CODEN:LUTEDX/(TEIE-3130)/1-58(2024)

Industrial Electrical Engineering and Automation

Loggningsfunktion för HKM-robot

Mohammed El-Dimassi

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Loggningsfunktion för HKM-robot

LTH

LUNDS TEKNISKA HÖGSKOLA

LTH vid Campus Helsingborg
Department of Industrial Electrical Engineering and Automation

Examensarbete:
Mohammed El-Dimassi

© Copyright Mohammed El-Dimassi

LTH vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

LTH at Campus Helsingborg
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

Tryckt i Sverige
Lunds universitet
Lund 2024

Sammanfattning

Detta examensarbete fokuserar på PLC-programmering av en loggningsfunktion med structured text för HKM-robot. Syftet var att skapa en förbättrad och användarvänlig lösning för loggning av meddelanden på olika nivåer, inklusive trace, debug, info, error och warn för att underlätta övervakning och felsökning av PLC-program för HKM-robot.

Genom en systematisk metodik och analys av befintliga loggfunktioner och studier av SLF4J-konceptet, utvecklades en loggningsfunktion stegvis. Loggningsfunktionen var kodad i kombination av modulär design och sekventiell programmering, vilket resulterade i en välstrukturerad och lättförståelig kod.

Resultaten visade att loggningsfunktionen fungerade under testning, både för skrivning av loggmedelande till debug output trace och till filer enligt angivna inställningar som funktionen har, såsom antal rader eller antal dagar som får skrivas till en fil. Prestandatesterna visade också olika exekveringstider för alla tillstånd som loggfunktions koden består av.

Sammanfattningsvis ledde examensarbete till en fungerande och användbar loggningsfunktion för PLC-programmering som kan användas för felsökning och övervakning av PLC-koder.

Nyckelord:

PLC-Programmering
Loggningsfunktion
Structured Text
SLF4J
HKM-robot

Abstract

This thesis focuses on the PLC programming of a logger function using structured text for the HKM robot. The aim was to create an improved and user-friendly solution for logging messages of various levels, including trace, debug, info, error and warn, to facilitate monitoring and debugging of PLC programs for the HKM robot.

Through a systematic methodology and analysis of existing logging functionalities, along with studies of the SLF4J concept, a logger function was developed gradually. The logger function was coded using a combination of modular design and sequential programming, resulting in a well-structured and understandable code.

The results showed that the logger function worked during testing, both for writing log messages to debug output trace and to files according to a specified settings that the function had, such as the number of lines or number of days that can be written to a file. Performance tests also showed that the different execution times for all states that the logger function code consists of.

In conclusion, this thesis led to a working and useful logger function for PLC programming that can be used for debugging and monitoring of PLC codes.

Keywords:

PLC-Programming
Logger function
Structured Text
SLF4J
HKM-robot

Förord

Det är med stor glädje att jag presenterar detta examensarbete om PLC-programmering av loggningsfunktion för HKM-robot. Jag vill tacka Mats Lilja, Mattias Wallinius och mina handledare Simon Susnjevic och Bernard Schmidt för stöd och hjälpsamhet samt företaget Cognibotics AB för möjligheten att genomföra detta arbete hos dem.

Innehållsförteckning

1 Inledning **8**
1.1 Bakgrund 8
1.1.1 Tekniskt område 8
1.1.2 Företag 9
1.2 Syfte 9
1.3 Målformulering 10
1.4 Problemformulering 10
1.5 Motivering av examensarbete 11
1.6 Metod 11
1.7 Avgränsningar 11
1.8 Resurser 11
2 Teknisk bakgrund **12**
2.1 PLC 12
2.2 Structured Text 13
2.3 CODESYS V3.5 13
2.4 Keba KeStudio 14
2.5 HKM1800 15
2.6 Lucidchart 17
2.7 FileZilla 18
2.8 SLF4J 18
3 Metod **19**
3.1 SLF4J Konceptet 19
3.2 Flödesschema 20
3.3 PLC programmering 20
3.3.1 Den nya loggningsfunktionen 22
3.3.2 CB_Logger program 23
3.3.3 CB_Logger metoder 24
3.3.4 Implementering av hela koden 25
3.4 Tids test samt test med FileZilla 27
3.5 CB_getNumberOfLines 29
3.7 Programmet Read_File 30
3.8 Källkritik 31
4 Analys **32**
4.1 Utmaningar 32
4.2 Stränghantering i PLC 33
4.3 SLF4J 34
4.4 Loggningsfunktion som bibliotekskomponent 34
5 Resultat **35**
5.1 Funktionell testning 35
5.2 Prestandatestning 36
5.3 Användarupplevelse 42
6 Slutsats **43**

6.1 Reflektion över etisk aspekt 43
6.2 Utvecklingsmöjligheter 44
7 Terminologi 45
8 Källförteckning 46
9 Appendix 47
CB_Logger (PRG) 47
CB_Logger.Debug (METHOD) 53
CB_Logger.Info (METHOD) 53
CB_Logger.Error (METHOD) 54
CB_Logger.Trace (METHOD) 54
CB_Logger.Warn (METHOD) 55
CB_GetNumber_Of_Lines (FB) 55

1 Inledning

1.1 Bakgrund

Examensarbetets fokus ligger på att förbättra loggningsfunktionaliteten för HKM-robotens PLC-kod. HKM står för “Hybrid Kinematic Machine” och roboten används för “pick and place”- applikationer utvecklad av Cognibotics. För närvarande hanteras flödeslogiken och robotens rörelser av PLC kod, skriven i structured text. Det finns en logg som redan nu skrivs i realtid för att underlätta övervakning, felsökning och underhåll. Förbättringar kan göras för att göra loggfunktionaliteten mer omfattande och återanvändbar som en bibliotekskomponent.

1.1.1 Tekniskt område

För att förstå syfte, mål och problemformuleringar, kommer examensarbetet förklara tekniska termer som plc-programmering, Structured Text, flödeslogik, robotrörelsekontroll, loggfunktionalitet, affinitet, allvarlighetsgrad, flaggor.

Här är förklaringar av de tekniska termerna:

1. PLC som står för Programmable Logic Controller, är en digital datoranordning skriven i kod för att styra olika maskiner och processer.

2. Structured Text är ett PLC-programmeringsspråk som är textbaserat och strukturerat.

3. Flödeslogik är sekvens av steg och beslut som styr hur en process eller ett system fungerar.

4. Robotrörelsekontroll innebär övervakning och reglering av hur en rör sig och interagerar med omgivning.

5. Loggfunktionalitet involverar skapande och sparning av händelselogg för övervakning samt felsökning.

6. Affinitet syftar på vilken kategori en loggpost tillhör för att underlätta organisation av loggar.

7. Allvarlighetsgrad indikerar hur allvarlig en viss loggpost är.

8. Flaggor är loggnivåer.

1.1.2 Företag

Cognibotics grundades 2013 med fokus på att använda motorvridmoment och klämanordningsdata för att kalibrera robotkinematik. Cognibotics har utvecklats till ett företag inom robotik och automation, och har industrialiserat sina innovationer, samt erbjuder idag olika produkter och tjänster. Några exempel på industrialiserade robotar som de har jobbat med är HKM, Sigma Tau och Motion precision tool suite. Företaget strävar efter att skapa en hållbar framtid där robotar stödjer och kompletterar mänskliga förmågor.

1.2 Syfte

Syftet med examensarbetet är att skapa en komplett loggverktygslösning för HKM-robotens PLC-kod och bör vara återanvändbar som en bibliotekskomponent för HKM:s PLC-programmerare. Nyttan av loggfunktionaliteten inkluderar möjligheten att underlätta felsökning, övervaka interna tillstånd och stödja underhållsarbete.

1.3 Målformulering

Examensarbetet ska delas in i tre delar som bygger på varandra.

Steg 1

- Analysera och förstå befintlig loggfunktionalitet.

Steg 2

- Utöka den befintliga loggfunktionaltieten med kodspårning för att identifiera var i källkoden loggfunktionen anropas.
- Förbättra den befintliga loggfuntionaliteten genom att införa flaggor för att styra affinitet och allvarlighetsgrad för loggposter. Dessa flaggor ska kunna kontrolleras centralt, vilket ger möjlighet att ställa in affinitet och allvarlighetsgrad.

Steg 3

- Göra den nya funktionaliteten återanvändbar som en loggbibliotekskomponent.

1.4 Problemformulering

Följande frågor kommer att besvaras på examensarbetet:

1. Hur kan befintlig loggfunktionalitet förbättras för att möjliggöra enklare felsökning och underhåll av HKM-robotens PLC-kod?
2. Hur kan kodspårning implementeras för att lokalisera anrop i källkoden och underlätta identifiering av problem.
3. Vilka loggnivåer är relevanta för att uppfylla användarens behov av att kontrollera både affinitets-och allvarlighetsgrad för loggar.
4. Hur kan den utökade loggfunktionaliteten göras återanvändbar som en bibliotekskomponent för PLC-programmerare?

1.5 Motivering av examensarbete

Motiveringen av examensarbetet är behovet av att förbättra loggfunktionaliteten för HKM-robotens PLC-kod för att underlätta underhåll och felsökning. Fördelarna för företaget inkluderar att kunna analysera mjukvarans beteende och funktioner i efterhand.

1.6 Metod

Metoder som kommer användas är analys av befintlig kod, omvandling till existerande loggfunktionalitet, att implemetera kodspårning och flaggor för affinitet- och allvarlighetsgrad, samt utvärdering av den förbättrade loggfunktionaliteten.

1.7 Avgränsningar

Arbetet kommer bara fokusera på förbättring av loggfunktionaliteten inom PLC-koden i structured text och kommer inte inkludera optimering av hårdvara eller HKM-robotens rörelsemotorer.

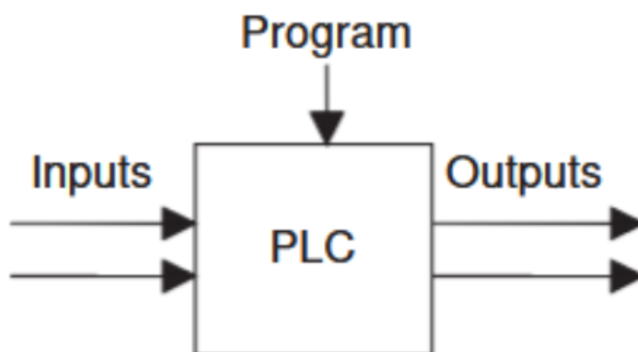
1.8 Resurser

Resurser som behövs är en dator, en arbetsplats på ett företag och ett program som heter KeStudio LX-FlexCore som kan användas för PLC-programmering hos Cognibotics.

2 Teknisk bakgrund

2.1 PLC

En PLC (Programmable Logic Controller) är en form av mikroprocessor för att styra maskiner och processer genom sitt programmerbart minne som används för att spara instruktioner och implementera funktioner såsom logik, sekvenser, räkning, aritmetik och tid (Figur 1). PLC används i stor utsträckning inom olika automationsapplikationer inom till exempel industrier och produktion.



Figur 1. En programmerbar logik kontroller [1].

En PLC kan arbeta i miljöer med ljud, skiftande temperatur, fukt och vibrationer eftersom de är designade för att vara robusta, snabba och driftsäkra. PLC:er använder ett programmerat logiskt schema som behandlar övervakade insignaler från omvandlare, givare och andra enheter och ger utsignaler som styr motorer och andra enheter [1].

Program som styr PLC:er består av programmeringsspråk som följer IEC standarden 611313. Enligt denna standard finns det Ladder Diagram, Instruction List, Sequential Function Chart, Function Block Diagram och Structured Text [1]. Structured Text programming har använts för att skapa en loggfunktion eller i själva verket en mapp med olika loggmetoder. Structured Text användes under examensarbetet eftersom Cognibotics använder Structured Text för robotar.

2.2 Structured Text

Structured Text programming är ett högnivåspråk på PLC som liknar Pascal mycket och som följer IEC standarden 611313. Structured Text (ST) är skapad för textbaserad styrning och kontroll av PLC:er och är utformat för att vara enkelt och lättläst samt användbar för att skriva funktioner och skapa komplexa algoritmer [2]. Med några rader kod kan du skapa en matematisk beräkning som skulle vara mycket jobbigare i Ladder Diagram.

Enligt Simon Susnjevic (personlig kommunikation, 10 april 2024), en av programmerare vid Cognibotics, skiljer sig ST programmering från traditionella programmeringsspråk såsom Java genom att ST kod körs cykliskt i PLC, det vill säga att koden kommer att köras till exempel 1000 gånger per sekund. Traditionella programmeringsspråk kör kod från början till slut efter att koden har kört klart.

2.3 CODESYS V3.5

CODESYS V3.5 är en populär och fri programmeringsmiljö som används för att utveckla och implementera PLC-program i industriella automationssystem. Förkortningen CODESYS står för “Controller Development System". CODESYS erbjuder många olika funktioner och verktyg för att skapa och underhålla PLC programvara [3].

Textbaserad editor för Structured Text är inkluderad i CODESYS, vilket kan vara användbart för mer avancerad och komplex PLC-Programmering. CODESYS följer IEC 61131-3, vilket innebär att den också stöder Function Block Diagram, Ladder Diagram, Instruction List och Sequential Function Chart [3]. Utvecklaren har då möjlighet att välja programmeringsspråket som passar bäst efter deras behov.

CODESYS är standardiserad och plattformsoberoende vilket ger kompatibilitet med olika hårdvaruplattformar från olika tillverkare med direkt stöd i programvaran eller via bibliotek. Det ger många valmöjligheter för utvecklarna när det gäller att välja hårdvara, samt kan användaren utöka och anpassa dess funktionalitet efter behov på grund av sin modulär uppbyggnad.

CODESYS innehåller verktyg och funktioner för att testa, skapa och felsöka PLC-kod, med hjälp av till exempel visualisering och simulering [3].

2.4 Keba KeStudio

Keba Kestudio är en utvecklingsmiljö och programvaruplattform som är byggd på CODESYS 3.5.18.40 och används för konfigurering samt programmering av automationsenheter och styrsystem.
Några exempel av Keba Kestudios funktioner inkluderar:

- Intuitiv konfiguration för kontroll.
    - Har enkel konfiguration för robotar och axlar.
    - Intuitiv konfiguration för fieldbus interface.
    - Script-interface kan användas för automation [4].

- Control programming
    - Det finns variationer av programmering såsom klassisk kontrolluppgifter till att använda C/C++ moduler för IEC 61131-3 programmering.
    - KAIRO instruction set används för robotprogrammering.
    - Innehåller objektorienterade extensions för komplexa kontrolluppgifter och enkel hantering [4].

- Simulering och testning
    - Möjliggör simulering och testning av automatiserade system för identifiera och lösa eventuella problem innan de implementeras till en färdig produkt [4].

Anledningen till att Keba används för tillfället enligt Simon Susnjevic (personlig kommunikation, 10 april 2024) är:

- Stöd för egen kinematik.
- Det är ett fullt ut fungerande robotsystem med ett programmeringsspråk för att styra rörelser.
- En separat tech-pendant.
- Hela systemet är öppet så att Cognibotics kan utveckla deras applikation från grunden.

2.5 HKM1800

En av robotar som Cognibotics utvecklar är HKM1800 och HKM står för Hybrid Kinematic Machine. HKM1800 byggs med starka lågviktsmaterial och är konfigurerad så att mycket av de tunga delarna på roboten (främst motorerna) ligger i basen. Detta minimerar energiförbrukningen och möjliggör hög prestanda. Roboten används bland annat för materialhantering i högautomatiserade lager där det sker flera plock och placering positioner över en stor area (figur 2) [5].

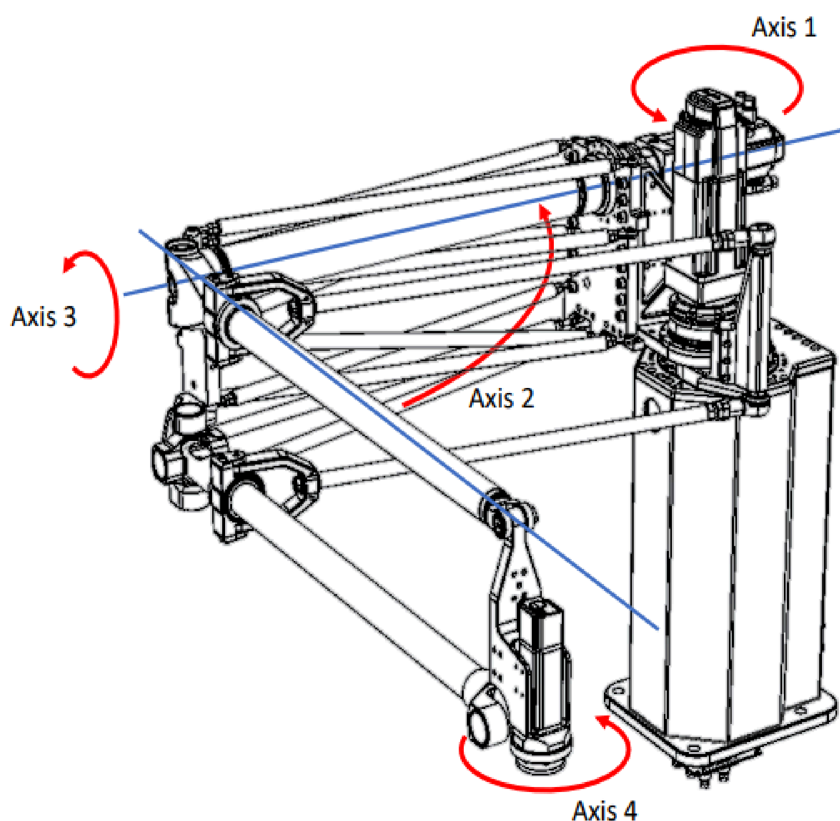
Figur 2. En bild på HKM1800 [6].

HKM1800 är designad att minska fotavtrycken eftersom den tar mindre golvyta och är optimerad för lång räckvidd, vilket möjliggör mer kompakta produktionslinjer och robotar kan jobba tillsammans i en mindre arbetsplats eller produktionsyta. Roboten har dynamisk spårning av rörliga objekt [5].

HKM1800 är byggt med 4 rörliga axlar vilket gör det mer unik än många standard robotar som har 6 axlar (figur 3). Maximala räckvidd som roboten har är 1800 mm radie och max vikt som den kan plocka är 7,5 kg. Roboten väger mindre än 125 kg, har area på 280x320 mm är 1036 mm lång [6].

Movement

Axis	Range	Max. speed
Axis 1	± 180°	500 °/s
Axis 2	± 56°	645 °/s
Axis 3	± 32°	409 °/s
Axis 4	Unlimited	1200 °/s



Figur 3. En bild på HKM1800 med 4 axlar [6].

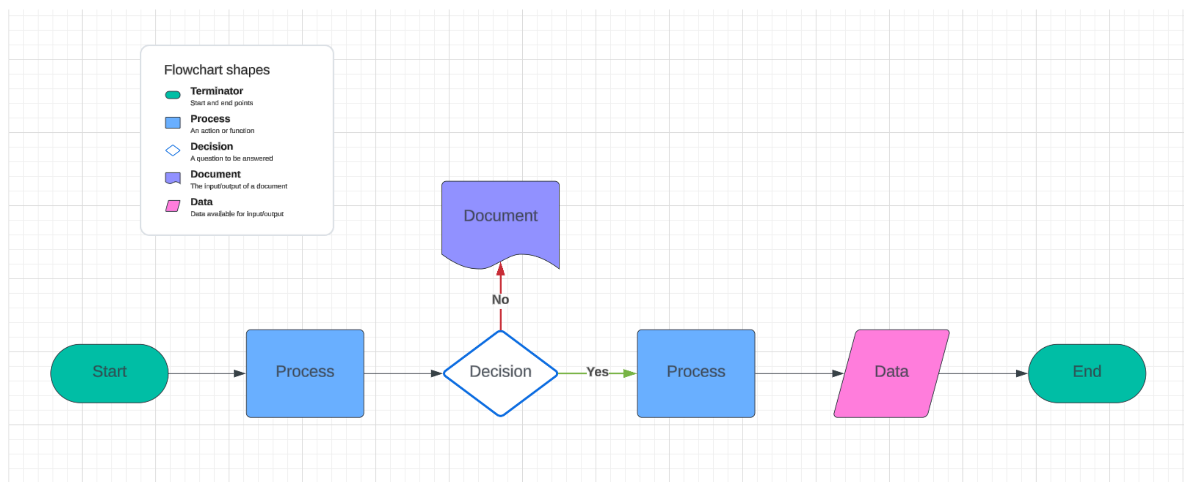
HKM1800 är byggt på hybrid kinematik som kombinerar fördelar av både seriell och parallellkinematik, och det möjliggör lägsta rörlig massa med en kort cykeltid på ett cirkulärt arbetsområde. Robotens plockcykel är 2000 mm på 1,6s, vilket innebär att roboten kan plocka föremål som är 2000 mm ifrån varandra på 1,6 sekunder [5].

2.6 Lucidchart

Lucidchart är en webbsida som används för att skapa och dela flödesschema, flödesdiagram och andra typer av visuella samt grafiska representationer. Skapandet för diagram underlättas av ett intuitivt gränssnitt med en rad olika mallar och verktyg.

Användare kan helt enkelt dra och släppa objekt för att bilda ett diagram på Lucidcharts webbplats och plattformen ger möjligheten för samarbete med andra användare för att arbeta på samma diagram.

Lucidchart kan användas för att skapa flödesschema för arbetsflöden och processer, diagram för hierarkier och organisationsstrukturer, teknisk diagram för systemarkitektur [7]. Cognibotics använder lucidchart för att till exempel bygga tillståndsmaskiner med flödesschema. Se figur 4 för ett exempel på flödesschema.



Figur 4. Ett exempel på ett flödesschema.

2.7 FileZilla

FileZilla är en FTP (File Transfer Protocol) -klient som är en användarvänlig och en öppen källa programvara som finns tillgänglig för Windows, Linux och macOS. Programmet används för att hantera överföring av filer mellan en dator och en server.

Användare kan helt enkelt överföra filer mellan datorer och servrar via FTP, FTPS (FTP över TLS/SSL) och SFTP (SSH File Transfer Protocol). På grund av sitt intuitiva gränssnitt kan användare enkelt dra och släppa filer mellan datorer och servrar samt navigera filstrukturen på båda. FileZilla har andra funktioner såsom överföringshastighet begränsningar, filåterställning, osv [8].

2.8 SLF4J

SLF4J står för "Simple Logging Facade for Java" och är ett koncept över olika loggningsramverk för Java programmering. SLF4J är ett populär utvecklingsverktyg för loggning inom Java på grund av dess enkelhet och kompatibilitet med olika loggnigsramverk. Det ger utvecklare möjlighet att använda samma kod för loggning oavsett vilket loggningramverk som används under kodning. SLF4J gör att användare behöver inte vara bunden till något specifikt loggningsbibliotek som till exempel Log4j [9].

SLF4j följer de loggningsnivåer som stöds av loggningsramverket som används av utvecklare. Loggningsnivåer som stöds av de flesta loggningsramverks med SLF4J är:

- TRACE: Används för att spåra exekveringsflödet och ger information som kan vara användbar för att spåra buggar och fel.
- DEBUG: Används för att logga debuginformation för diagnosering och felsökningsproblem under till exempel testning eller utveckling.
- INFO: Används för att logga information eller händelser för att övervaka programmet.
- WARN: Används för att logga varningar som inte orsakar fel men kräver uppmärksamhet.
- ERROR: Används för att logga fel som har inträffat och som kan påverka programmets funktionalitet [10].

3 Metod

Under arbetets gång påbörjades projektet först genom att analysera och förstå företagets tidigare loggfunktion samt gå genom konceptet av SLF4J för att få idéer om förbättring och omvandling av den existerande loggfunktionen. Den existerande loggfunktionen var bara en KTrace funktion som är skapad av Keba och funktionen användes för logga meddelanden till debug output trace.

Loggfunktionen utvecklades stegvis med nya förbättringar och funktioner efter nya idéer från handledare och hans kollega på Cognibotics, samt att loggningsfunktionen följde SLF4J konventionen. Regelbundna möten mellan handledare och examensarbetare på Cognibotics säkerställde att examensarbetare var på rätt spår under programmeringen och för att snabbt kunna anpassa koden för loggfunktionen efter nya idéer.

3.1 SLF4J Konceptet

Initialt genomgicks konceptet av SLF4J för att få en bättre förståelse för hur en loggningsfunktion kan skapas för PLC-programmering med structured text, vilket är användbart för nybörjare som inte har bra koll över loggningsfunktioner eller som inte har använt loggningsfunktioner innan. Dessa koncept och principer om SLF4J kan tillämpas på andra programmeringsspråk såsom ST för PLC-programmering även om SLF4J används primärt inom Java.

Videor om SLF4J har använts för att lära sig hur loggermedelande kan struktureras och används på ett effektivt och lämpligt sätt, vilket kan underlätta övervakning, felsökning, underhållning och diagnostik av PLC-program. Det som lärdes under studering var olika typer av loggningsnivåer och formatering av loggmedelanden. De olika loggningsnivåer som planerades att använda under implementation av loggfunktionen för PLC är TRACE, DEBUG, INFO, WARN och ERROR.

3.2 Flödesschema

Flödesschema användes med hjälp av webbsidan Lucidchart som ett verktyg för design och planering, för att utveckla och implementera state machines i PLC-kod med Structured Text. Flödesschema möjliggjorde en tydlig visualisering av olika tillstånd och övergångar, vilket underlättade strukturen och förståelsen i state machines.

Olika symboler och former användes för att enkelt kunna definiera och identifiera olika tillstånd och övergångar i flödesschema. De olika tillstånden som skapades med flödesschema är INIT, IDLE, NBR_OF_DAYS_MODE, NBR_OF_ROWS_MODE, OPEN_FILE, TAKE_TIME, WRITE_TO_FILE, CLOSE_FILE, KTRACE och ERROR_HANDLING.

Design av state machines med hjälp av flödesschema översattes till PLC-kod genom ST. Olika strukturer och programmeringstekniker såsom modulärirtet och sekventiell programmering användes för att implementera tillstånd, övergångar och beteenden enligt design från flödesscheman. Mer i detalj om implementation av loggningsfunktionen med hjälp av flödesschema kommer i 3.3 PLC programmering.

3.3 PLC programmering

Innan ett flödesschema skapades gjordes en enkel loggningsfunktion för att få en bättre förståelse på ST programmering och dess konventioner som t.ex att CASE sats används för state machine programmering, hur variabler ska nämnas beroende på variabel typ och hur kallas en metod eller funktion/funktionsblock. Se figur 5 för en enkel implementation av loggningsfunktionen.

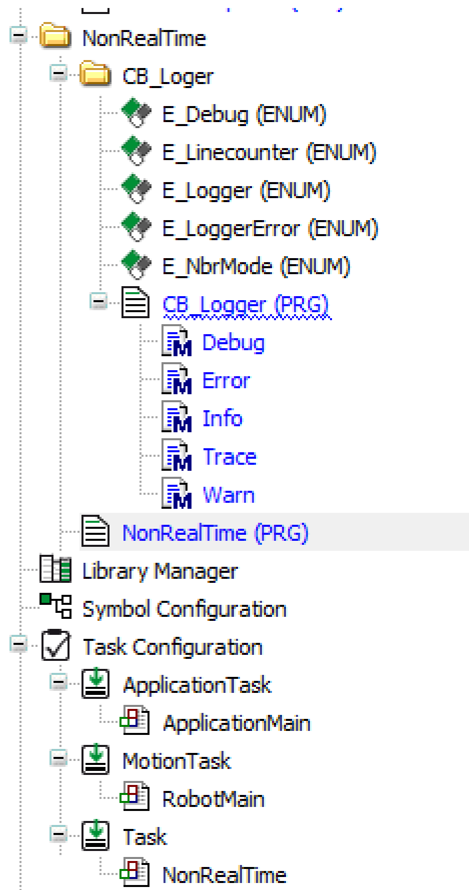
```
METHOD WriteToLogg : BOOL
VAR_INPUT
    sLoggMessage : STRING;  //input
    eLogger : E_Logger;  //input
    instaceOfOpen : FILE.Open();
    instanceIfWrite : FILE.Write();
END_VAR

VAR
    nLoggerLevel : UINT := 0; //not input
END_VAR
```

```
CASE eLogger OF

    E_Logger.Info: //additional information
        nLoggerLevel:=1;

    E_Logger.Warn: //Warning
        nLoggerLevel:=2;

    E_Logger.Error: //error
        nLoggerLevel:=3;

    E_Logger.Debug: //log messages for debugging
        nLoggerLevel:=4;

    E_Logger.Trace: //log messagee for tracing
        nLoggerLevel:=5;

END_CASE
KTrace(Str := CONCAT(CONCAT(sLoggMessage, ' Level: '),TO_STRING(nLoggerLevel)));
```

Figur 5. En enkel kod för loggningsfunktion.

3.3.1 Den nya loggningsfunktionen

I detta avsnitt beskrivs POU (programmable operation unit) program och metoder som loggfunktionen består av.



Figur 6. En bild på CB_Logger (PRG) med metoder

Koden är uppdelad i program och metoder som har olika funktioner beroende på hur utvecklaren vill logga meddelande. Koden har utgått utifrån sekventiell programmering med modularitet och dessa moduler är sedan sammankopplade för att bygga upp till en hel loggningsfunktion. Fördelen med modulär design är enklare förståelse och dokumentation av koden, enklare underhållning och ökad återanvändning. Att koda allting i ett POU program skulle göra koden mindre läsbar och svårare att förstå, samt skulle koden inte hinna exekveras under en cykel om cykeltiden är låg. PLC programmerare måste ofta ta hänsyn till detta annars kan det orsaka fel och problem med ett PLC program om koden skulle ta längre än en exekveringscykel.

Loggningsfunktionen är uppdelad i ett POU program och metoder. Flera metoder kan sedan skapas om det behövs eftersom de byggs på samma POU program som heter CB_Logger. Figur 6 visar PLC-programmets struktur för CB_Logger funktion.

3.3.2 CB_Logger program

CB_Logger programmet är programmerat för att ge flexibilitet och effektivitet när det gäller att logga meddelanden. Det ger användaren möjligheten att välja mellan att logga meddelanden till debug output trace (som fungerar som ett terminalgränssnitt för debug), till en fil eller både trace och fil samtidigt. CB_Logger tar även hänsyn till olika loggningsinställningar, såsom antalet filer som ska användas för loggning, antalet rader med loggmeddelanden som ska sparas i varje fil eller hur länge en fil ska samla in loggmeddelanden innan programmet går vidare till nästa fil.

CB_Logger är som ett main program som körs i bakgrunden och hanterar alla metoder som utvecklaren kan använda inom loggning såsom TRACE, ERROR, INFO, WARN och DEBUG. Programmet CB_Logger innehåller CASE satser för state machine programmering, IF satser, funktionsblock, deklarationer av konstanter, instanser, strängar, vektorer med strängar, tid variabler och andra input variabler som används av metoder.

Programmet “CB_Logger” innehåller följande funktioner :

- hOpenFile: En instans av funktionsblocket FILE.Open, som används för att öppna en fil för att kunna läsa eller skriva i en fil. De olika filemodes för FILE.open är FILE_MWRITE - write, FILE_MREAD - read, FILE_MRDWR - read & write, FILE_MAPPD - append. Om en fil inte existerar så skapas en ny fil om funktionsblocket är på FILE_MWRITE eller FILE_MRDWR. Den returnerar också ett värde som heter file handle, som kan användas som en input “hFile” i funktionsblocken FILE.read, FILE.Write och FILE.Close.

- hWriteToFile: En instans av funktionsblocket FILE.Write som används för att skriva in data till en fil, som nyligen öppnades via FILE.Open funktionsblock. I det här fallet används funktionsblocket för att skriva in loggmeddelandet, som är en sträng, till den öppnade filen.

- hCloseFile: En instans av funktionsblocket FILE.Close används för att terminera tillgången till en fil, alltså stänger den öppnade filen.

- hTimeFile: En instans av funktionsblocket FILE.GetTime returnerar datum och tid för sista modifiering som har skett på en fil.

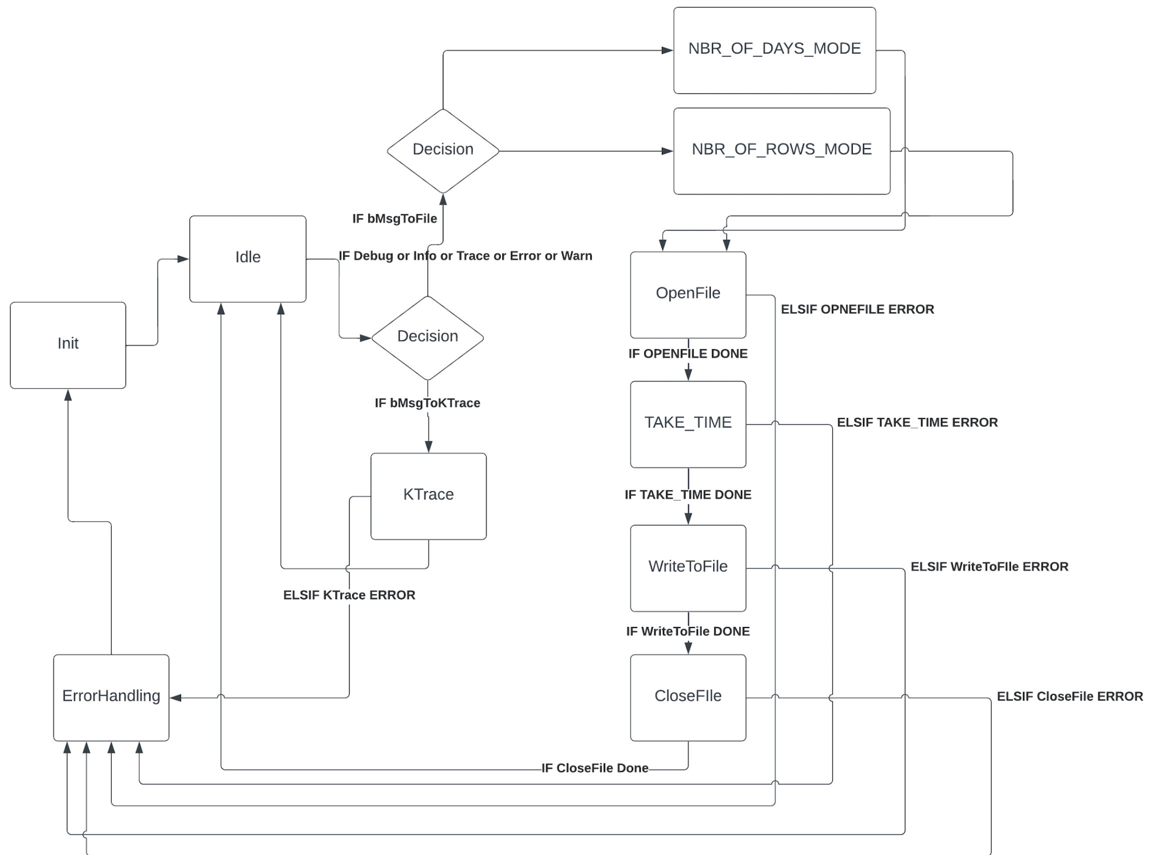
- KTrace: Funktionen KTrace lägger till en sträng på debug output trace

- GetDateTime: Funktionen GetDateTime returnerar nuvarande datum och tid i millisekunder.

3.3.3 CB_Logger metoder

Loggningsfunktionen består av flera metoder. Metoderna för loggningsfunktionen är Info, Debug, Trace, Warn och Error. Metoderna bygger på programmet CB_Logger och används för att logga meddelande beroende på affinitets- och allvarlighetsgrad. Koden i alla metoder är i princip samma bara att de skriver ut olika namn på loggtyp. Metoderna har tre parametrar som input. Först ges loggmedelandet som sträng, följt av två booleska värden. Det första boolean-värdet för variabeln bMsgToTrace, bestämmer om loggmedelandet ska skickas till debug output trace, medan den andra boolean-värdet för variabeln bMsgToFile indikerar om det ska loggas till en fil. Därefter tilldelas strängar till två variabler som är vektorer av strängar. En vektor som heter LogMsgToTrace som tar in strängar för KTrace och en annan vektor som heter LogMsgToFile som tar in strängar för att sedan skriva i en fil. Vektorerna inkrementeras med ett varje gång de får in strängar.

3.3.4 Implementering av hela koden



Figur 7. Flödesschema för loggningsfunktionen.

Implementering för logger funktionen ser ut enligt figur 7 och kan beskrivas med ord enligt följande:

1. CB_Logger börjar på INIT tillstånd där några variabler återställs innan den övergår till tillståndet IDLE.
2. I tillståndet IDLE väntar programmet CB_Logger på några inputs från metoder såsom strängen som ska loggas, bolean input för bMsgToFile, det vill säga om logg strängen ska loggas på filen och boolean input på bMsgToTrace, om loggsträngen ska loggas på debug output trace.
3. Programmet har också några egna inputs som den väntar på IDLE, såsom eNbrMode, aFileLocations, nNbrOfRows, nNBR_OF_DAYS. Variabeln eNbrMode ger möjligheten att välja hur mycket användaren vill skriva meddelande till en fil beroende på antal dagar eller antal rader. Variablen nNbrOfRows ställer in gränsen för antalet tillåtna rader

i en fil. Variabeln nNBR_OF_DAYS ställer in antalet dagar tillåtet att skriva till filen innan det byter till nästa fil. Variabeln aFileLocations är en vektor av strängar som tar in filplatser.

4. När programmet har fått några inputs och om bMsgToFile=TRUE kommer den övergå antingen till NBR_OF_DAYS_MODE eller NBR_OF_ROWS_MODE.
5. Vid tillståndet NBR_OF_ROWS_MODE kontrollerar den om filen har uppnått den maximala gränsen för antalet rader i en fil. Om filen har uppnått maximala antalet rader kommer den byta till nästa fil och nollställa varaibeln nRowCounter som fungerar som en räknare till antalet rader i en fil, annars fortsätter den i samma fil. Därefter övergår programmet till tillståndet OPEN_FILE.
6. Om programmet är i tillståndet NBR_OF_DAYS_MODE kollar den om filen har uppnått maximalt antal dagar för att byta till nästa fil, annars fortsätter programmet på samma fil. Därefter övergår programmet till OPEN_FILE.
7. Vid tillståndet OPEN_FILE används funktionsblocket hOpenFile för att öppna en fil eller skapa en fil om en fil existerar inte. Om funktionsblocket har lyckats göra det övergår den till nästa tillstånd som heter TAKE_TIME, annars övergår den till tillståndet ERROR_HANDLING för att hantera fel.
8. Vid tillståndet TAKE_TIME tar den tid i datum för sista modifiering som har skett på en fil med hjälp av funktionsblocket hTimeFile. Om funktioneblocket fungerade går den över till tillståndet WRITE_TO_FILE, annars övergår den till tillståndet ERROR_HANDLING.
9. I tillståndet WRITE_TO_FILE används funktionsblocket hWriteToFile för att skriva in en eller flera strängar till en fil.. Om det fungerar räknas nRowCounter upp och övergår vidare till tillståndet CLOSE_FILE. Annars går den vidare till ERROR_HANDLING för att hantera fel.
10. Tillståndet CLOSE_FILE använder hCloseFile för att stänga ner den öppnade filen och om det fungerar, övergår programmet tillbaka till IDLE, annars kör den ERROR_HANDLING om någonting har gått fel.
11. Däremot om bMsgToTrace är TRUE kommer programmet att övergå direkt till tillståndet KTRACE från IDLE. I KTRACE loggar programmet en sträng på debug output trace och om det fungerar går den tillbaka till IDLE, annars blir det ERROR_HANDLING.

3.4 Tids test samt test med FileZilla

Tester har utförts på PLC och linuxdelen av PLC med hjälp av filezilla för att se om loggningsfunktionen fungerar som den ska under programmeringen. Flera tester på loggningsfunktionen har utförts för att verifiera att loggmeddelanden genereras och skrivs på en fil på ett önskat sätt. Det som kontrollerades under testning med hjälp av Filezilla är om loggmeddelandet visas i rätt format, att de innehåller relevanta information och att de skrivs på rätt plats samt hur många loggmeddelande går att skriva till en fil.

Debug output trace användes också mycket under testet för att testa om loggningsfunktionen kan logga meddelandet genom olika metoder på debug output trace med hjälp av funktionen KTrace.

Tiden togs mellan olika tillstånden och hela CB_Logger programmet för att testa dess prestanda och hastighet. Tillstånden som testades var KTRACE, OPEN_FILE, TAKE_TIME, WRITE_TO_FILE, CLOSE_FILE, NBR_OF_DAYS_MODE och NBR_OF_ROWS_MODE. Två samma tidsfunktioner som heter util.GetDateTime() användes för att ta sin nuvarande tid. Den första tidsfunktionen används före ett tillstånd och tilldelade tiden till variabeln LastTimeTest. Den andra tidsfunktionen användes direkt innan programmet gick över till nästa tillstånd och tilldelade till variabeln CurrentTimeTest. Därefter subtraherades tiderna med varandra och tilldelade resultatet till varaibel Time_it_took för att veta tiden det tog för att tillståndet ska köra klart innan programmet CB_Logger kan gå till nästa tillstånd. Figur 8 visar ett exempel på tidsmättning på OPEN_FILE tillstånd för att visa hur de gulmarkerade tidsfunktionerna placeras i koden för varje tillstånd.

```
49                  END_IF
50                  LastTimeTest:=ULINT_TO_TIME(util.GetDateTime());
51                  eLog:= E_Logger.OPEN_FILE;
52              END_CASE
53                  bMsgToFile:=FALSE;
54                  StringFilemsg := CONCAT('$n', CONCAT(CONCAT(TO_STRING(Time_taken), sLoggType),(CONCAT(' ', sLoggMsg)))); //string msg that will be sent to a file
55          END_IF
56
57      E_Logger.OPEN_FILE:
58          hOpenFile(xExecute:=TRUE, sFileName:=aFileLocations[nFileIndex], eFileMode := FileMode);      //opening the file
59          CB_GetNumberOfLines(aFileLocations:=aFileLocations[nFileIndex]);
60          IF hOpenFile.xDone THEN
61              hFile := hOpenFile.hFile;
62              linecounter:=CB_GetNumberOfLines.lineCount;
63              hOpenFile(xExecute:=FALSE);
64              CurrentTimeTest:=ULINT_TO_TIME(util.GetDateTime());
65              Time_it_took:=CurrentTimeTest-LastTimeTest;
66              eLog:= E_Logger.WRITE_TO_FILE;  //switching to wirte to file state if opening file worked
67          ELSIF hOpenFile.xError THEN
68              hOpenFile(xExecute:=FALSE);
69              eLoggerError := E_LoggerError.OPEN_FILE_ERROR;   //changing the error to open file error
70              eLog:= E_Logger.ERROR_HANDLING;  // switching to error_handling state
71          END_IF
72
73      E_Logger.WRITE_TO_FILE:
74          hWriteToFile(xExecute:=TRUE, xAbort := FALSE, udiTimeOut := 9000, hFile:=hFile, pBuffer:= ADR(StringFilemsg), szSize:= LEN(StringFilemsg)); //writing the msg
```

Figur 8. En bild på var tidsfunktioner placeras i programmet CB_Logger.

Sist togs tiden för hela CB_Logger Programmet genom att lägga första tidsfunktion på tillstånd som heter IDLE som ska tilldela tiden till LastTimeTest och den andra tidsfunktionen var på tillståndet CLOSE_FILE efter att funktionen hCloseFile har kört klart för att sedan tilldela tiden till variabel CurrentTimeTest. Se figur 9 och 10 för att var tidsfunktioner placeras i koden.

```
1   CASE eLog OF
2
3       E_Logger.INIT:    //resetting some variables
4           bMsgToFile:=FALSE;
5           bMsgToTrace:=FALSE;
6           eLog:= E_Logger.IDLE;
7           eLoggerError := E_LoggerError.NO_ERROR;
8
9       E_Logger.IDLE:  //idling
10          CB_Logger_Trace_Done:=FALSE;
11          IF bMsgToTrace AND CB_Logger_File_Done THEN    //choose write to file or to log?
12              LastTimeTest:=ULINT_TO_TIME(util.GetDateTime());
13              CB_Logger_Trace_Done:=FALSE;
14              IF bMsgToFile THEN
15                  CB_Logger_File_Done:=FALSE;
16              END_IF
17              bMsgToTrace:=FALSE;
18              eLog:= E_Logger.KTRACE;
19          ELSIF bMsgToFile THEN
20              CB_Logger_File_Done:=FALSE;
21
22              CASE eNbrMode OF      //choose if you want log msg in to files with nbr of days or rows
23
24                  E_NbrMode.NBR_OF_DAYS_MODE:
25                      bNbr_Of_Days_Mode:=TRUE;
26                      bNbr_Of_Rows_Mode:=FALSE;
27                      nArraySize:= UPPER_BOUND(aFileLocations,1)-LOWER_BOUND(aFileLocations,1);
28                      CurrentTimestamp:=ULINT_TO_TIME(util.GetDateTime());  // getting the current time of the day
```

Figur 9. En bild på tidsfunktionen med variabeln LastTimeTest i IDLE.

```
121             IF i<=j THEN
122                 i:=1;
123                 j:=1;
124
125                 eLog:= E_Logger.CLOSE_FILE;  //switching to close file state if writting to file is done
126             END_IF
127
128         E_Logger.CLOSE_FILE:
129
130             hCloseFile(xExecute:=TRUE, hFile:= hFile);    // Closing the file
131             IF hCloseFile.xDone THEN
132                 hCloseFile(xExecute:=FALSE);
133             IF nRowCounter < nNbrOfRows THEN
134                 bMsgToFile:=FALSE;
135             ELSIF bNbr_Of_Days_Mode THEN
136                 bMsgToFile:=FALSE;
137             END_IF
138                 IF i<=j THEN
139                     CB_Logger_File_Done:=TRUE;
140                     CurrentTimeTest:=ULINT_TO_TIME(util.GetDateTime());
141                     Time_it_took:=CurrentTimeTest-LastTimeTest;
142                 END_IF
143                 eLog:= E_Logger.IDLE;   //going back to IDLE
144             ELSIF hCloseFile.xError THEN
145                 hCloseFile(xExecute:=FALSE);
146                 eLoggerError := E_LoggerError.CLOSE_FILE_ERROR;
147                 eLog:= E_Logger.ERROR_HANDLING;  //Error-->errorhandling
148             END_IF
```

Figur 10. Variabeln CurrentTimeTest med tidsfunktion i CLOSE_FILE.

Tiderna togs 10 gånger för alla tillstånd och programmet CB_Logger. Sedan beräknades medelvärdet genom att addera ihop tiderna för varje tillstånd och för CB_Logger programmet och dividera med antalet mätningar för att få deras medelvärde. Att ta medelvärde ger en bättre indikation på den genomsnittliga exekveringstiden och gör det möjligt att jämföra prestandan för olika tillstånd och hela programmet. Standardavvikelsen beräknades också med.

3.5 CB_getNumberOfLines

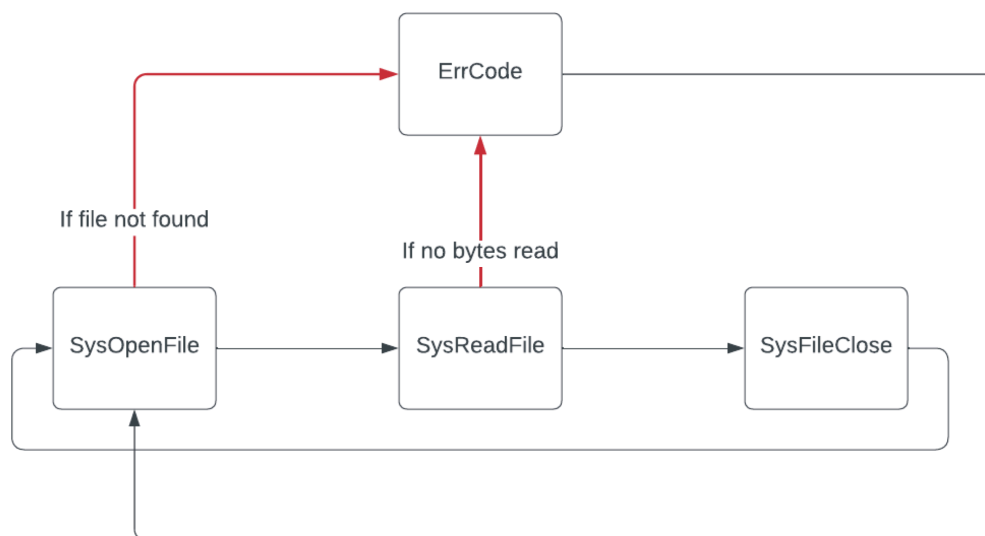
Funktionsblocket CB_getNumberOfLines är byggt för att läsa in en fil och sedan räkna antalet rader som finns i en fil. De antalet rader som räknas från en fil används för att addera ihop med nRowCounter på programmet CB_Logger efter omstart på PLC-system.

Funktionblocket består av en CASE sats med tre tillstånd, vilket är INIT, READ_FILE och COUNTNUMBER_OF_LINES. INIT har uppgiften att återställa visa variabler, READ_FILE använder en program som heter Read_File som öppnar en fil, läser in det till bytes och sedan stänger filen, och COUNTNUMBER_OF_LINES itererar genom alla bytes med ASCII-värdet för en radbrytning (newline), som är 10, för att räkna antalet rader. När den är klar övergår funktionsnlocket CB_getNumberOfLines till tillståndet INIT. Se figur 11 för implementering av CB_getNumberOfLines.

Figur 11. Flödesschema på CB_getNumberOfLines.

3.7 Programmet Read_File

En program som heter Read_File har skapats för funktionsblocket CB_getNumberOfLines som har uppgiften att läsa innehållet i en fil och lagra det i en buffert för vidare bearbetning eller användning. Koden börjar med att öppna filen med SysFileOpen() funktionen och om filen inte hittas tilldelas ett felstatusvärde till en variabel som heter ErrCode. Om filen öppnas läses dess innehåll med SysFileRead() funktionen och lagras i en buffert. Antalet lästa bytes sparas i BytesRead variabel. Lämpliga statusvärden tilldelas till ErrCode beroende på om några byte har lästs eller inte. Slutligen stängs filen med SysFileClose() funktionen. Se figur 12 för implementeringen.



Figur 12. Flödesschema på programmet Read_File.

3.8 Källkritik

Referenserna [1] och [2] kommer från en bok av en erkänd författare inom området PLC. Bolton är känd för sina bidrag till området inom PLC och automatisering. Boltons bidrag är välkända och respekterade inom ämnet , vilket ger hans källor trovärdighet.

Referens [3] är från en officiell webbplats för CODESYS, vilket är en välkänd plattform för PLC-programmering och ger information om CODESYS samt dess fördelar. Referens [4] är också från en officiell webbplats för Keba som förklarar om KeStudio och dess fördelar.

Referenserna [5] och [6] är källor från Cognibotics webbplats som ger detaljerad information om HKM1800. Dessa källor anses vara pålitliga då de kommer från företaget som utvecklar roboten. Referenserna [7] och [8] är källor från Lucidchart och Filezilla som är officiella webbplatser som ger information om sina verktyg.

Referens [9] kommer från en officiell webbplats för SLF4J som ger bra information om sitt API och referensen [10] är från sematext som ger extra bra information om SLF4J och loggning. Sematext är en respekterad källa inom området för loggning och övervakning. Artikeln om “ Understanding Logging Levels: What They Are & How To Use Them” ger bra insikt om olika loggningsnivåer och hur de bör användas, vilket relaterar till ämnet om SLF4J på ett bra nivå.

4 Analys

I detta kapitel diskuteras och analyseras de olika delarna av projektet som utvecklades under programmeringsfasen. Dessutom belyses de utmaningar som uppstod längs vägen och de strategier och lösningar som implementerades för att övervinna dem.

4.1 Utmaningar

Under demotestet uppstod det ett problem med varaibeln nRowCounter. Problemet var att värden för nRowCounter sparades inte efter omstart på PLC eftersom persistenta värden i PLC-systemet kunde inte sparas korrekt vid omstart. För att lösa detta skapades ett funktionsblock för programmet CB_Logger som kan räkna antalet rader i en fil efter omstart av PLC:n.

Funktionsblocket CB_getNumberOfLines skulle läsa in hela filen och omvandla den till bytes, jämföra varje byte med ASCII-värdet för en radbrytning (newline), som är 10, för att räkna antalet rader. Tyvärr fungerade inte detta funktionsblocket som förväntat eftersom det är svårt att läsa in och analysera stora mängder data i ST-programmering, särskilt med tanke på cykeltiden. Funktionsblocket hinner inte att gå igenom varje linje på en fil innan en cykeltid har gått. Men även om funktionsblocket fungerade skulle utvecklaren behöva till exempel vänta 50 sekunder för att räkna antal rader på en fil som innehåller 50000 rader.

Ett sätt som skulle förmodligen gå att räkna antalet rader i en fil är att skapa en funktionsblock som öppnar en annan fil, tar värdet för antalet rader i en fil för loggning, tar bort det gamla värdet för antalet rader, skriva in det senaste antalet rader som finns i filen med loggmeddelanden och sedan stänga ner filen.

En annan utmaning som påträffades var att om användaren ska till exempel logga fem strängar i rad så hoppade över loggningsfunktionen alla strängar förutom det sista strängmeddelandet som sedan skrevs i debug output trace och i fil. Problemet var att om användaren till exempel skulle använda fem metoder för att logga fem meddelande så användes bara sista strängen av den sista metoden av programmet CB_Logger. Lösningen var att skapa vektorer

för CB_Logger som tar in strängar och tilldelar dem till vektorer i metoder både för trace och fil. Varje gång en metod kallas sparas strängar i vektorer och inkrementerar vektorerna med ett för nästa sträng från andra metoder. Efteråt gick det att logga flera meddelande i både trace och fil.

4.2 Stränghantering i PLC

Stränghantering kan leda till prestandaproblem eftersom det kan vara ineffektivt och krävande när det gäller minnesallokering och för att PLC-System ofta har begränsade resurser, vilket kan leda till minnesbrist när strängar används. Att implementera ett program eller funktionsblock i PLC för stränghantering kan var komplicerat och öka komplexiteten i en PLC-kod. Detta kan göra det svårare för användare att underhålla och felsöka programmet.

Fördelarna med stränghantering i PLC inkluderar användningen av funktioner som “CONCAT” för att sammanslå strängar, möjligheten att spara strängar i arrays för effektiv hantering och förmågan att skriva in eller läsa ut från en fil. Dessa funktioner möjliggör en mångsidig användning av textbaserade data i PLC-programmering och gör det möjligt att utföra olika typer av operationer. Genom att utnyttja stränghanteringens potential kan en programmerare skapa mer flexibla och effektiva automationslösningar.

Däremot kan stränghantering vara krävande när det gäller systemresurser och kan leda till minnesbrist och prestandaproblem i PLC-system med begränsade resurser. Många PLC-programmeringsmiljöer erbjuder bara grundläggande stränghanteringsfunkitoner och saknar avancerade funktioner som finns i vanliga programmeringsspråk. Detta kan begränsa dess användbarhet för mer komplexa applikationer.

4.3 SLF4J

Fördelar med att använda SLF4J i PLC:

- Loggningsfunktion som följer SLF4J konventionen ger olika loggningsnivåer och konfigurationsalternativ, vilket ger flexibilitet att anpassa loggningsprocessen efter specifika behov och krav i PLC. Detta gör det möjligt att fokusera på relevanta händelser och information för att underlätta analys och felsökning av ett PLC-program.
- SLF4J gör det enkelt för användaren att lägga till detaljerade loggmeddelande i PLC-kod för att spåra och identifiera potentiella problem och fel i ett PLC-program.

Nackdelar med att använda SLF4J i PLC:

- Att inkludera SLF4J i en PLC-applikation kan leda till ökad minnesanvändning och krav på PLC, vilket kan vara problematiskt. Det är viktigt att överväga resursbehoven och optimeringsstrategierna för att minimera eventuella negativa effekter på PLC:ns prestanda.
- Användare som är nya för SLF4J kan behöva lära sig dess konventioner och bästa sättet att använda loggningsfunktionen effektivt.

4.4 Loggningsfunktion som bibliotekskomponent

Efter att loggningsfunktionen har programmerats klart exporterades mappen till en bibliotekskomponent. Loggningsfunktionen bör vara en bibliotekskomponent istället för att vara öppen källkod av flera skäl. För det första möjliggör en bibliotekskomponent enklare återanvändning och integrering i olika projekt och system. Detta sparar tid och resurser för utvecklare genom att tillhandahålla en förpackad och färdig lösning för loggning. För det andra kan en bibliotekskomponent skydda loggningsfunktion från kodändring annars gör någon ändring på koden som orsakar att loggningsfunktionen inte fungerar längre. Slutligen kan en bibliotekskomponent hanteras och underhållas mer effektivt genom en centraliserad distributionsprocess, vilket säkerställer att alla användare har tillgång till den senaste versionen och eventuella förbättringar.

5 Resultat

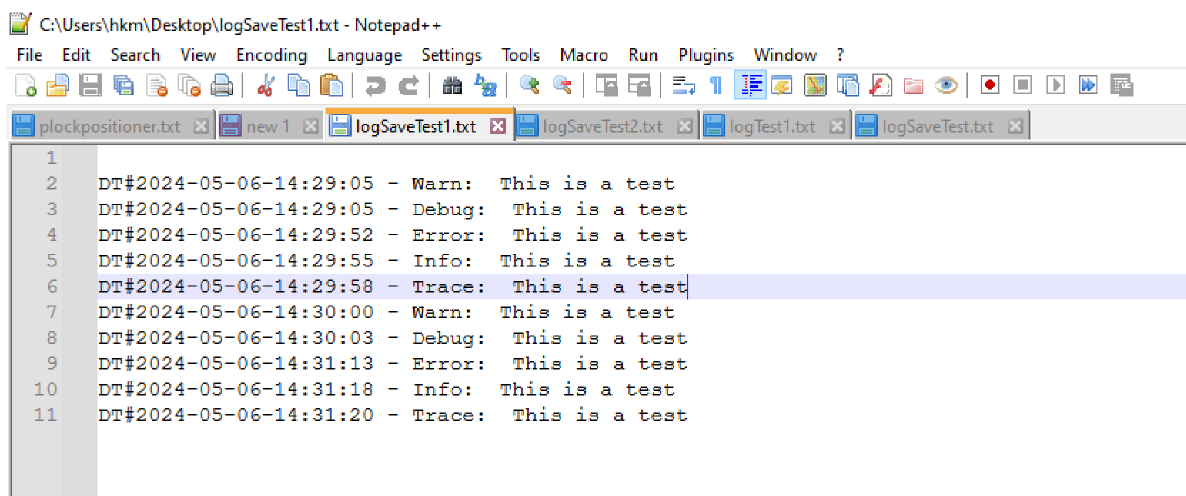
5.1 Funktionell testning

Loggningsfunktionen fungerade som den ska under testet. Loggningsfunktionen kunde skriva in alla loggmeddelande såsom Trace, Info, Debug, Warn och Error på debug output trace. Se figur 13.

```
2484  RepSys   2024-05-06 12:37:39.967| --------
2485  KSysV3   2024-05-06 12:38:34.418|  - Debug:  This is a test
2486  KSysV3   2024-05-06 12:38:36.968|  - Error:  This is a test
2487  KSysV3   2024-05-06 12:38:39.328|  - Info:  This is a test
2488  KSysV3   2024-05-06 12:38:41.668|  - Trace:  This is a test
2489  KSysV3   2024-05-06 12:38:44.588|  - Warn:  This is a test
2490
```

Figur 13. Loggmeddelande i debug output trace

Att överföra loggmeddelande för alla loggnivåer till en fil fungerade under testet med FileZilla. Loggningsfunktionen kunde skriva in loggmeddelande till filer enligt antal rader eller antal dagar som har gått. Se figur 14.



Figur 14. Loggmeddelande skriven i en fil.

Det går även att logga meddelande både i debug output trace och i en fil på samma gång, vilket var förväntat då användaren behöver kunna se loggmeddelanden live på trace och spara dem i en fil på samma gång.

5.2 Prestandatestning

Tiden som mättes för tillståndet KTRACE var korta, vilket var förväntat. Däremot var andra tillstånd längre och programmet CB_Logger tog nästan 100 ms för att skriva in ett loggmeddelande både i trace och i en fil. Ju högre antal meddelande som skrevs i rad i en kod (se exemplet i figur 17 i sida 41) desto större blev tiden för tillståndet WRITE_TO_FILE, KTRACE och programmet CB_Logger. Se tabeller nedan för tiderna i millisekunder, för olika tillstånden och för hela programmet CB_Logger. I testerna skrevs ett, två, tre och flera loggmeddelande i rad för varje skanningscykel.

Antal test för 1 logg	OPEN_FILE (ms)	WRITE_TO_FILE (ms)	CLOSE_FILE (ms)	KTRACE (ms)	TAKE_TIME (ms)	CB_Logger (ms)
1	20	20	21	11	20	92
2	19	21	20	10	21	99
3	20	19	25	10	20	99
4	20	20	23	11	21	100
5	19	19	20	10	21	100
6	20	20	20	10	20	100
7	21	18	19	10	19	99
8	20	21	25	11	21	97
9	20	20	19	10	20	100
10	20	20	20	9	20	100
Medelvärde	19,9	19,8	21,2	10,2	20,3	98,6
Standardavvikelse	0,5676462122	0,9189365835	2,299758441	0,6	0,6749485577	2,503331114

Tabell 3. Tider för alla tillstånd och programmet CB_Logger.

Antal test för 2 loggar i rad	OPEN_FILE (ms)	WRITE_TO_ FILE (ms)	CLOSE_FILE (ms)	KTRACE (ms)	TAKE_TIME (ms)	CB_Logger (ms)
1	20	41	21	20	20	130
2	20	40	20	20	20	129
3	19	39	19	19	19	130
4	19	40	20	20	21	130
5	20	40	24	20	19	131
6	20	40	21	21	20	131
7	20	40	20	20	19	131
8	20	41	20	19	21	131
9	20	39	20	20	20	130
10	21	40	20	20	20	130
Medelvärde	19,9	40	20,5	19,9	19,9	130,3
Standardavvikelse	0,5676462122	0,6666666667	1,354006401	0,5676462122	0,7378647874	0,6749485577

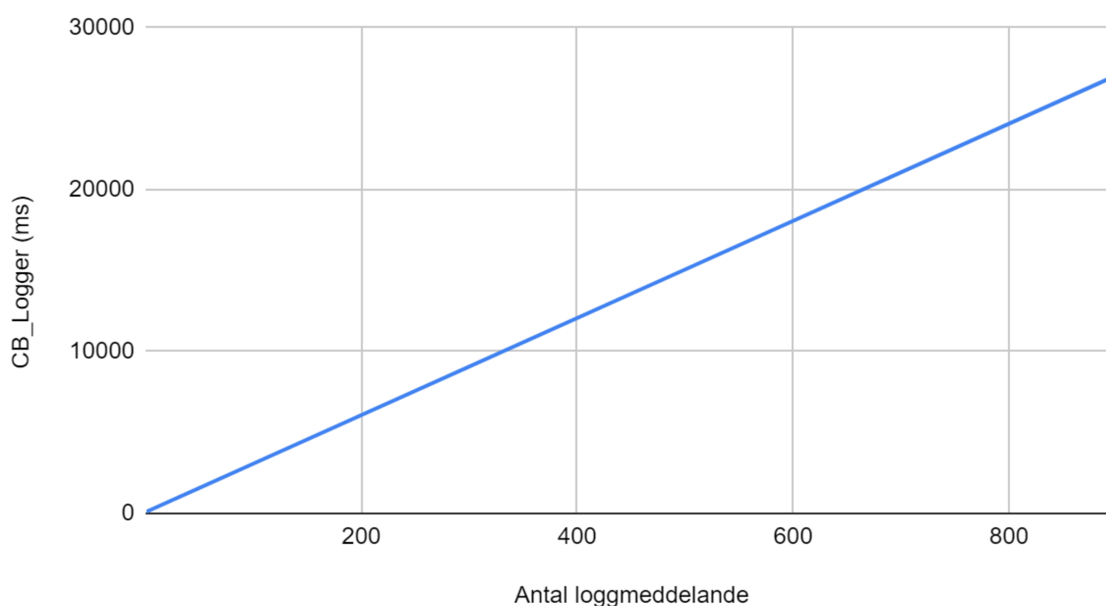
Tabell 2. Tider för alla tillstånd och programmet CB_Logger.

Antal test för 3 loggar i rad	OPEN_FILE (ms)	WRITE_TO_ FILE (ms)	CLOSE_FILE (ms)	KTRACE (ms)	TAKE_TIME (ms)	CB_Logger (ms)
1	20	60	20	30	20	159
2	20	60	19	30	21	160
3	20	61	20	30	20	161
4	19	60	16	31	21	161
5	21	59	19	30	20	160
6	20	58	20	30	20	159
7	20	60	20	28	20	160
8	20	61	19	31	20	160
9	21	60	20	30	19	160
10	20	61	20	30	21	161
Medelvärde	20,1	60	19,3	30	20,2	160,1
Standardavvikelse	0,5676462122	0,9428090416	1,251665557	0,8164965809	0,632455532	0,7378647874

Tabell 3. Tider för alla tillstånd och programmet CB_Logger.

Antal loggmeddelande	CB_Logger (ms)
4	190
5	220
10	370
100	3070
1000	Går ej mätta tiden

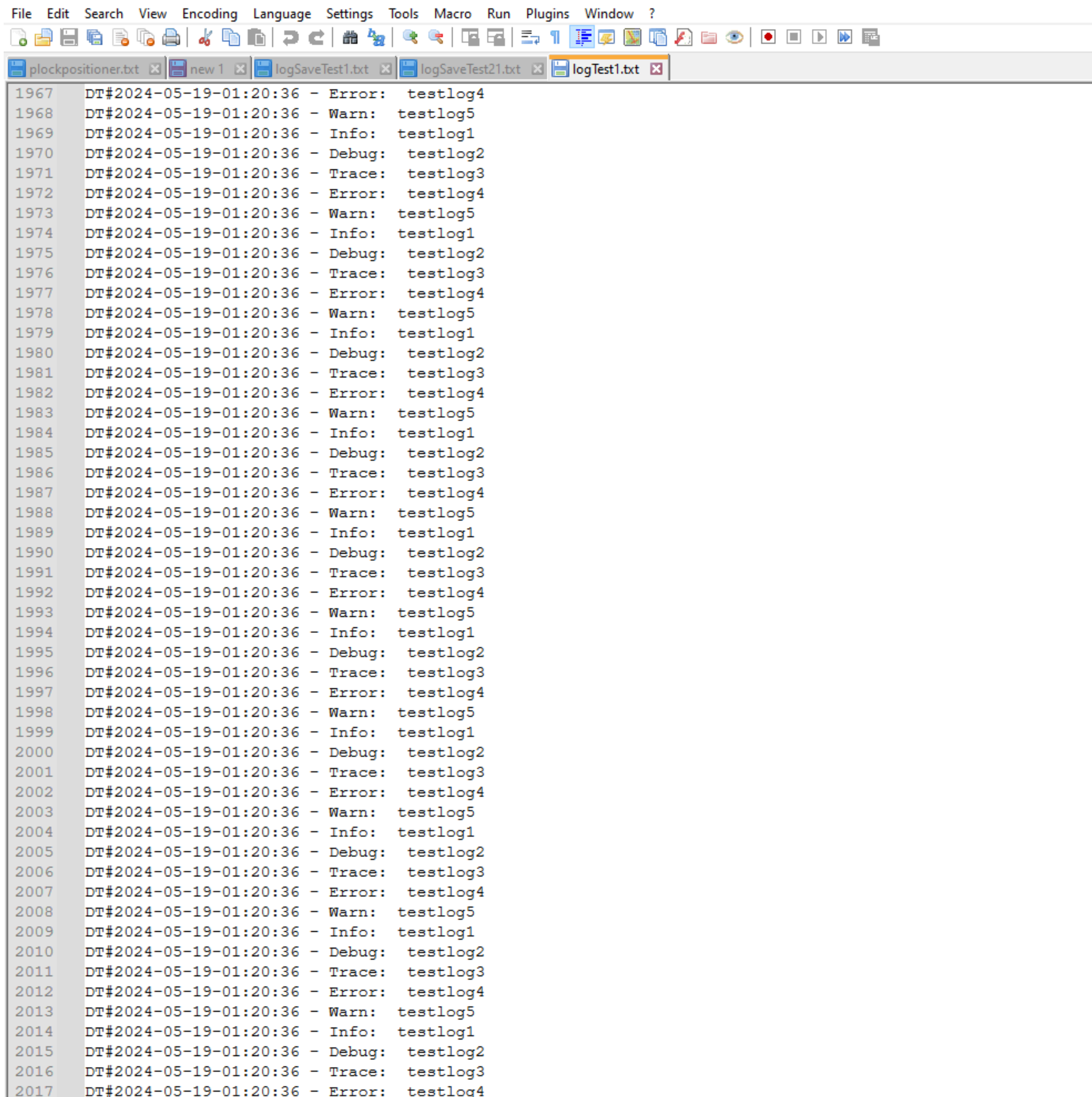
Tabell 4. Tabellen visar tiden för hela programmet CB_Logger ju mer loggmeddelande skrivs i fil och i trace.



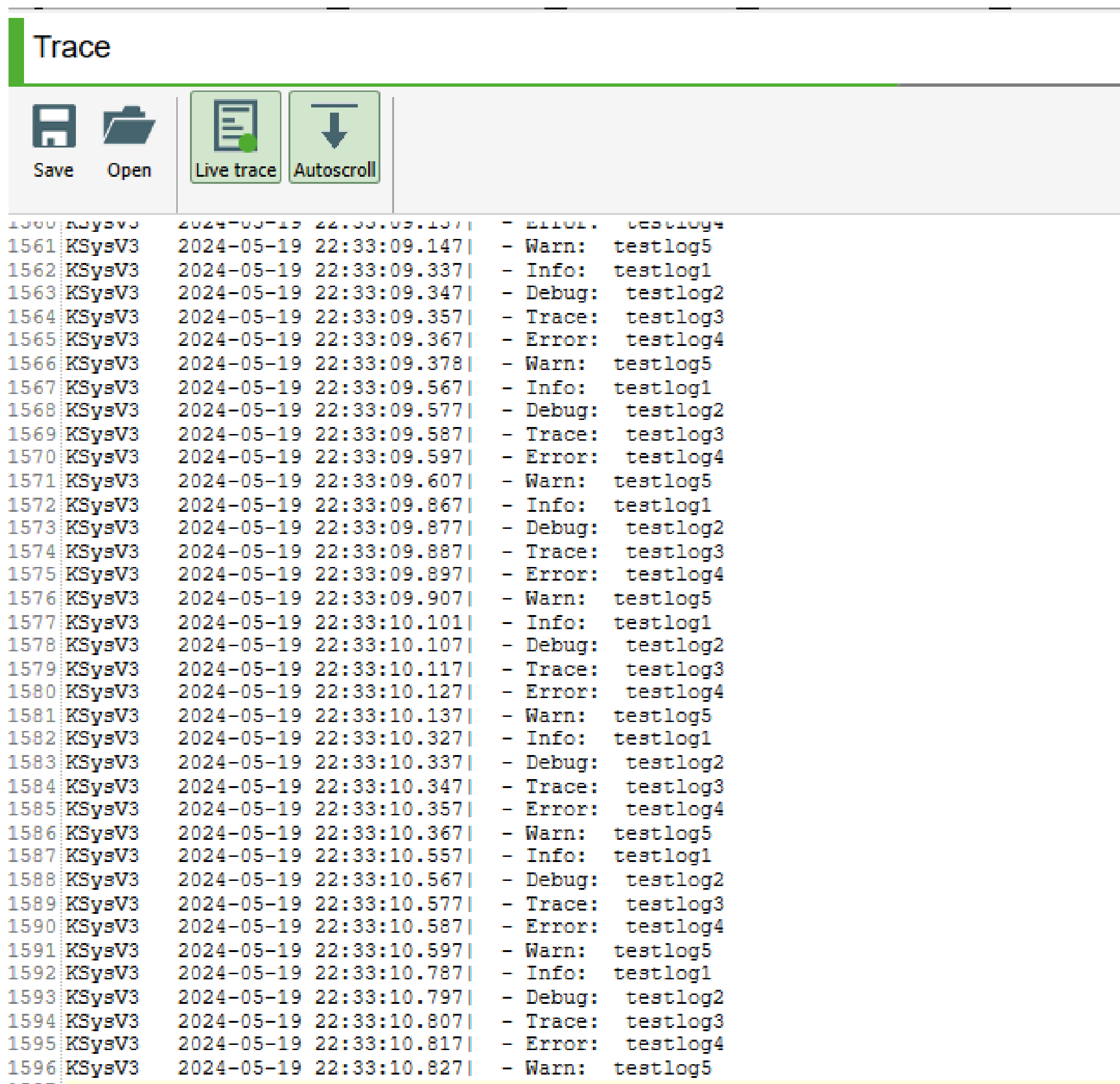
Grafen visar hur tiden för programmet CB_Logger ökar proportionellt ju mer loggmeddelande skrivs i fil och i trace.

Som kan ses i alla tabeller ökades tiden för tillståndet KTRACE med 10 millisekunder i genomsnitt och 20ms i genomsnitt för tillståndet WRITE_TO_FILE varje gång ett extra loggmeddelande skrevs i en fil och i trace. Detta ledde till att programmet CB_Logger tog längre tid att köra klart ju fler loggmeddelande skrevs i rad (se grafen ovan). Tiden mättes för tillstånden NBR_OF_DAYS_MODE och NBR_OF_ROWS_MODE men dem var nästan försumbart då de tog mindre än 1 ms att köra klart.

Loggningsfunktionen testades också för att se hur många loggmeddelande som går att skriva innan programmet CB_Logger kraschar eller slutar att fungera. Loggningsfunktionen kunde skriva många loggmeddelande så länge den inte skriver väldigt många loggmeddelande i rad. Se figur 15 och 16 nedan.



Figur 15. Visar bild på en fil som innehåller mer än 1000 loggmeddelande.



Figur 16. Visar bild på en trace som innehåller mer än 1000 loggmeddelande.

Om användaren försöker logga många meddelande i rad mer än vad vektorerna LogMsgToTrace och LogMsgToFile kan spara i programmet CB_Logger, kommer programmet att sluta fungera och visa exception error meddelandet om att programmet är överbelastat. Vektorerna i programmet CB_Logger kan maximalt innehålla 1000 strängar, annars upphör loggningsfunktionen att fungera. Detta skulle också hända om PLC hade ont om minne för dem två vektorer även om vektorerna inte har nått sina gränser. Se figur 17 och 18 för exception error på över 1000 loggmeddelande.

```
 6      bError : BOOL:=TRUE;
 7      bInfo : BOOL:=TRUE;
 8      bTrace : BOOL:=TRUE;
 9      bWarn : BOOL:=TRUE;
10
11
12      oFileLocations : ARRAY[0..2] OF STRING := ['/home/admin/logTest1.txt','/home/admin/logSaveTest1.txt', '/home/admin/logSaveTest21.txt'];
13
14  END_VAR
15
```

```
982     CB_Logger.Info(LoggMsg:='testlog1',MsgToFile:=TRUE, MsgToTrace:=TRUE);
983     CB_Logger.Debug(LoggMsg:='testlog2',MsgToFile:=TRUE, MsgToTrace:=TRUE);
984     CB_Logger.Trace(LoggMsg:='testlog3',MsgToFile:=TRUE, MsgToTrace:=TRUE);
985     CB_Logger.Error(LoggMsg:='testlog4',MsgToFile:=TRUE, MsgToTrace:=TRUE);
986     CB_Logger.Warn(LoggMsg:='testlog5',MsgToFile:=TRUE, MsgToTrace:=TRUE);
987     CB_Logger.Info(LoggMsg:='testlog1',MsgToFile:=TRUE, MsgToTrace:=TRUE);
988     CB_Logger.Debug(LoggMsg:='testlog2',MsgToFile:=TRUE, MsgToTrace:=TRUE);
989     CB_Logger.Trace(LoggMsg:='testlog3',MsgToFile:=TRUE, MsgToTrace:=TRUE);
990     CB_Logger.Error(LoggMsg:='testlog4',MsgToFile:=TRUE, MsgToTrace:=TRUE);
991     CB_Logger.Warn(LoggMsg:='testlog5',MsgToFile:=TRUE, MsgToTrace:=TRUE);
992     CB_Logger.Info(LoggMsg:='testlog1',MsgToFile:=TRUE, MsgToTrace:=TRUE);
993     CB_Logger.Debug(LoggMsg:='testlog2',MsgToFile:=TRUE, MsgToTrace:=TRUE);
994     CB_Logger.Trace(LoggMsg:='testlog3',MsgToFile:=TRUE, MsgToTrace:=TRUE);
995     CB_Logger.Error(LoggMsg:='testlog4',MsgToFile:=TRUE, MsgToTrace:=TRUE);
996     CB_Logger.Warn(LoggMsg:='testlog5',MsgToFile:=TRUE, MsgToTrace:=TRUE);
997     CB_Logger.Info(LoggMsg:='testlog1',MsgToFile:=TRUE, MsgToTrace:=TRUE);
998     CB_Logger.Debug(LoggMsg:='testlog2',MsgToFile:=TRUE, MsgToTrace:=TRUE);
999     CB_Logger.Trace(LoggMsg:='testlog3',MsgToFile:=TRUE, MsgToTrace:=TRUE);
1000    CB_Logger.Error(LoggMsg:='testlog4',MsgToFile:=TRUE, MsgToTrace:=TRUE);
1001    CB_Logger.Warn(LoggMsg:='testlog5',MsgToFile:=TRUE, MsgToTrace:=TRUE);
1002    CB_Logger.Info(LoggMsg:='testlog1',MsgToFile:=TRUE, MsgToTrace:=TRUE);
1003    CB_Logger.Debug(LoggMsg:='testlog2',MsgToFile:=TRUE, MsgToTrace:=TRUE);
1004    CB_Logger.Trace(LoggMsg:='testlog3',MsgToFile:=TRUE, MsgToTrace:=TRUE);
1005    CB_Logger.Error(LoggMsg:='testlog4',MsgToFile:=TRUE, MsgToTrace:=TRUE);
1006    CB_Logger.Warn(LoggMsg:='testlog5',MsgToFile:=TRUE, MsgToTrace:=TRUE);
1007    CB_Logger.Info(LoggMsg:='testlog1',MsgToFile:=TRUE, MsgToTrace:=TRUE);
```

Figur 17. Visar en bild på hur över 1000 loggmeddelande skulle skrivas i debug output trace och i en fil.

```
 1  sLoggType['- Trace: '] := ' - Trace: ';
 2  sLoggMsg['testlog3']:=LoggMsg['testlog3'];
 3  bMsgToFile[TRUE] := MsgToFile[TRUE];
 4  bMsgToTrace[TRUE] := MsgToTrace[TRUE];
 5  LogMsgtoTrace[k[1701011826]][???]:=CONCAT( sLoggType['- Trace: '], (CONCAT(' ', sLoggMsg['testlog3'])));
 6  LogMsgToFile[i[1411394848]][???]:= CONCAT('$n', CONCAT(CONCAT(TO_STRING(Time_taken[DT#1970-1-1-0:0:0]), sLoggType['- Trace: ']),(CONCAT(' ', sLoggMsg['testlog3']))));
 7  IF CB_Logger_Trace_Done[???] AND bMsgToTrace[TRUE] THEN
 8      k[1701011826]:=k[1701011826]+1;
 9  END_IF
10  IF CB_Logger_File_Done[???] AND bMsgToFile[TRUE] THEN
11      i[1411394848]:=i[1411394848]+1;
12  END_IF[RETURN]
```

h 1

ession	Application	Type	Value	Prepared value	Execution point	Address	Comm...
NonRealTime.FB_CB_logger.nRowCou...	HKM1800_Controller...	UINT	0		Cyclic Monitoring		
NonRealTime.FB_CB_logger.nFileIndex	HKM1800_Controller...	UINT	0		Cyclic Monitoring		
NonRealTime.FB_CB_logger.LogMsgto...	HKM1800_Controller...	ARRAY [1..1000...			Cyclic Monitoring		
NonRealTime.FB_CB_logger.i	HKM1800_Controller...	UDINT	1411394848		Cyclic Monitoring		
NonRealTime.FB_CB_logger.Time_it_t...	HKM1800_Controller...	TIME	T#0ms		Cyclic Monitoring		
NonRealTime.FB_CB_logger.k	HKM1800_Controller...	UDINT	1701011826		Cyclic Monitoring		

/atch 1 | Watch Monitor | Variable Browser | Cross Reference List

ages - Total 0 error(s), 27 warning(s), 302 message(s)

nload | 0 error(s) | 0 warning(s) | 1 message(s)

Device user: Administrator | Last build: 0 21 | Precompile | STOP | Program loaded - EXCEPTION | Program unchanged

Figur 18. Visar en bild på programmet CB_Logger som slutade fungera med exception error som är rödmarkerad.

5.3 Användarupplevelse

Under testningen med HKM1800 upplevdes loggningsfunktionen som användarvänlig och lätt att integrera i PLC-programmet. Den möjliggjorde både live-spårning av loggmeddelanden på debug output trace och sparande av loggar i filer samtidigt, vilket mötte användarnas förväntningar och behov. Inga stopp eller förseningar skedde under testet då det inte var många loggmeddelande som skrevs för varje tillstånd i en CASE sats på ett "pick and place" program. Vanligtvis skriver PLC-Programmerare några loggmeddelande i en IF sats eller CASE sats. Dessutom kördes loggningsfunktionen i en egen task, det vill säga tråd så att den inte påverkar roboten. Om loggningsfunktionen inte körde i en egen task skulle den påverka roboten och leda till förseningar i robotens rörelse för plockning och placering.

6 Slutsats

Resultaten av detta examensarbete visar att loggningsfunktionen har utvecklats och implementerats nästan framgångsrikt för PLC-programmering med structured text. Funktionen möjliggör loggning av meddelande på olika nivåer, inklusive Trace, Debug, Info, Warn och Error, både i filer och i debug output trace. Loggningsfunktionen kan användas av PLC-programmerare för att underlätta övervakning och felsökning av PLC-program samt för att spåra händelser och fel i realtid. Däremot visar prestandatestningen att loggningsfuntkionen kan ha en viss fördröjning om många loggmeddelande skrivs samtidigt i en fil, men detta skulle inte påverka roboten för att den kör på egen task.

Alltså har loggfunktionen förbättrats genom implementering av relevanta loggnivåer och kodspårning för att underlätta felsökning och underhållning av HKM-robotens PLC-kod. Kodspårning har implementerats så att programmeraren kan se olika typer av loggar i debug output trace eller i en fil för att lokalisera anrop i en kod och underlätta identifiering av problem. Sist har loggningsfunktionen gjorts om till en bibliotekskomponent för PLC-programmerare. Det enda problemet är att funktionen inte kan räkna antal rader i en fil på PLC-system vid omstart och detta behövs på grund av persistent value i PLC i Cognibotics fungerar inte som den ska för att spara värde för antal rader. Även om funktionen fungerade skulle det ta väldigt lång tid att räkna klart antal rader för stora filer som till exempel filer med 50000 rader. Alltså ju större är variabeln nNbrOfRows i programmet CB_Logger, desto större är filen och då kommer det ta väldigt lång tid att köra klart funktionsblocket CB_getNumberOfLines för att räkna antal rader.

6.1 Reflektion över etisk aspekt

När det gäller etiska aspekter vid användning av loggningsfunktionen, särskilt inom robotik eller automationsindustrin, är det viktigt att överväga hur data samlas in, lagras och används. En av de främsta etiska hänsyn är sekretess och konfidentiell information. Med ökad användning av loggfunktion inom robotik eller automationsindustri kan känslig information samlas in och lagras i filer. Det är viktigt att säkerställa att information i filer behandlas konfidentiellt och att endast används för dess avsedda ändamål. Eventuell obehörig åtkomst till information måste förhindras för att skydda dess integritet.

6.2 Utvecklingsmöjligheter

I framtiden kan loggningsfunktionen utvecklas genom att integrera den med andra system och plattformar. Det kan inkludera integration med molnbaserade logghanteringsplattformar. Detta kan leda till mer skalbarhet för att hantera större volymer av loggmedelanden och bättre stöd för realtidanalalys och övervakning av PLC-System.

Loggningsfunktionen kan utvecklas vidare med flera avancerade funktioner och förbättringar. Ett exempel på förbättring kan vara att kunna filtrera bort vissa loggmeddelanden för enklare konfiguration och hantering av loggar.

En annan lösning på att kunna räkna antal rader i en fil vid omstart av ett PLC-system är att skriva in antal rader det finns i en fil längst ner på en fil efter varje inskrivning av loggmeddelande. Sedan använda en funktionsblock som bara tar in siffran som finns i sista raden av en fil och addera ihop det med varaiblen nRowCounter som finns i CB_Logger-programmet.

7 Terminologi

PLC	Programmable Logic Controller
Structured Text	Textbaserad PLC-programmeringsspråk
ST	Structured Text
Flödeslogik	Styr sekvensen av steg som driver en process eller system
Loggfunktionalitet	Innebär att skapa och spara händelselogg för övervakning och felsökning
Affinitet	Vilken kategori tillhör ett loggmeddelande
Allvarlighetsgrad	Indikering på hur allvarligt ett loggmeddelande är
Flaggor	Loggnivåer

8 Källförteckning

[1] W. Bolton, Programmable Logic Controllers, 6th ed., Oxford, UK: Newnes, 2015, kap. 1, pp. 1–18.

[2] W. Bolton, Programmable Logic Controllers, 6th ed., Oxford, UK: Newnes, 2015, kap. 1, pp. 167-173.

[3] CODESYS.”Why CODESYS?”. [Elektronisk resurs]. Tillgänglig: https://www.codesys.com/the-system/why-codesys.html. Hämtad: 10-04-2024

[4] Keba. “Kestudio”. [Elektronisk resurs]. Tillgänglig:https://www.keba.com/en/industrial-automation/products/software/kestudio-engineering. Hämtad: 11-04-2024

[5] Cognibotics. “HKM”. [Elektronisk resurs]. Tillgänglig: https://www.cognibotics.com/en/products/hkm. Hämtad: 15-04-2024

[6] Cognibotics. “HKM1800”. [Elektronisk resurs]. Tillgänglig: https://a.storyblok.com/f/251941/x/e0c472773d/hkm-1800-product-sheet-rev-af-downloadable.pdf. Hämtad: 15-04-2024

[7] Lucidchart. “Getting Started in Lucidchart: The Basics”. [Elektronisk resurs]. Tillgänglig: https://www.lucidchart.com/blog/getting-started-in-lucidchart#inspirationwhy. Hämtad: 11-04-2024

[8] FileZilla. “Overview”. [Elektronisk resurs]. Tillgänglig: https://filezilla-project.org/. Hämtad: 16-04-2024

[9] SLF4J. “SLF4J user manual”. [Elektronisk resurs]. Tillgänglig: https://www.slf4j.org/manual.html. Hämtad: 17-04-2024

[10] Sematext. “ Understanding Logging Levels: What They Are & How To Use Them”. [Elektronisk resurs]. Tillgänglig: https://sematext.com/blog/logging-levels/. Hämtad: 17-04-2024

9 Appendix

CB_Logger (PRG)

```
Unset
PROGRAM CB_Logger

VAR
	bMsgToTrace: BOOL;
	bMsgToFile: BOOL;
	bTrace: BOOL;
	bWrite_To_File:BOOL:=TRUE;


	sLoggMsg:STRING(255);  //log msg
	sErrorMsg:STRING;
	StringFilemsg: STRING(255);   //msg to file

	eLog:  E_Logger:=E_Logger.INIT;

	hCloseFile: file.Close();
	hTimeFile: file.GetTime();
	hWriteToFile: file.Write();
	hOpenFile: file.Open();
	hReadFile: file.Read();
	FileSize: UDINT;
	fileContent: STRING;     // Buffer for file content
	charIndex: UDINT;        // Index for iterating through fileContent

	Time_taken: DT;
	CurrentTimestamp: TIME;
	LastsavetTimestamp: TIME;

	nFileIndex : UINT := 0;
	nRowCounter : UINT := 0;

	hFile : CAA.HANDLE;
	szFile: CAA.SIZE;
	pFile: CAA.PVOID;

	sLoggType : STRING := '';

	FileMode : FILE.MODE;

	nArraySize: DINT;
	LogMsgToFile: ARRAY [1..1000] OF STRING(255);
	LogMsgtoTrace: ARRAY [1..1000] OF STRING(255);
	i: UDINT:=1;
	k:UDINT:=1;
	j: UDINT:=1;
	bNbr_Of_Rows_Mode: BOOL;
```

```
        bNbr_Of_Days_Mode: BOOL;
        tonTimer: TON;  // Declare a TON timer
        CB_Logger_Trace_Done:BOOL:=TRUE;
        CB_Logger_File_Done:BOOL:=TRUE;

END_VAR
VAR_INPUT

        nNbrOfRows : UINT := 100;     //set the limit for number of rows
allowed in a file
        nNBR_OF_DAYS: TIME:=T#2M;    //set the number days needed to write
to file before switching to the next file
        eNbrMode : E_NbrMode:= E_NbrMode.NBR_OF_ROWS_MODE;  // Choose if you
want to write messages to the same file depending on the ammount of rows or
number of days allowed
        aFileLocations : ARRAY[0..2] OF STRING;     // files need to be
created before use. You can also increase the array size inorder to add
more files in to the array
END_VAR
VAR_OUTPUT
        eLoggerError : E_LoggerError;
END_VAR
```

Unset

```
CASE eLog OF

        E_Logger.INIT:    //resetting some variables
                bMsgToFile:=FALSE;
                bMsgToTrace:=FALSE;
                eLog:= E_Logger.IDLE;
                eLoggerError := E_LoggerError.NO_ERROR;

        E_Logger.IDLE:  //idling
                IF bMsgToTrace AND CB_Logger_File_Done THEN    //choose write
to file or to log?
                        CB_Logger_Trace_Done:=FALSE;
                        IF bMsgToFile THEN
                                CB_Logger_File_Done:=FALSE;
                        END_IF
                        bMsgToTrace:=FALSE;
                        eLog:= E_Logger.KTRACE;
                ELSIF bMsgToFile THEN
                        CB_Logger_File_Done:=FALSE;

                        CASE eNbrMode OF      //choose if you want log msg in
to files with nbr of days or rows

                                E_NbrMode.NBR_OF_DAYS_MODE:
```

```
                                bNbr_Of_Days_Mode:=TRUE;
                                bNbr_Of_Rows_Mode:=FALSE;
                                nArraySize:=
UPPER_BOUND(aFileLocations,1)-LOWER_BOUND(aFileLocations,1);

CurrentTimestamp:=ULINT_TO_TIME(util.GetDateTime());  // getting the
current time of the day
                                IF (CurrentTimestamp -
LastsavetTimestamp) >=nNBR_OF_DAYS THEN
                                        IF nFileIndex >= nArraySize THEN
//if we reached the last the last file then go back to the first file
                                                nFileIndex  := 0;  //
reseting back to the first file
                                                LastsavetTimestamp:=
DT_TO_TIME(Time_taken); //getting the date and time for when the first open
and write occured in the next file next
                                        ELSE
                                                nFileIndex := nFileIndex +
1; //swiching to the next file
                                                LastsavetTimestamp:=
DT_TO_TIME(Time_taken);  //getting the date and time for when the first
open and write occured in the next file next
                                        END_IF
                                        FileMode := FILE.MODE.MWRITE;
//overwrite the next file
                                ELSE
                                        FileMode := FILE.MODE.MAPPD;  //
else continue adding msg to the current file
                                END_IF

                                eLog:= E_Logger.OPEN_FILE; //Swtiching to
OPEN_FILE state

                        E_NbrMode.NBR_OF_ROWS_MODE:
                                bNbr_Of_Rows_Mode:=TRUE;
                                bNbr_Of_Days_Mode:=FALSE;
                                nArraySize:=
UPPER_BOUND(aFileLocations,1)-LOWER_BOUND(aFileLocations,1);
                                IF nRowCounter >=nNbrOfRows THEN    //if
the nbr of rows exceeds the limit for ammount of rows allowed   test
SIZEOF(myArray)
                                        nRowCounter := 0;
                                        IF nFileIndex >= nArraySize THEN
//if we reached we last the last file then go back to the first file
                                                nFileIndex  := 0; //
reseting back to the first file
                                        ELSE
                                                nFileIndex := nFileIndex +
1;  //swtiching to the next file
                                        END_IF
                                        FileMode := FILE.MODE.MWRITE;
                                ELSE
                                        FileMode := FILE.MODE.MAPPD;
                                END_IF
```

```
                                eLog:= E_Logger.OPEN_FILE; //Swtiching to
OPEN_FILE state
                    END_CASE
                END_IF

        E_Logger.OPEN_FILE:
                hOpenFile(xExecute:=TRUE,
sFileName:=aFileLocations[nFileIndex], eFileMode := FileMode);
//opening the file
                IF hOpenFile.xDone THEN
                    hFile := hOpenFile.hFile;
                    hOpenFile(xExecute:=FALSE);
                    eLog:= E_Logger.TAKE_TIME;  //switching to TAKE_TIME
state if opening file worked
                ELSIF hOpenFile.xError THEN
                    hOpenFile(xExecute:=FALSE);
                    eLoggerError := E_LoggerError.OPEN_FILE_ERROR;
//changing the error to open file error
                    eLog:= E_Logger.ERROR_HANDLING;  // switching to
error_handling state
                END_IF


        E_Logger.TAKE_TIME:
                hTimeFile(xExecute:=TRUE,
sFileName:=aFileLocations[nFileIndex]);    //getting the date and time of
when the file was modified
                IF hTimeFile.xDone THEN
                    Time_taken:=hTimeFile.dtLastModification;  //assigning
the date and time to another variable
                    hTimeFile(xExecute:=FALSE);
                    eLog:= E_Logger.WRITE_TO_FILE;  //switching to
WRITE_TO_FILE state if done
                ELSIF hTimeFile.xError THEN
                    hTimeFile(xExecute:=FALSE);
                    eLoggerError := E_LoggerError.TAKE_TIME_ERROR;
                    eLog:= E_Logger.ERROR_HANDLING;
//Error-->errorhandling
                END_IF

        E_Logger.WRITE_TO_FILE:
                hWriteToFile(xExecute:=TRUE, xAbort := FALSE, udiTimeOut :=
9000, hFile:=hFile, pBuffer:= ADR(LogMsgToFile[j]), szSize:=
LEN(LogMsgToFile[j])); //writing the msg to file
                IF hWriteToFile.xDone THEN
                    j:=j+1;
                    nRowCounter:=nRowCounter+1;  //The number of rows in
text file increases when more msg is written to the file
                    IF nRowCounter >=nNbrOfRows AND bNbr_Of_Rows_Mode THEN
                        eLog:= E_Logger.CLOSE_FILE;
                    END_IF
                    hWriteToFile(xExecute:=FALSE);
                    hWriteToFile(xAbort := TRUE);
                ELSIF hWriteToFile.xError THEN
```

```
                    hWriteToFile(xExecute:=FALSE);
                    hWriteToFile(xAbort := TRUE);
                    eLoggerError := E_LoggerError.WRITE_TO_FILE_ERROR;
                    eLog:= E_Logger.ERROR_HANDLING;
//Error-->errorhandling
             END_IF
             IF i<=j THEN
                    i:=1;
                    j:=1;
                    eLog:= E_Logger.CLOSE_FILE;  //switching to close file
state if writting to file is done
             END_IF

      E_Logger.CLOSE_FILE:
             hCloseFile(xExecute:=TRUE, hFile:= hFile);    // Closing the
file
             IF hCloseFile.xDone THEN
                    hCloseFile(xExecute:=FALSE);
                    IF nRowCounter < nNbrOfRows THEN
                           bMsgToFile:=FALSE;
                    ELSIF bNbr_Of_Days_Mode THEN
                           bMsgToFile:=FALSE;
                    END_IF
                    IF i<=j THEN
                           CB_Logger_File_Done:=TRUE;
                    END_IF
                    eLog:= E_Logger.IDLE;   //going back to IDLE
             ELSIF hCloseFile.xError THEN
                    hCloseFile(xExecute:=FALSE);
                    eLoggerError := E_LoggerError.CLOSE_FILE_ERROR;
                    eLog:= E_Logger.ERROR_HANDLING;
//Error-->errorhandling
             END_IF

      E_Logger.KTRACE:
             bTrace:=KTrace(Str :=LogMsgtoTrace[j]);   //sending msg to
Trace
             IF NOT bTrace THEN
                    bMsgToTrace:=FALSE;
                    eLog:= E_Logger.ERROR_HANDLING;
//Error-->errorhandling
             END_IF
             j:=j+1;
             IF k<=j THEN
                    IF bTrace THEN    //if sending msg to trace was
succesfull
                           bTrace:=FALSE;
                           bMsgToTrace:=FALSE;
                           k:=1;
                           j:=1;
                           CB_Logger_Trace_Done:=TRUE;
                           eLog:= E_Logger.IDLE;  //going back to IDLE if
sending msg to trace was succesfull
                    END_IF
```

```
            END_IF

        E_Logger.ERROR_HANDLING:
            CASE eLoggerError OF
                E_LoggerError.KTRACE_ERROR:

                E_LoggerError.OPEN_FILE_ERROR:
                    sErrorMsg:='ERROR: Logger function used
incorrectly or something went wrong, try again.';

                E_LoggerError.TAKE_TIME_ERROR:
                    sErrorMsg:='ERROR: Logger function used
incorrectly or something went wrong, try again.';

                E_LoggerError.WRITE_TO_FILE_ERROR:
                    sErrorMsg:='ERROR: Logger function used
incorrectly or something went wrong, try again.';
                    eLog:= E_Logger.FLUSH_WRITE_BUFFER;

                E_LoggerError.CLOSE_FILE_ERROR:
                    sErrorMsg:='ERROR: Logger function used
incorrectly or something went wrong, try again.';

            END_CASE

                KTrace(Str :=sErrorMsg);  //sending Error msg to Trace
                eLog:= E_Logger.INIT;
END_CASE
```

CB_Logger.Debug (METHOD)

```
Unset
METHOD Debug : BOOL //This method is used for messages that are primarily
useful for debugging purposes.

VAR_INPUT
       sLoggMsg:STRING(255);
       bMsgToFile: BOOL:=FALSE;
       bMsgToTrace: BOOL;
END_VAR

sLoggType := ' - Debug: ';
sLoggMsg:=LoggMsg;
bMsgToFile := MsgToFile;
bMsgToTrace := MsgToTrace;
LogMsgtoTrace[k]:=CONCAT( sLoggType,(CONCAT(' ', sLoggMsg)));
LogMsgToFile[i]:= CONCAT('$n', CONCAT(CONCAT(TO_STRING(Time_taken),
sLoggType),(CONCAT(' ', sLoggMsg))));
IF CB_Logger_Trace_Done AND bMsgToTrace THEN
       k:=k+1;
END_IF
IF CB_Logger_File_Done AND bMsgToFile THEN
       i:=i+1;
END_IF
```

CB_Logger.Info (METHOD)

```
Unset
METHOD Info : BOOL //This method is used to provide informational messages
about the application's execution.

VAR_INPUT
       sLoggMsg:STRING(255);
       bMsgToFile: BOOL:=FALSE;
       bMsgToTrace: BOOL;
END_VAR

LogMsgtoTrace[k]:=CONCAT( sLoggType,(CONCAT(' ', sLoggMsg)));
LogMsgToFile[i]:= CONCAT('$n', CONCAT(CONCAT(TO_STRING(Time_taken),
sLoggType),(CONCAT(' ', sLoggMsg))));
IF CB_Logger_Trace_Done AND bMsgToTrace THEN
       k:=k+1;
END_IF
IF CB_Logger_File_Done AND bMsgToFile THEN
       i:=i+1;
END_IF
```

CB_Logger.Error (METHOD)

Unset

```
METHOD Error : BOOL //This method is used to provide messages that indicate
serious errors that may require attention or intervention.

VAR_INPUT
       sLoggMsg:STRING(255);
       bMsgToFile: BOOL:=FALSE;
       bMsgToTrace: BOOL;
END_VAR

sLoggType := ' - Error: ';
sLoggMsg:=LoggMsg;
bMsgToFile := MsgToFile;
bMsgToTrace := MsgToTrace;
LogMsgtoTrace[k]:=CONCAT( sLoggType,(CONCAT(' ', sLoggMsg)));
LogMsgToFile[i]:= CONCAT('$n', CONCAT(CONCAT(TO_STRING(Time_taken),
sLoggType),(CONCAT(' ', sLoggMsg))));
IF CB_Logger_Trace_Done AND bMsgToTrace THEN
       k:=k+1;
END_IF
IF CB_Logger_File_Done AND bMsgToFile THEN
       i:=i+1;
END_IF
```

CB_Logger.Trace (METHOD)

Unset

```
METHOD Trace : BOOL // This mehtod is used for detailed diagnostic
information.

VAR_INPUT
       sLoggMsg:STRING(255);
       bMsgToFile: BOOL:=FALSE;
       bMsgToTrace: BOOL;
END_VAR

LogMsgtoTrace[k]:=CONCAT( sLoggType,(CONCAT(' ', sLoggMsg)));
LogMsgToFile[i]:= CONCAT('$n', CONCAT(CONCAT(TO_STRING(Time_taken),
sLoggType),(CONCAT(' ', sLoggMsg))));
IF CB_Logger_Trace_Done AND bMsgToTrace THEN
       k:=k+1;
END_IF
IF CB_Logger_File_Done AND bMsgToFile THEN
       i:=i+1;
END_IF
```

CB_Logger.Warn (METHOD)

```
Unset
METHOD Warn : BOOL //This method is used to provide messages that indicate
potential issues or situations that might lead to errors.


VAR_INPUT
       sLoggMsg:STRING(255);
       bMsgToFile: BOOL:=FALSE;
       bMsgToTrace: BOOL;
END_VAR

LogMsgtoTrace[k]:=CONCAT( sLoggType,(CONCAT(' ', sLoggMsg)));
LogMsgToFile[i]:= CONCAT('$n', CONCAT(CONCAT(TO_STRING(Time_taken),
sLoggType),(CONCAT(' ', sLoggMsg))));
IF CB_Logger_Trace_Done THEN
       k:=k+1;
END_IF
IF CB_Logger_File_Done THEN
       i:=i+1;
END_IF
```

CB_GetNumber_Of_Lines (FB)

```
Unset
FUNCTION_BLOCK CB_getNumberOfLines
VAR_INPUT
       aFileLocations : STRING;
END_VAR
VAR_OUTPUT
       done : BOOL := FALSE;
       lineCount: UINT;
END_VAR
VAR
       tempString : STRING;

       newline: __XWORD := 10; // Represents a newline character

       eLinecounter:  E_Linecounter:=E_Linecounter.INIT;

       charIndex: UDINT;        // Index for iterating through fileContent


       pSysFileResult : POINTER TO SysFile.RTS_IEC_RESULT;
       stFileHandle : SysFile.RTS_IEC_HANDLE;
       sReadString : STRING;
       pBuffer : POINTER TO BYTE := ADR(sReadString);
       nBufferSize : __XWORD;
       nSize : __XWORD;
```

```
        aData :  ARRAY[0..Read_File.MAX_BUFFER] OF __XWORD;

END_VAR



CASE eLinecounter OF
        E_Linecounter.INIT:
                linecount:=0;
                done:=FALSE;
                eLinecounter:= E_Linecounter.READ_FILE_SIZE;

        E_Linecounter.READ_FILE_SIZE:
                Read_File(FileName := aFileLocations);
                IF Read_File.Done THEN
                        aData := Read_File.BytesRead;
                        eLinecounter := E_Linecounter.COUNTLINES_IN_FILE;
                END_IF


                E_Linecounter.COUNTLINES_IN_FILE:
                IF NOT charIndex=(SIZEOF(aData)) THEN
                        charIndex:=charIndex+1;
                        IF aData[charIndex] = newline THEN
                lineCount := lineCount + 1;
                END_IF
                ELSIF charIndex=(SIZEOF(aData)) THEN
                        done:=TRUE;
                        charIndex:=0;
                        eLinecounter := E_Linecounter.INIT;
                END_IF


END_CASE
```