörer använder sig av CAN-bussen för att möjliggöra kommunikation mellan olika elektroniska styrenheter i ett fordon. För att underlätta övervakningen av signaler från motorstyrningen eller bromsar kan ett visualiseringsverktyg användas. Det hjälper dem att felsöka problem såsom abnormala datavärden, då komplexa databitar visualiseras på ett simplifierat sätt, sådant att mindre tid går åt att tolka datan och mer tid går åt att felsöka felaktig data.

Ett annat företagsområde där kommunikation över UART och CAN är användbart är inom utvecklingen av inbyggda system, där mikrokontrollers som STM32 används. Inom dessa system spelar både UART och CAN-protokollen en viktig roll för dataöverföring, speciellt för de företag som utvecklar mikrokontroller under högspänningssituationer, där alternativen för datakommunikation är begränsade. Därmed kan utvecklare inom detta område dra nytta av ett visualiseringsverktyg för att på ett simpelt och intuitivt sätt följa förändringar i realtid på den data som sensorer genererar. Det visualiseringsverktyg som implementerats gör det möjligt att omvandla den råa datan från dessa protokoll till en grafisk avbildning av datan, till exempel linjegrafer som uppdateras i realtid.

Fördelen med ett sådant visualiseringsverktyg kommer från just denna egenskap att underlätta mängden tid som måste spenderas på felsökning, samt att det även minskar komplexiteten under felsökning. Istället för att tolka en lång ström av bytes i en terminal, kan utvecklaren istället direkt se vad varje bit representerar med hjälp av grafik; exempelvis skulle användaren kunna ställa in så att applikationen visar variabler såsom temperatur, hastighet eller spänning och kan därmed observera hur

dessa värden förändras över tid. Visualisering är särskilt viktig när flera sensorer eller signaler som representerar variabler är involverade, för då blir det mycket tydligare vilken variabel som representerar vad. Ett fel som annars hade tagit timmar att identifiera via manuell analys kan nu istället direkt synas på applikationen genom ett ovanligt hopp på linjediagrammet.

En annan fördel med ett visualiseringsverktyg är att det flyttar bort komplexitet från användaren och till själva applikationen. Alla arbetare inom en organisation är kanske inte bekväma med att tolka rå CAN-data, men ett linjediagram gör den utgående datan från inbyggda system mer tillgänglig för exempelvis testare eller produktägare. Det ger också en chans att verifiera att datan faktiskt reflekterar verkliga händelser, exempelvis att spänningen verkligen ökar när man ökar strömmen.

Ett annat tydligt argument som gör den tillverkade produkten attraktiv på företagsnivå är dess flexibilitet. Bland annat är den utvecklad sådant att den är plattformsoberoende, vilket betyder att oavsett vilket operativsystem företagen använder sig av, kan de använda programvaran. Detta gäller för både Windows och Linux operativsystem. Detta innebär att på en företagsnivå behöver företagen inte införskaffa specifika datorer för att använda programvaran. Utöver att produkten är operativsystemsoberoende är applikationen även högpresterande, det vill säga att det inte krävs en alldeles för dyr dator för att kunna använda sig av denna applikationen, vilket betyder att företag inte behöver investera i nya datorer.

6.2 Användning av MVVM arkitektur

I detta delkapitel kommer användandet av MVVM arkitektur mönstret att diskuteras, mer specifikt dess fördelar samt problematik som denna arkitektur medför. Bland annat hur användningen av MVVM ger lägre koppling mellan klasser, men att det kostar prestanda på programvaran. Dessutom kommer det att diskuteras varför en helt komplett MVVM struktur inte har använts.

Att använda MVVM som arkitektur i en applikation som bygger på AvaloniaUI är ett naturligt val då Avalonia är byggt med stöd för databindning vilket är naturligt för MVVM mönstret. MVVM reducerar kopplingen genom att View-klasserna bara har ansvar för presentation av data, och inte känner till någon annan extern logik annat än genom dess kopplade bindningar. All logik som genererar data ligger i ViewModel, och View binder sig till egenskaper och kommandon i ViewModel, vilket reducerar kopplingen. Som följd av detta så blir det enklare att testa logiken i ViewModels utan att behöva involvera gränsnittet på applikationen, vilket i längden leder till bättre kodkvalitet och en mer stabil produkt. Den verkliga utmaningen uppstår när man introducerar ett externt kodbibliotek, såsom ScottPlot, som inte är anpassat för att fungera med MVVM-strukturen. ScottPlots funktionalitet är att det är ett kraftfullt bibliotek för datavisualisering, men det är designat på ett sätt där man direkt manipulerar plottens innehåll, vilket bryter MVVM arkiterturmönstret.

Även om ett specifikt bibliotek såsom Scottplot kräver lite avsteg från MVVM-tänket, är detta inga problem för programvaran i sig. För det finns två tydliga lösningar på detta problem, den första lösningen är att använda sig av en wrapperklass som används runt användningen av Scottplot och denna wrapperklassen kan i sin tur stödja databindningar. Problematiken med denna lösning är att det är komplext att

få det att fungera utan att påverka prestandan för mycket. Därför används den andra lösningen, vilket är att tillåta denna delen av programvaran att bryta mot själva mönstret istället för att försöka fixa en komplett MVVM-struktur. Fördelen med denna lösning är att prestandan för datavisualiseringen blir betydligt högre, vilket är attraktivt, då det är själva utritningen som kan anses vara en bottleneck. Därför är det väldigt rimligt att försöka optimera prestandan på denna del av programvaran även om det bryter mot MVVM. Nackdelen med denna lösning är att kopplingen mellan klasserna blir striktare.

6.3 Valet av teknologier

Under projektets gång har gruppen diskuterat olika sorters teknologier eller verktyg som skulle kunna användas för att utföra utvecklingen av realtidsapplikationen. Många av verktygen som togs fram användes inte på grund av bristfällig prestanda eller att verktyget ansågs vara för komplext att använda sig av. I denna delsektion kommer det därför diskuteras om gruppen gjort rimliga val när det kommer till teknologi val, det vill säga ifall det var rätt att kasta undan de alternativa valen.

Under inledningen av utvecklingsarbetet utvärderades ett flertal olika tekniker och ramverk för att hitta den mest lämpade lösningen för den realtidsapplikationen som skulle utvecklas. Som tidigare nämnt i metod-delen var ett av de första alternativen som undersöktes att använda sig av Python som programmeringsspråk. Tanken var att Python är ett välkänt programmeringsspråk som erbjuder ett stort ekosystem av färdiga bibliotek och ramverk, dessutom ansåg gruppen att Python är ett enkelt språk att utveckla programvara, vilket gjorde Python till ett attraktivt verktyg, i alla fall i början av projektet. Precis som nämnt i metod-delen utfördes ett par tester för att testa prestanda, detta gjordes genom att implementera delar av applikationen i Python, men det upptäcktes snart att prestandan inte levde upp till våra krav.

Biblioteken som testades var helt enkelt inte tillräckligt högpresterande för projektets användningsområde, det vill säga avläsning och uppritning av realtidsdata. Detta eftersom att data skulle kunna gå till miste ifall applikationen inte var snabb nog, därför ansåg gruppen att det antagligen var smartare att testa andra programmeringsspråk. Fördelen hade dock varit att applikationen antagligen hade varit enklare att utveckla i denna utvecklingsmiljö, eftersom att Python antagligen har tillräckligt många bibliotek för att göra utvecklingen av en applikation som använder sig av seriell kommunikation enklare. Testerna som gjordes var initiala och saknade optimering, det skulle nog vara möjligt att utveckla applikationen med Python som programmeringsspråk, gruppen valde bara att inte ta risken, då de initiala testerna blev negativa.

Gruppen övervägde också att utveckla applikationen som en webbapplikation med hjälp av ElectricUI och sedan konvertera den till en stationär applikation med Electron. Denna lösning hade fördelen att Electron möjliggör plattformsoberoende distribution sådant att den applikationen kan användas på både Windows och Linux utan problem. En annan fördel är att denna lösning verkade vara högpresterande, det vill säga att den antagligen hade klarat av gruppens prestandakrav, dock verifierades inte detta. Trots dessa fördelar valde gruppen att inte ens testa att utveckla med detta alternativ. Gruppen var överens om att webbaserade lösningar kan innebära

säkerhetsrisker i form av att det kan anses att ramverket i sin helhet kan innehålla ovanliga buggar som enkelt kan uppstå om de inte hanteras på ett korrekt sätt.

Ett annat problem som gruppen upptäckte var att gruppen tolkade ramverket som volatilt, då gruppen ansåg att webapplikationers ramverk uppdateras ständigt, vilket betyder att i längden hade en sådan applikation behövt ha mer underhåll, det vill säga att det finns risken att applikationen måste uppdateras ständigt. Vid projektets start hade gruppen även ingen kunskap alls inom webapplikationsutveckling, vilket indirekt betyder att medlemmarna av gruppen hade behövt spendera en stor mängd tid på att lära sig bra design inom webbapplikationer, samt att lära sig syntax, vilket skulle ta extra lång tid. Ett annat problem är att ElectricUI inte är ett gratis verktyg, vilket skiftade gruppen mer åt andra verktyg.

Efter att ha utvärderat de två andra alternativen valde gruppen att testa att utveckla applikationen i programmeringsspråket C#. Beslutet baseras på flera faktorer, exempelvis att gruppen hittade ett par relevanta bibliotek för att utveckla ett visualiseringsverktyg. Dels fann gruppen bibliotek som ScottPlot, för att rita grafer, och AvaloniaUI, som tillåter ett användargränssnitt som fungerar på både Linux och Windows. Dessutom är C# ett objektorienterat programmeringsspråk vilket gruppen är erfarna inom, något som bidrog till att utvecklingen kunde ske effektivt och utan ett omfattande inlärningsmoment.

Precis som tidigare nämnt i metod-delen gjordes ett prestandatest för att säkerställa att valda tekniker kunde hantera gruppens krav. Detta prestandatest testade specifikt grafritningen som Scottplot erbjöd. Visade detta prestandatest att ScottPlot i kombination med C# uppnådde våra förväntade prestandakrav med god marginal, vilket bekräftade att valet av Scottplot var lämpligt utifrån ett rent tekniskt perspektiv. En nackdel som gruppen stött på under utvecklingen var det inbyggda biblioteket för seriell kommunikation i C#. Det visade sig vara bristfälligt i vissa avseenden, vilket ledde till problem med stabilitet och funktionalitet. Gruppen tvingades därför lägga extra tid på att ta fram en alternativ lösning, vilket innebar mer arbete än planerat för att säkerställa pålitlig kommunikation med den externa hårdvaran.

6.4 Användarvänlighet

I detta avsnitt kommer applikationens användarvänlighet att utvärderas. Bland annat kommer de åtgärder som gruppmedlemmarna vidtagit för att förbättra UI-interaktionen för användarna samt även framtida åtgärder som skulle kunna förbättra interaktionen.

För att underlätta för användarna har gruppen lagt viktiga funktioner längst upp i applikationens header. Headern innehåller funktioner såsom att kunna ladda eller att spara konfigurationsfiler, samt möjligheten för användare att kunna reglera vilka sorters grafer som ska kunna visas. Att dessa funktioner är placerade högst upp i gränssnittet gör dem lättillgängliga och vilket anses som ett positivt designval av gruppmedlemmarna eftersom att gruppen anser att huvudfunktioner bör placeras i användarens direkta synfält. Dessutom medför applikationen friheten att växla mellan stapel- och linjediagram, men också att användarna kan välja att ha på båda sorternas diagram samtidigt, vilket anses vara positivt ur ett användarperspektiv.

Utöver headern finns även sidopanelen som har två vyer, där den första vyen håller på logiken bakom alla sorters konfigurationsinställningar och är ett effektivt sätt att gruppera alla värden som krävs för att starta kopplingen mellan hårdvara och applikation. Den andra vyen innehåller inställningar som påverkar grafen. Detta anses vara ett effektivt sätt att separera olika användningsområden, speciellt med tanke på att båda vyerna har olika användningsområden som båda två är koherenta. Detta betyder att beroende på vad användaren vill göra, behöver de inte navigera mellan två olika vyer. Utan om de ska koppla till hårdvaran kan de vara på en vy och ska de påverka grafens utritning kan de vara på den andra. Navigeringen mellan de två vyerna är simpel, detta är gjort genom att ha relevant text på knapparna som växlar mellan vyerna. Knapparna som växlar mellan vyerna ligger längst upp i sidopanelen för att underlätta för användarna, och har tydlig design för att göra det tydligt att det är knappar som man kan klicka på.

Ett annat stor aspekt som gruppmedlemmarna lagt fokus på är inputvalidering, det vill säga att se till sådant att alla fält som användaren kan interagera med, endast accepterar data på rätt format. Med andra ord kan användaren inte klicka på connect-knappen ifall alla fält inte är på korrekt format. Utöver att connect-knappen inte går att klicka på ifall något av fälten är på felaktigt format, blir connect-knappen även grå för att indikera att något saknas i något av fälten. Problematiken med input validering är att det kan vara förvirrande för användarna att förstå vilket format alla statusfält ska ha, det är därför gruppmedlemmarna även implementerade hjälptexter i applikationen. Hjälptexterna visar på vilket format varenda fält ska ha, och ger även exempel på vanliga infyllningar som skulle kunna ske. Kombinationen av allt detta anser gruppen är användarvänligt, eftersom att användarna får direkt feedback ifall de har skrivit in på ett felaktigt sätt, samt att de får hjälpmedel i form av texter som underlättar förståelsen av hur programvaran fungerar, men även vilka format som gäller för att användarna ska kunna använda applikationen på ett korrekt sätt.

Gruppen har även lagt fokus på säkerhet. Säkerhet i detta fall betyder att förhindra användarna från att förstöra applikationen och inläsningen av data. Ett tydligt exempel på detta är att under inläsning av data låses många av fälten sådant att det inte är möjligt för användaren att ändra datan i dessa fält. Detta är en säkerhetsåtgärd som finns för att garantera att programvaran fungerar på ett förutsägbart sätt. Ett exempel på detta är att ändra baudrate i mitten av inläsning kommer att ge felaktiga resultat som inte går att läsa av. Därför är det rimligt att stänga av sådant användarna inte gör sådana val av misstag, det vill säga att förhindra användarna från att förstöra för sig själva. Ett annat exempel på detta är att strömmen mellan applikationen och hårdvaran stängs av automatiskt om användaren skulle stänga av applikationen under avläsning. Anledningen till detta är att annars hade hårdvaran fortsatt överföra data medan applikationen inte är uppkopplad till hårdvaran. Hade strömmen fortfarande varit ingång, hade applikationen och hårdvaran blivit osynkroniserade nästa gång som applikationen kopplas upp med hårdvaran och värdena som läses av från hårdvaran blir därmed oläsbara, vilket inte är användbart för användarna. Det är därför sådana åtgärder har tagits, för att garantera att applikationen fungerar i så många situationer som möjligt, vilket i sin tur betyder att användarvänligheten för applikationen även ökar, då det finns väldigt få sätt för användarna att förstöra applikationen.

Ett undantag till detta är att gruppen inte ser till att strömmen stängs av ifall användarna väljer att medvetet dra ut sladden. Problematiken är då att applikationen försöker skicka ut ett meddelande för att stänga av överföringen av data, men eftersom sladden är utdragen kommer inte detta meddelande fram, och därför kommer data bli osynkroniserad precis som nämnts tidigare. Anledningen till varför detta inte fixats är på grund av att det hade varit komplext. För att detta skulle kunna fungera, måste applikationen kontinuerligt kommunicera med hårdvaran för att kontrollera att det fortfarande finns en koppling, och därmed påverkas prestandan, dessutom hade detta tagit tid att implementera. Därför anser gruppen att det är mer rimligt att lämna kvar detta problem och hoppas att användarna inte rycker ut sladden och sedan klickar på disconnect-knappen.

Tidigare har visuell feedback i form av att connect-knappen blir färgad grå ifall knappen inte går att klicka på. På ett liknande sätt görs visuell feedback i botten av applikationen, det vill säga i applikationens footer, fast i detta fall är detta gjort med text i form av både fel- och statusmeddelande. Dessa meddelanden visas för användaren om något har gått fel under användningen av applikationen. Ett tydligt exempel på en sådan situation är ifall en användare skulle försöka koppla applikationen till en port som inte finns, eller om applikationen saknar behörighet till denna port. Att dessa fel visas upp direkt i applikationen underlättar för användaren att lista ut vad det är som har gått fel genom att texten uppdateras efter vad som gått fel. Detta gör i sin tur att användaren snabbt kan identifiera vad som gått fel och möjligtvis även varför felet har inträffat. Utifrån användarens perspektiv minskar därmed frustrationen när programvaran inte fungerar som förväntat, samt att felsökningen blir mer effektiv. Detta kan ge värdefulla insikter och ökar förståelsen för de problemen som skulle kunna uppstå. Det gör det också lättare för användaren att själv vidta rätt åtgärd utan att behöva konsultera extern dokumentation.

Utöver negativ feedback innehåller footern meddelanden som berättar om saker fungerar som förväntat, det vill säga positiv feedback. Ett exempel på detta är när en anslutning till den externa hårdvaran har lyckats eller om användaren försöker koppla ifrån sig från hårdvaran och när en frånkoppling har genomförts på ett korrekt sätt. Detta är en egenskap som underlättar för användaren, då de får direkt visuell feedback som bekräftar att programvaran har kopplat upp precis som förväntat. Genom att bekräfta att en åtgärd utförts som förväntat får användaren trygghet och tydlighet i interaktionen med applikationen vilket bidrar till en känsla av kontroll, vilket är centralt för en god användarupplevelse.

6.5 Utvecklingsmöjligheter

Applikationen i dess nuvarande läge erbjuder applikationen möjligheten för kommunikation mellan en PC och externa enheter såsom mikrokontroller via informationsöverföringsprotkoll som UART och CAN. Utöver bara kommunikation tillåter applikationen även visualisering av insamlad data i form av både stapel- och linjediagram. Trots detta finns flera tydliga utvecklingsmöjligheter som skulle kunna öka applikationens flexibilitet, användningsområden och användarvänlighet ytterligare.

En uppenbar vidareutveckling är att utöka antalet sorters kommunikationskanaler som applikationen kan hantera, det vill säga utöka sådant att det finns fler alternativ än bara CAN och UART. Exempel på detta är att implementera sådant att applikationen stödjer fler sorters överföringsprotokoll, exempelvis stöd för överföring via Serial Peripheral Interface, Inter-Integrated Circuit eller Transmission Control Protocol/Internet Protocol. Genom att utveckla stöd för flera överföringsprotokoll skulle verktyget kunna användas av fler hårdvaror vilket betyder fler tekniska miljöer och projekt, där olika protokoll krävs beroende på hårdvarans natur. Denna utökning skulle öka applikationens relevans för en bredare användargrupp. Bland annat skulle det vara nytta för inbyggda systemutvecklare men möjligtvis även nätverksanalytiker.

När gruppmedlemmarna utvecklade applikationen lades mycket fokus på utvecklingen av triggerfunktionalitet, men gruppen anser att mycket mer funktionalitet kan läggas till gällande triggerhantering. I nuläget stödjer applikationen endast single trigger och normal trigger, detta görs endast vid positiv flank. Detta begränsar användningen av applikationen, då det finns tillfällen där andra sorters triggerfuntkionalitet skulle vara användbart. Exempelvis kanske en arbetare som testar en hårdvara vill att triggers ska ske endast på fallande flank. Att implementera stöd för fler typer av triggers, som stigande, fallande, och till och med nivåbaserade triggers, skulle göra applikationen mer kraftfull vid felsökning och signalanalys.

Vidare kan även fler sorters grafer implementeras för att bredda visualiseringsfunktionaliteten. För nuvarande är alternativen som implementerats begränsade till stapel- och linjediagram. Scottplot stödjer dock ett flertal andra sorters grafer, vilket betyder att möjligheten finns att införa flera sorters diagramtyper, exempelvis stöd för scatterplot-grafer. Dessa grafer skulle kunna användas för att tydligt visualisera relationer mellan olika variabler, vilket skulle ge användaren fler sätt att tolka och analysera datan beroende på kontext och behov.

En mer avancerad fast en mer betydande funktionell förbättring vore möjligheten att koppla flera kommunikationskanaler till separata grafer samtidigt. Till exempel skulle en användare kunna skapa två parallella linjediagram, där det ena visar data från en CAN-anslutning och det andra från en UART-anslutning. Att kunna skapa flera grafer skulle skapa helt nya möjligheter för användaren. Användaren skulle då kunna jämföra olika signaler, analysera tidsfördröjningar eller verifiera samspel mellan systemkomponenter i realtid. Det skulle även förenkla felsökning, eftersom man inte behöver byta konfiguration eller starta om anslutningar för att växla mellan olika datakällor. Anledningen till varför det anses vara mer avancerat än de tidigare exemplen, är på grund av att koden hade behövts skrivas om relativt mycket, då den är baserad på att endast från en port hämta in extern data.

Slutligen finns det stort värde i att införa exportfunktionalitet för grafdatan, exempelvis till Excel-format. Detta skulle möjliggöra vidare analys, dokumentation eller rapportering utanför applikationen, vilket skulle vara användbart då det finns tillfällen där man vill extrahera datan till ett annat externt verktyg och då är Excel ett passande filformat att extrahera data till. Detta skulle kunna vara användbart på en företagsnivå där företag ofta använder sig Excel eller liknande för att hantera data.

6.6 Etikaspekter

I denna delsektion kommer en kortsiktig etisk analys av relevanta etikramverk att genomföras, och även en diskussion till varför dessa ramverk anses vara relevanta för detta projekt. Utifrån denna analys kommer det även att avgöras om utvecklingen och användningen av applikationen kan anses vara etiskt försvarbar eller inte.

De ramverken som gruppen valt att diskutera är pliktetik, konsekvensetik i större utsträckning. Anledningen till varför fokus läggs på pliktetik och konsekvensetik är på grund av att dessa ramverk anses av gruppen handla mer om handlingar och möjliga konsekvenser till dessa handlingar vilket anses mer relevant än ramverk som handlar om vilka personlighetsdrag som är etiska. Istället för ramverk som fokuserar på personlighetsdrag såsom dygdetik anser gruppen nämligen att det är mer relevant att fokusera på allt som har med applikationen att göra. Exempelvis hur den skulle kunna användas i framtiden av användare, och hur ansvariga gruppmedlemmarna är för konsekvenser som skulle kunna ske i och med publiceringen av denna applikation.

Det första ramverket som kommer diskuteras är pliktetik som handlar om att göra det rätta oavsett konsekvenserna. Detta ramverk är väldigt brett och det går att diskutera om nästan vad som helst, det är därför gruppen har valt att definiera vilka plikter som gruppen anser vara relevanta för utvecklingen av applikationen. Plikten att följa juridiska lagar är det första som anses relevant. Gruppen anser att det enklaste sättet att bryta mot lagen i denna situation är att av misstag bryta mot någon av licenserna som medförs av alla externa bibliotek. Därför har mycket fokus lagts på att undersöka vilka licenser som applikationen får av alla externa bibliotek, sådant att gruppen inte bryter mot någon av dessa. Bland annat, som nämnts i systemkonstruktionens avsnitt, finns tre sorters licenser på de externa biblioteken. Vilket inkluderar MIT, BSD-3 samt PEAKCANs egna licens. Ingen av dessa säger emot publicering av ett open source projekt, utan de hade endast kunnat vara problematiska ifall gruppen försökt att sälja produkten kommersiellt. Utifrån denna aspekt kan applikationen anses uppfylla pliktetik.

Det andra ramverket som kommer att diskuteras är konsekvensetik. Detta ramverk hanterar endast konsekvenser av handlingar och eftersom applikationen kommer att vara open source, kommer den att vara öppen att användas för alla, vilket innebär att i längden kommer konsekvenser att ske. En positiv konsekvens är att både företag och privatpersoner får tillgång till ett visualiseringsverktyg som klarar av att använda både CAN och UART-protokollen. Vilket enligt konsekvensetisk tankesätt kan anses vara positivt, då både företag och privatpersoner kan spara pengar genom att använda realtidsapplikationen.

Enligt konsekvensetiken kan man även argumentera att gruppen är oetiska i form av licensen som används till produkten. Produkten använder som sagt MIT, vilket tar bort ansvaret från tillverkarna, det vill säga gruppmedlemmarna för det här projektet, och lägger det istället på användarna som använder produkten. Rent effektivt betyder detta att gruppen inte ansvarar för ifall något negativt händer när någon använder sig av applikationen. Rent konsekvensetiskt tankesätt är detta en negativ sak, då gruppen tillåter att negativa konsekvenser sker utan att ta direkt ansvar för dem.

Anledningen till varför gruppen anser ändå att denna problematik är accepterbar,

är på grund av att gruppen endast består av två medlemmar som båda två är studenter, och att ta ansvar för varenda privatperson eller företag som använder sig av produkten anses vara en orimlig mängd extra arbete för endast två personer. En annan anledning är att, annars hade gruppen även behövt ta ansvar för missanvänding av applikationen, det vill säga, att gruppen hade behövt ta ansvar för att kriminella använder applikationen för att tillverka olagliga produkter. Värt att notera är att gruppen har vidtagit åtgärder för att se till sådant att personer som använder sig av applikationen är medvetna om att det är de som ansvarar ifall något dåligt skulle ske.

6.7 Hållbarhetsaspekter

Detta delavsnitt kommer att innehålla en diskussion om hur hållbar applikationen är utifrån de tre hållbarhetsdimensionerna. Mest fokus kommer läggas på ekonomisk och social hållbarhet, då dessa två anses vara mer relevanta för applikationen.

Den första dimensionen som kommer nämnas är social hållbarhet och en viktig aspekt av social hållbarhet i vårt projekt är att applikationen utvecklas med öppen källkod. Rent direkt innebär detta att vem som helst kan använda sig av applikationen samt att vidareutveckla systemet applikationen oberoende av syftet. Vilket betyder att applikationen finns tillgänglig för personer vars syfte är utbildning, att utveckla hobbyprojekt eller professionellt bruk inom ett företag. Genom att gruppen har tillgängliggjort både källkod och dokumentation främjar gruppen både transparens och delaktighet då användarna kan direkt se hur programvaran är uppbyggd och vad programvaran innehåller.

Precis som nämnts tidigare fokuserar gruppmedlemmarna på användarvänlighet, vilket även kan anses vara en central del av social hållbarhet. Det är därför som applikationens gränssnitt är designat med tydliga färgkontraster för utritningen av variabler, vilket gör det lättare att särskilja de olika variablerna som ritas ut. Detta underlättar för personer med bristande färgseende eller bara rent av nedsatt syn som annars hade haft problem med att tolka data visualisationen. Precis som nämnts tidigare är detta en positiv sak för social hållbarhet då applikationen blir tillgänglig för en bredare grupp av användare, såsom att människor med andra förutsättningar inte exkluderas. Denna tillgänglighetsanpassning stärker därmed både jämlikhet och inkludering, vilket är centrala mål inom social hållbarhet.

Utifrån ett ekonomiskt hållbarhetsperspektiv erbjuder applikationen möjligheten att ersätta resurskrävande mätutrustning med en mjukvara på vanliga persondatorer. Fördelen med detta är att mätutrustningarna ofta kommer med komplex hårdvara som ofta består av en betydande mängd material och komponenter. Dessa komponenter skulle kunna vara metaller, kretskort eller plast, som ofta är dyra i termer av rent material, det vill säga mängden resurser som finns tillgängliga. Detta betyder att med hjälp av applikationen kan man byta ut dessa resurser mot CAN eller UART-kablar och adaptrar som är betydligt billigare, både i termer av materialen de är gjorda av, men även kostnaden att producera dessa kablar. Med andra ord kan man använda enkla kommunikationskablar och digital signalbehandling via mjukvara istället för mätutrustning, men ändå få in samma typ av data. Vid användning av programvaran insamlas och analyseras data även med avsevärt reducerad resursförbrukning.

7
Slutsats

Samtliga mål som gruppen satte upp i början av projektet har uppnåtts med goda marginaler utmätt i både teknisk kvalitet och funktionalitet. Bland annat har ett C-bibliotek för UART-kommunikation utvecklats med stöd för dataöverföring samt tidsstämpling, och är designat så att det kan integreras in i befintliga kodbaser som använder STM32-mikrokontrollers.

Utöver C-biblioteket har även ett plattformsoberoende PC-program utvecklats som stödjer koppling till överföringsprotokollen CAN och även UART, vilket tillåter användaren att läsa av data från en extern hårdvara. Programmet innehåller ett användarvänligt och grafiskt gränssnitt som gör det möjligt att filtrera och specificera vilka variabler som ska läsas av från kommunikationstrafikens data. För att ytterligare stärka visualiseringsförmågan implementerades grafiska diagram i formen av både linje- och stapeldiagram. Linjediagrammet ritar ut en eller flera variabler över tid, och stödjer möjligheten att zooma in och ut på grafen samt möjligheten att se historisk data. Medan stapeldiagrammet visar det senaste mottagna värdet per variabel, lämpar sig väl för snabb överblick på vilket värde varje variabel har i relation till varandra. Triggerfunktionalitet liknande den i ett oscilloskop har även implementerats, vilket möjliggör analysen för specifika signalhändelser i realtid. Applikationen har dessutom stöd för att spara och ladda konfigurationsfiler, vilket underlättar återanvändning och effektiviserar uppkoppling mot olika system.

Kodbasen för PC-programmet är strukturerad på ett modulärt sätt i den utsträckning som det ansågs vara möjligt och utformad på ett sådant sätt att andra utvecklare ska kunna vidareutveckla applikationen samt att den är lätt att underhålla. Mycket fokus har även lagts på att tillämpa SOLID-principerna men att använda lämpliga designmönster där det anses ha varit relevant. Vilket har använts för att underlätta utökning av funktionalitet utan att kompromissa med kodkvalitet.

Syftet med denna rapport har nämligen varit att undersöka om det är möjligt att tillverka ett visualiseringsverktyg som har stöd för CAN och UART, men som samtidigt är modulärt, högpresterande och även användarvänligt. För att svara på frågan har projektet resulterat i ett verktyg för realtidsanalys av extern data, med ett starkt fokus på både användarvänlighet och att applikationen ska vara tekniskt hållbar. Projektets resultat visar även att alla mål som satts har uppfyllts.

Källförteckning

[1] Robert Bosch GmbH, "CAN Specification Version 2.0," 1991. [Online]. Tillgänglig: http://esd.cs.ucr.edu/webres/can20.pdf (hämtad 2025-02-15).

[2] P. Csurcsia, A. Bánovics och I. Kollár, "Digital oscilloscope displays results together with confidence bounds," 2009, s. 81–86. DOI: 10.1109/WISP.2009.5286576.

[3] I. Fushshilat och D. Barmana, "Low Cost Handheld Digital Oscilloscope," Issue: 1, vol. 384, 2018. DOI: 10.1088/1757-899X/384/1/012027.

[4] S. Schierig, T. Steiner och M. Jochim, "HV AC generation based on resonant circuits with variable frequency for testing of electrical power equipment on site," 2008, s. 684–691. DOI: 10.1109/CMD.2008.4580378.

[5] M. Nithin, S. Shetty och B. Gopal, "Fatal electrocution by over-head wire: A case report," *Journal of South India Medicolegal Association*, årg. 5, nr 2, s. 73–75, 2013.

[6] Tektronix, "TBS2000 Series User Manual," 2025. [Online]. Tillgänglig: https://www.tek.com/en/oscilloscope/tbs2000-basic-oscilloscope-manual/tbs2000-series-3 (hämtad 2025-03-01).

[7] KraftPowercon Sweden AB, "Lösningar och produkter för industriell kraftomvandling," 2024. [Online]. Tillgänglig: https://kraftpowercon.com/sv/sv (hämtad 2025-03-01).

[8] J. W. Valvano, *Embedded systems / Jonathan W. Valvano. 2: Real-time interfacing to the Arm® Cortex(TM)-M Microcontrollers / Jonathan W. Valvano*, en, 4th. ed. s.l.: Eigenverl. d. Verf, 2014, ISBN: 978-1-4635-9015-4.

[9] STMicroelectronics, "STM32F4DISCOVERY - Discovery kit with STM32F407VG MCU * New order code STM32F407G-DISC1 (replaces STM32F4DISCOVERY) - STMicroelectronics," 2025. [Online]. Tillgänglig: https://www.st.com/en/evaluation-tools/stm32f4discovery.html (hämtad 2025-02-22).

[10] STMicroelectronics, "ST UM1472 USER MANUAL Pdf Download," [Online]. Tillgänglig: https://www.manualslib.com/manual/2872952/St-Um1472.html (hämtad 2025-02-22).

[11] STMicroelectronics, "STM32CubeMX - STM32Cube initialization code generator - STMicroelectronics," 2025. [Online]. Tillgänglig: https://www.st.com/en/development-tools/stm32cubemx.html (hämtad 2025-02-22).

[12] "Description of STM32F4 HAL and LL drivers," [Online]. Tillgänglig: https://static.stmcu.com.cn/upload/pdf_html/81b9b329af80a0ea371ca58ff999b1f4.html (hämtad 2025-02-22).

[13] Kvaser, "CAN Bus Protocol Tutorial," 2025. [Online]. Tillgänglig: https://kvaser.com/can-protocol-tutorial/ (hämtad 2025-02-15).

[14] NI, "Controller Area Network (CAN) Protocol Overview," 2025. [Online]. Tillgänglig: https://www.ni.com/en/shop/seamlessly-connect-to-third-party-devices-and-supervisory-system/controller-area-network--can--overview.html (hämtad 2025-02-15).

[15] AutoPi.io, "What is CAN bus: Everything You Need to Know the Protocol," 2024. [Online]. Tillgänglig: https://www.autopi.io/blog/can-bus-explained/ (hämtad 2025-02-15).

[16] G. Packard, "CAN bus protocol: How it works and understanding applications," aug. 2024. [Online]. Tillgänglig: https://www.processingmagazine.com/process-control-automation/article/55134704/gemini-valve-can-bus-protocol-how-it-works-understanding-applications (hämtad 2025-02-15).

[17] L. Davis, "CAN Bus Interface Description CANbus Pin Out, and Signal Names. Controller Area Network," 2009. [Online]. Tillgänglig: https://cms-emu-slicetest.web.cern.ch/904/Documentation/Manuals/CAN%20Bus%20Interface.htm (hämtad 2025-02-15).

[18] Actisense, "What is CAN Bus and how does it work?" 2023. [Online]. Tillgänglig: https://actisense.com/news/what-is-can-bus-and-how-does-it-work/ (hämtad 2025-02-15).

[19] S. Campbell, "Basics of UART Communication," febr. 2016. [Online]. Tillgänglig: https://www.circuitbasics.com/basics-uart-communication/ (hämtad 2025-02-15).

[20] W. Huang och G. Sheng, "Analysis and Research on UART Communication Protocol," i *2024 4th Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*, febr. 2024, s. 768–771. DOI: 10.1109/ACCTCS61748.2024.00140. Tillgänglig: https://ieeexplore.ieee.org/document/10612928 (hämtad 2025-02-15).

[21] E. Peña och M. G. Legaspi, "UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter | Analog Devices," 2020. [Online]. Tillgänglig: https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html (hämtad 2025-02-15).

[22] L. Cao, J. Chen och J. Li, "Working principle and application analysis of UART," 2023, s. 255–259. DOI: 10.1109/EEBDA56825.2023.10090571.

[23] M. Tsitoara, *Beginning Git and GitHub: A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer*, en. Berkeley, CA: Apress, 2020, ISBN: 978-1-4842-5312-0 978-1-4842-5313-7. DOI: 10.1007/978-1-4842-5313-7. Tillgänglig: http://link.springer.com/10.1007/978-1-4842-5313-7 (hämtad 2025-02-15).

[24] M. Tsitoara, *Beginning Git and GitHub, A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer*, 1. utg. Berkeley, CAs: Apress, 2020.

[25] GitHub, "About GitHub and Git," 2025. [Online]. Tillgänglig: https://docs-internal.github.com/en/get-started/start-your-journey/about-github-and-git (hämtad 2025-02-15).

[26] B. W. Kernighan och D. M. Ritchie, *The C Programming Language*, 2. utg. Englewood Cliffs, NJ: Prentice Hall, 1988.

[27] JetBrains, "What are Domain-Specific Languages (DSL) | MPS by JetBrains," 2025. [Online]. Tillgänglig: `https://www.jetbrains.com/mps/concepts/domain-specific-languages/` (hämtad 2025-02-15).

[28] EuroCC, "Motivation and overview — CMake Workshop documentation," 2020. [Online]. Tillgänglig: `https://coderefinery.github.io/cmake-workshop/motivation/` (hämtad 2025-03-03).

[29] Triangles, "Introduction to modern CMake for beginners," 2020. [Online]. Tillgänglig: `https://www.internalpointers.com/post/modern-cmake-beginner-introduction` (hämtad 2025-03-03).

[30] Kitware, "cmake-buildsystem(7) — CMake 4.0.0-rc2 Documentation," 2025. [Online]. Tillgänglig: `https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html` (hämtad 2025-03-03).

[31] Microsoft, "Overview - A tour of C#," maj 2025. [Online]. Tillgänglig: `https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview` (hämtad 2025-02-15).

[32] Microsoft, "C# | Modern, open-source programming language for .NET," 2025. [Online]. Tillgänglig: `https://dotnet.microsoft.com/en-us/languages/csharp` (hämtad 2025-02-15).

[33] Microsoft, "Introduction to .NET - .NET," jan. 2025. [Online]. Tillgänglig: `https://learn.microsoft.com/en-us/dotnet/core/introduction` (hämtad 2025-02-15).

[34] Microsoft, ".NET programming languages | C#, F#, and Visual Basic," 2025. [Online]. Tillgänglig: `https://dotnet.microsoft.com/en-us/languages` (hämtad 2025-02-15).

[35] Microsoft, "What is NuGet and what does it do?" Okt. 2022. [Online]. Tillgänglig: `https://learn.microsoft.com/en-us/nuget/what-is-nuget` (hämtad 2025-02-15).

[36] Avalonia UI, "Avalonia UI: Cross-Platform UI Frameworks for .NET Developers," 2025. [Online]. Tillgänglig: `http://avaloniaui.net` (hämtad 2025-02-15).

[37] Avalonia UI, "Avalonia XAML | Avalonia Docs," 2024. [Online]. Tillgänglig: `https://docs.avaloniaui.net/docs/basics/user-interface/introduction-to-xaml` (hämtad 2025-02-15).

[38] Avalonia UI, "Data Binding | Avalonia Docs," 2024. [Online]. Tillgänglig: `https://docs.avaloniaui.net/docs/basics/data/data-binding` (hämtad 2025-02-15).

[39] Microsoft, "Avalonia 11.2.4," 2025. [Online]. Tillgänglig: `https://nuget.org/packages/Avalonia/` (hämtad 2025-02-15).

[40] S. W. Harden, "ScottPlot - Interactive Plotting Library for .NET," 2025. [Online]. Tillgänglig: `https://ScottPlot.NET` (hämtad 2025-02-15).

[41] S. W. Harden, "ScottPlot 5.0 Cookbook," 2025. [Online]. Tillgänglig: `https://ScottPlot.NET` (hämtad 2025-02-15).

[42] S. W. Harden, "Signal Plot Performance - ScottPlot 5.0 Cookbook," 2025. [Online]. Tillgänglig: `https://ScottPlot.NET` (hämtad 2025-02-15).

[43] S. W. Harden, “Avalonia Quickstart,” 2025. [Online]. Tillgänglig: https://ScottPlot.NET (hämtad 2025-02-15).

[44] S. W. Harden, “MVVM and Data Binding - ScottPlot FAQ,” 2025. [Online]. Tillgänglig: https://ScottPlot.NET (hämtad 2025-02-15).

[45] Microsoft, “ScottPlot 5.0.54,” 2025. [Online]. Tillgänglig: https://nuget.org/packages/ScottPlot/ (hämtad 2025-02-15).

[46] M. Goodwin, “What Is an API (Application Programming Interface)? | IBM,” april 2024. [Online]. Tillgänglig: https://www.ibm.com/think/topics/api (hämtad 2025-02-15).

[47] MuleSoft, “Types of APIs and how to determine which to build,” 2025. [Online]. Tillgänglig: https://www.mulesoft.com/api/types-of-apis (hämtad 2025-03-06).

[48] T. Biscontini, “Application programming interface (API). - EBSCO,” 2024. [Online]. Tillgänglig: https://research.ebsco.com/c/lu54te/viewer/html/4c7nfshxeb (hämtad 2025-03-06).

[49] T. Biscontini, “Application programming interface (API). - EBSCO,” 2024. [Online]. Tillgänglig: https://research.ebsco.com/c/lu54te/viewer/html/4c7nfshxeb?auth-callid=fe143e0b-1c64-4fa5-a07f-eaeb63b1818c (hämtad 2025-02-15).

[50] T. Ægidius Mogensen, *Programming Language Design and Implementation* (Texts in Computer Science), en. Cham: Springer International Publishing, 2022, ISBN: 978-3-031-11805-0 978-3-031-11806-7. DOI: 10.1007/978-3-031-11806-7. Tillgänglig: https://link.springer.com/10.1007/978-3-031-11806-7 (hämtad 2025-02-15).

[51] R. C. Martin, *Agile software development: principles, patterns, and practices* (Alan Apt series), eng. Upper Saddle River, N.J.: Prentice Hall, 2003, OCLC: 49649630, ISBN: 978-0-13-597444-5. Tillgänglig: http://catdir.loc.gov/catdir/toc/fy035/2002070056.html (hämtad 2025-02-15).

[52] F. Buschmann, R. Meunier, P. Sommerlad, M. Stal och H. Rohnert, *Pattern-oriented software architecture, a system of patterns*, eng. Hoboken, N.J.: Wiley, 2013, OCLC: 864918825, ISBN: 978-1-118-72526-9. Tillgänglig: https://www.overdrive.com/search?q=5DC410E7-9918-41A7-8B6E-1B63C779499C (hämtad 2025-02-15).

[53] Avalonia UI, “The MVVM Pattern | Avalonia Docs,” 2024. [Online]. Tillgänglig: https://docs.avaloniaui.net/docs/concepts/the-mvvm-pattern (hämtad 2025-02-15).

[54] Microsoft, “Model-View-ViewModel - .NET,” sept. 2024. [Online]. Tillgänglig: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm (hämtad 2025-02-15).

[55] E. Gamma, R. Helm, R. Johnson, J. Vlissides och G. Booch, *Design Patterns: Elements of Reusable Object-Oriented Software*, English, 1st edition. Reading, Mass: Addison-Wesley Professional, okt. 1994, ISBN: 978-0-201-63361-0.

[56] GeeksforGeeks, “Observer Design Pattern,” 2025. [Online]. Section: Design Pattern, Tillgänglig: https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/ (hämtad 2025-02-15).

[57] F. Hedenus, M. Persson och F. Sprei, *Hållbar utveckling : nyanser och tolkningar*. Studentlitteratur AB, 2022.

[58] S. O. Hansson, "Teknik och etik," sv, Tillgänglig: `https://people.kth.se/~soh/tekniketik.pdf`.

[59] Python, "Welcome to Python.org," mars 2025. [Online]. Tillgänglig: `https://www.python.org/about/` (hämtad 2025-03-12).

[60] Matplotlib, "Matplotlib — Visualization with Python," 2025. [Online]. Tillgänglig: `https://matplotlib.org/` (hämtad 2025-03-12).

[61] PyQtGraph, "PyQtGraph - Scientific Graphics and GUI Library for Python," 2021. [Online]. Tillgänglig: `https://www.pyqtgraph.org/` (hämtad 2025-03-12).

[62] Electric UI, "Features," 2025. [Online]. Tillgänglig: `https://electricui.com/features` (hämtad 2025-03-12).

[63] Electric UI, "Pricing," 2025. [Online]. Tillgänglig: `https://electricui.com/pricing` (hämtad 2025-03-12).

[64] OpenJS, "Build cross-platform desktop apps with JavaScript, HTML, and CSS | Electron," 2025. [Online]. Tillgänglig: `https://electronjs.org/` (hämtad 2025-03-12).

[65] QCustomPlot, "Qt Plotting Widget QCustomPlot - Introduction," [Online]. Tillgänglig: `https://www.qcustomplot.com/` (hämtad 2025-03-12).

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se

CHALMERS