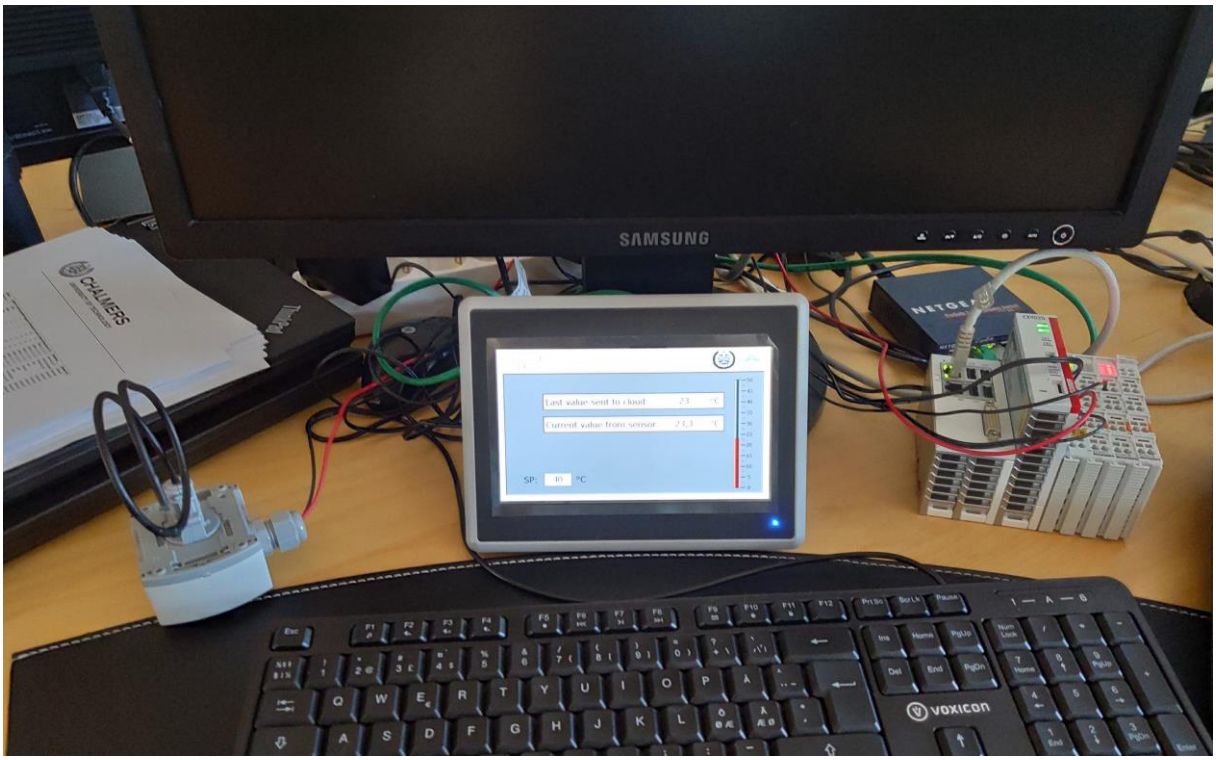
CHALMERS



PLC som en del av Internet of Things

Utveckling av ett kommunikationsbibliotek baserat på MQTT protokollet

Examensarbete inom högskoleingenjörsprogrammet Mekatronik

ANTON ANDERSSON
JOEL OLSSON

Institutionen för Elektroteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2020

EXAMENSARBETE 2020

PLC som en del av Internet of Things

Utveckling av ett kommunikationsbibliotek baserat på MQTT protokollet

ANTON ANDERSSON
JOEL OLSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Institutionen för elektroteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2020

PLC som en del av Internet of Things
Utveckling av ett kommunikationsbibliotek baserat på MQTT protokollet

Examensarbete inom högskoleingenjörsprogrammet Mekatronik
ANTON ANDERSSON
JOEL OLSSON

Handledare: Jacob Thorsell, Acobia AB
Examinator: Veronica Olesen, Institutionen för elektroteknik, Chalmers tekniska högskola

Institutionen för elektroteknik
Chalmers tekniska högskola
SE-412 96 Göteborg
Telefon: +46-(0)31 772 10 00

Förstasida: Testrigg som användes vid konceptvalidering.

[Egen bild]
Göteborg, Sverige 2020

Förord

Detta examensarbete är utfört av två mekatronikstudenter på Chalmers tekniska högskola under institutionen för elektronik. Arbetet genomfördes åt Acobia AB under våren 2020 och omfattade 15 högskolepoäng per student. Projektet innefattade ämnen inom mekatronik som programmering av styrenheter, signalhantering och nätverkskommunikation.

Vi vill framför allt tacka vår företagshandledare Jacob Thorsell för allt stöd vi fått under projektets gång. Vi vill även tacka de övriga anställda på Acobia AB som hjälpt oss under vår tid på kontoret. Slutligen vill vi ge ett stort tack till vår handledare och examinator Veronica Olesen för all feedback som hjälpt oss under projektet och utvecklingen av denna rapport.

Anton Andersson och Joel Olsson, juni 2020

Sammanfattning

Beckhoff Automation har med sin nya mjukvaruuppdatering till TwinCAT 3 gjort det möjligt för sina kontrollenheter (PLC) att kommunicera med nätverksprotokollet MQTT. Automationsföretaget Acobia AB vill med Beckhoffs nya verktyg framställa en företagsstandard för kommunikation mellan Beckhoff PLC:er och webapplikationer på molntjänsten Microsoft Azure.

Denna rapport beskriver det projekt som genomfördes för att konstruera ett funktionsblocksbibliotek som ska bygga grunden för Acobias nya kommunikationsstandard. Arbetet bestod av tre delar: framtagning av en metod för användning av Beckhoffs nya verktyg, konstruktion av bibliotek samt tillämpning av bibliotek som konceptvalidering.

Det resulterande biblioteket är en samling funktionsblock av tre typer: ett anslutningsblock, ett publiceringsblock för varje datatyp, samt tre prenumerationsblock för olika metoder att hantera meddelanden. Funktionsblocksbiblioteket utvärderades praktiskt via en testrigg för att påvisa funktion i realtid. Biblioteket avgränsades till Acobias egna JSON-baserade meddelandestruktur. Funktionsblocken fungerar bra i testrigg och uppnår de givna mål och krav.

Under projektet utvärderades olika metoder för att motverka den begränsade nivån av säkerhet som medföljer kommunikation via internet. Metoder som kryptering, dynamisk nyckeluppdatering samt serververifiering studerades. Acobia var nöjda med en lösning som baserades på TLS då en lösning med större säkerhet kräver större uppdateringar av flera delar i systemet.

Abstract

In a recent update for TwinCAT 3, Beckhoff Automation created the ability to connect their control units (PLC) to a cloud server with the communication protocol MQTT. The automation company Acobia AB wanted to create a set of standardized tools for communication between their Beckhoff systems and their web applications on the cloud service Microsoft Azure.

This paper describes the project to create a library of function blocks to build the foundation for Acobias new communication standard. The project was split into three parts: Creating a method on how to integrate Beckhoffs new tools with the existing systems, developing a function block library, and presenting an example of how all the parts of the library work within a validation project.

The resulting library contains three types of blocks: a connection block, a publish-block for each datatype and three subscription-blocks for different methods of handling messages. The library was tested with a test rig containing a Beckhoff control unit, a temperature sensor, an interface unit and a mobile phone application. The library is constrained to Acobia's internal communication protocol but worked as intended with the goals and requirements given.

Different security methods were discussed during the project because of the increased security concerns connecting devices over the internet. Methods like encryption, dynamic token updates, TLS and server authentication were discussed. Acobia was satisfied with a TLS based solution because bigger updates to the rest of the system would be required for more security to be implemented in the future.

Innehåll

Förkortningar

API. Application Programming Interface

Azure. Microsofts molntjänst

DNS. Domain Name System

FIFO. First in first out

I/O. Input/Output

IoT. Internet of things

JSON. JavaScript object notation

JWT. JSON Web Token

M2M. Maskin till Maskin

MQTT. Message Queueing Telemetry Transport

PLC. Programmable logic controller

REST. Representational State Transfer

SAS. Shared Access Signature

SCADA. Supervisory Control And Data Acquisition

TFS. Team foundation server

TLS. Transport Layer Security

VSC. Visual Studio Code

1. Inledning

Världen blir mer och mer uppkopplad mot internet. Nya smarta enheter utvecklas i flera branscher. Smarta enheter innebär att enheterna på något sätt har en processor och är uppkopplade mot ett nätverk. Enheterna har möjligheten att ge information och kommunicera och har implementerats för att till exempel avläsa temperatur i kylskåp, styra lampor och ventilation. De smarta enheterna har förändrat hur både privatpersoners och industriers vardag ser ut. För att garantera att alla dessa enheter skickar sin data till rätt mottagare krävs ett system som kan hantera alla typer av enheter. När varje enhet är uppkopplad blir det fort mycket information som måste hanteras. Alla meddelanden behöver då likt ett postsystem passera en sortering och skickas vidare till önskad mottagare. Ett sätt att hantera alla meddelanden är genom att implementera virtuella samlingsplatser, så kallade Internet of things (IoT) hubbar.

En liknelse kan då ses som att IoT-hubben är en lokal för ett postkontor där sorteringen ska skötas. Men lokalen är inte tillräckligt för att hela flödet ska fungera, en infrastruktur med adressering måste användas. Ett enkelt meddelandeprotokoll kallat MQTT kan då användas för att rätt information ska hamna hos rätt enhet.

Projektet innefattar implementering av ett programbibliotek som möjliggör att industriella datorer kommunicerar korrekt med IoT-hubbar och andra uppkopplade applikationer.

1.1. Bakgrund

Acobia AB har utvecklat mjukvara som ger möjligheten att kommunicera mellan webapplikationer och PLC:er. Innan projektet påbörjades skickades meddelanden genom flera delsystem. På grund av den komplexa meddelandehanteringen i lösningen är den inte alltid optimal. Acobia önskade därför att förenkla och effektivisera kommunikationsvägen.

System som användes för meddelandena hantering innan projektet påbörjades hade många steg och kräver ett antal konverteringar av den information som ska skickas. PLC-tillverkaren Beckhoff har lanserat en ny generation PLC som stödjer kommunikationsprotokollet Message Queueing Telemetry Transport, MQTT, vilket ger möjligheten att förkorta informationskedjan markant genom kommunikation direkt med en molntjänst.

1.2. Syfte

Projektet ska göra det möjligt för PLC:er att kommunicera direkt med molntjänsten Microsoft Azure utan att gå via operatörsgränssnittet och på så sätt förenkla informationsflödet till IoT-hubben. Uppkopplingen kommer ge möjligheten att styra och kommunicera med PLC direkt från applikationer, vilket förenklar felsökning, avläsning samt programmering av PLC-enheterna. Resultatet kan vara det första steget till att på ett nytt sätt effektivisera dagens styrtekniker och göra installation av styrsystem enklare och billigare.

1.3. Avgränsningar

- Den kommunikation som projektet ska producera är begränsad av Acobias existerande application programming interface, API, av typen representational state transfer, REST.
- Protokollen kommer endast hantera de typer av meddelanden som är existerande i IoT systemet:
    - Skicka taggars värden
    - Ta emot taggars värden
    - Skicka larmstatus
    - Kvittera och återställa larm
- Projektprodukten kommer inrikta sig på Beckhoffs produkter med TwinCat3 mjukvaran eftersom dessa enheter har MQTT kompabilitet.
- Produkten kommer endast stödja kommunikation mot Microsofts molntjänst Azure.

1.4. Precisering av mål och frågeställning

Målen med projektet är att konstruera ett programbibliotek för Beckhoff PLC:er som ska underlätta uppkoppling och kommunikation direkt med Acobias Azure applikationer. Programbiblioteket ska med hjälp av MQTT kunna skicka och ta emot meddelanden med Acobias kommunikationsstruktur.

Projektets delmål:

1. Skapa en fungerande anslutningsmetod mellan PLC och Azure.
2. Etablera enkel tvåvägskommunikation mellan PLC och Azure.
3. Konstruera ett funktionsblocksbibliotek för anslutning och kommunikation för Acobias kommunikationsstruktur.
4. Genomföra en konceptvalidering av kommunikationsbiblioteket.

Projektets frågeställningar:

- Vad krävs för att kommunicera och tolka taggar mellan Beckhoffs PLC och Azure med MQTT?
- Hur bör protokollen struktureras? Vilka standarder finns idag som bör implementeras i projektet
- Hur ska protokollen konstrueras så att driftsättning kan göras enklast på nya PLC?
- Hur kan kommunikationsprotokollen enklast felsökas, vilka felsignaler ska existera?
- Hur ska felsökning underlättas under utveckling?

- Uppkoppling till IoT-hubben kan leda till risk för intrång i systemet eftersom kommunikation inte är över ett lokalt nätverk. Hur ska detta hanteras för att systemet ska vara hållbart ur säkerhetssynpunkt?

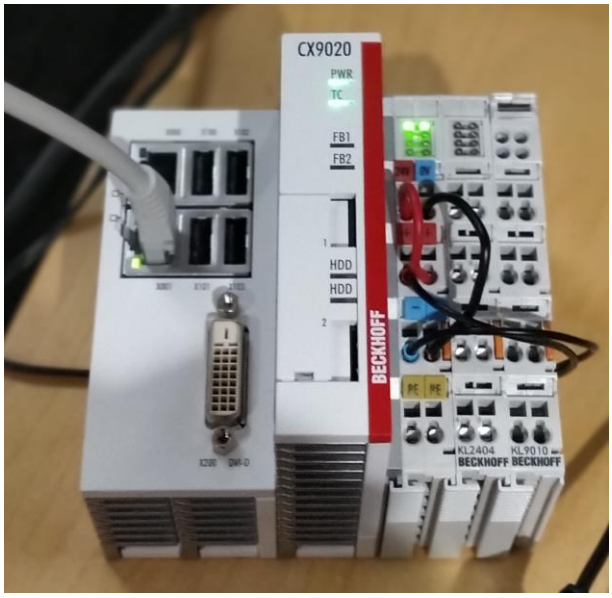
2. Teoretisk bakgrund

För att konstruera och testa kommunikationsbiblioteket användes en PLC från Beckhoff tillsammans med deras utvecklingsmiljö TwinCAT 3. För att ta emot och skicka meddelanden från och till PLC:n användes IoT-hub tjänsten på Microsoft Azure som kräver krypterad kommunikation. För att testa biblioteket i praktiken kompletterades PLC:n med en temperatursensor från Siemens och en HMI panel från Beijer Electronics.

2.1. Programmable logic controller (PLC)

En programmable logic controller är en styrenhet för automation av industriprocesser. Styrenheterna skapades för ersätta logiksystem baserade på relälteknik som användes fram till sena 60-talet och har sen lanseringen vidareutvecklats fram till dagens PLC system.

Grunden i en PLC-enhet är en processorenhet som kan programmeras för önskad tillämpning. Till processorenheten kan moduler av olika sorter anslutas för att möjliggöra kontakt med givare, styrdon, andra styrenheter med mera. I projektet har Beckhoffs processorenhet CX9020 CPU modul används tillsammans med input/output, I/O, moduler, se Figur 2.1 och har styrts med Beckhoffs mjukvarusystem TwinCAT 3. CX9020 har ett ”Microsoft Windows embedded compact 7” operativsystem installerat som möjligör konfigurering lokalt mot PLC:n via monitor och mus.



Figur 2.1:Beckhoff CX9020 CPU modul med två I/O-moduler.

Industriella PLC:er använder sig i stor utsträckning av olika trådbaserade kommunikationprotokoll som kopplas till varandra och till ett operatörsgränssnitt med namnet supervisory control and data acquisition, SCADA, -system. Inom industrin finns flera olika kommunikationsprotokoll vilket kan försvåra integration av operatörsgränssnittet om olika fabrikat används. Efter implementeringen av kommunikationsprotokollet ethernet hos PLC finns möjligheten att kommunicera med externa enheter via internet.

2.2. Motståndstermometer

För noggrann mätning av temperatur används olika typer av motståndstermometrar där resistansen i dess kretsar varierar med omgivningens temperatur. Under projektet används en Siemens PT-1000 temperaturgivare som är byggd för ventilationstrummor. Beteckningen PT-1000 betyder att den temperaturkänsliga delen av sensorn är gjord av platina och har en basresistans på 1000 Ohm.

Sensorn har en strömomvandlare som gör det möjligt att välja om man vill ha signalen från sensorn i form av spänning eller ström. I detta projekt valdes inställningar så att temperaturintervallet -50 till 50 °C representeras av strömintervallet 4 till 20 mA. För att tolka signalen används en analog till digital omvandlare, A/D-omvandlare. A/D-omvandlaren konverterar signalen till ett heltal med tecken på 16 bitar, där 4 mA motsvarar värdet 0 och 20 mA motsvarar $2^{15} - 1 = 32767$. Med signalen beräknades rumstemperaturen ut enligt ekvation (2.1).

$$T = \left(x - \frac{x_{max}}{2}\right) \cdot \frac{100\ (^{\circ}C)}{x_{max}},\ x_{max} = 2^{15} - 1 = 32767 \qquad (2.1)$$

Där T är rumstemperaturen i °C, x är omvandlade insignalen och x_{max} är största möjliga insignal.

2.3. Human Machine Interface (HMI)

För att direkt kunna presentera och mata in information till ett PLC system kan ett så kallat Human Machine Interface (HMI) användas. Moderna HMI:er består oftast av en pekskärm som kopplas till nät av styrenheter. Genom att länka variabler mellan ett HMI och till exempel en PLC kan programvariabler grafiskt presenteras och ändras av en operatör. I projekten användes en HMI panel från Beijer tillsammans med tillhörande utvecklingsmiljö iX Developer.

2.4. TwinCAT

För att programmera sina styrsystem så utvecklade Beckhoff mjukvaran TwinCAT som kan gör det möjligt att programmera kompatibla styrenheter med PLC kod [1]. TwinCAT stödjer ett antal utvecklingsspråk som är förknippade med PLC. Program utvecklas med TwinCAT integrerat i Microsoft Visual Studio och tillåter lokaluppkoppling via Ethernet till PLC.

I den senaste versionen av mjukvaran, TwinCAT 3, har Beckhoff lagt in ett funktionsblocksbibliotek Tc3_IotBase. Biblioteket möjliggör kommunikation enligt protokollet av MQTT och därmed uppkoppling mot molntjänster som Azure.

2.5. Microsoft Azure IoT-hub

Internet of things, IoT, är ett koncept där ett antal elektroniska enheter av olika komplexitet kommunicerar med varandra och utbyter information. En IoT-enhet kan vara enkla enheter från lampor och hushållsapparater till avancerade styrenheter. IoT gör att väldigt många olika typer av enheter kan kommunicera. En IoT enhet ansluts till en hub antingen direkt eller via en gateway för att skapa ett sammankopplat system av kommunicerande enheter. [2].

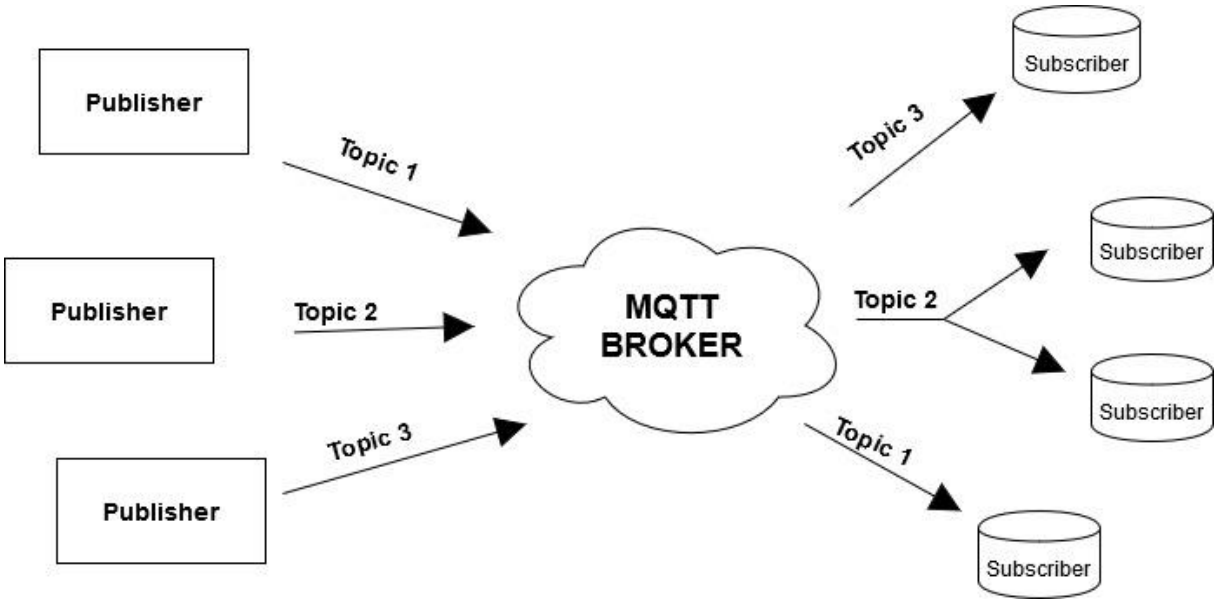
Azures IoT-Hub är en hanteringstjänst och används som en kommunikationscentral för tvåvägskommunikation mellan IoT-applikationer och enheter. Microsofts IoT-hub erbjuder säker och stabil anslutning genom krav på säkerhetsprotokoll som etableras genom token-baserad autentisering [3]. Microsoft har implementerat MQTT som ett meddelandeprotokoll för kommunikation med IoT-enheter vilket gör det möjligt för dem att kommunicera med IoT-hubben [4].

Information som levereras från de olika enheterna fördelas regelbaserat vidare via så kallade ”endpoints” som sedan Azure-applikationer är kopplade till. Ett exempel är en applikation är Acobias REST-API.

2.6. Message Queueing Telemetry Transport (MQTT)

MQTT är ett maskin till maskin, M2M, kommunikationsprotokoll och är utvecklad för maskiner med begränsad prestanda till exempel mikroprocessorer eller smarta givare. Protokollet bygger på publicerings/prenumeration där enheter som är ”Subscribers” läser från en adress som benämns som ”Topic”. ”Publishers” är de enheter som kan publicera till Topics. Genom olika Topics och Topic-hierarkier skapas kommunikationsstrukturer [2].

Alla meddelanden mellan enheterna hanteras och fördelas genom en MQTT ”Broker” som tar emot meddelanden från Publishers till ett Topic och skickar vidare dem till alla Subscribers som prenumererar på det ämnet, exempel i Figur 2.2 nedanför.



Figur 2.2: Exempel på meddelandehantering med MQTT.

2.7. Mosquitto

Eclipse Mosquitto är en open source mjukvara som kan användas för MQTT kommunikation. Programmet kräver inte mycket processorkraft och kan implementeras på allt från enklare enheter till stora servrar. Mosquitto körs via kommandotolken och kan användas som broker, subscriber och publisher. Programmet möjliggör både lokal och internetbaserad kommunikation.

2.8. Transport Layer Security (TLS)

TLS är ett säkerhetsprotokoll vilket skapar en krypterad anslutning mellan en klient och en server. För att förhindra att informationen avläses eller förändras mellan server och klient så behövs en handskakning vid anslutning.

Handskaket är uppbyggt genom att klienten skickar ett meddelande att den vill ansluta till servern. Servern svarar med att skicka ett certifikat till klienten, vilket klienten använder för att generera en krypterad nyckel. Klienten skickar sedan nyckeln och identifieringsinformation som då ska matcha med vad servern har genererat för samma användare. Om nycklar matchar har handskakningen lyckats och klienten får ansluta. Vid fortsatt kommunikation från samma klient kan den gamla nyckeln användas och full handskakning krävs ej [5].

Ett annat sätt att ansluta klient med server är att manuellt ange den krypterade nyckeln lokalt hos klienten. Med mjukvaror som Azure IoT Explorer kan en nyckel från servern, kallad shared access signature (SAS-token), genereras för den specifika klienten. Då kan klienten direkt skicka nyckeln till servern för handskakning.

2.9. Azure IoT explorer

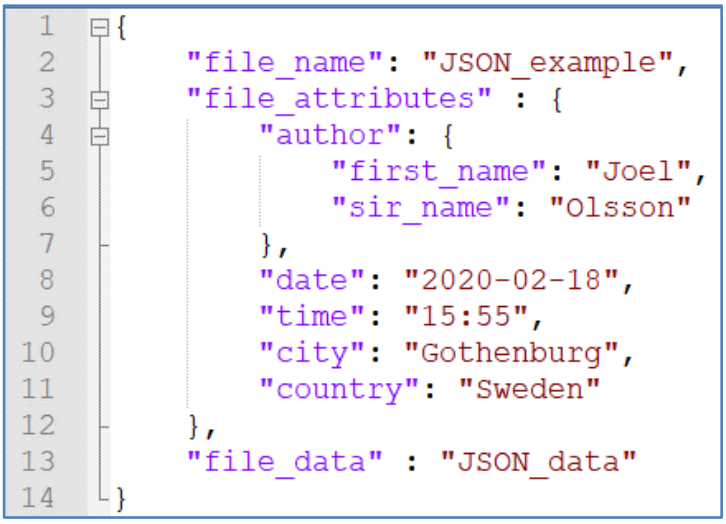
För generering av SAS-tokens kan ett antal olika program användas som Microsoft utvecklat. Det senaste som upprätthåller den aktuella standarden är Azure IoT Explorer som erbjuder uppkoppling till IoT-hubbar via en uppkopplingssträng som genereras på hubben. Efter uppkoppling kan en uppkopplings-”device” skapas som enheterna kan koppla upp sig mot. För varje device finns ett antal verktyg tillgängliga så som SAS-generering, telemetriövervakning och cloud to device meddelanden [6].

Azure IoT Explorer finns även som en insticksmodul för Visual Studio Code (VSC) som möjliggör konfigurering och övervakning av IoT-hubbens devices samtidigt som man utvecklar applikation för Azure.

2.10. JavaScript Object Notation (JSON)

JSON är en formatstandard för att lagra dataobjekt i text och används mycket för datautbyte. JSON formatet är lättläst för människan och enkelt för maskiner att läsa och generera. Objekten byggs upp av par med taggar och data som kan observeras i exemplet i Figur 2.3. Maskiner kan med hjälp av sträng-hantering dela upp strängen och para ihop taggarna med dess data. I exemplet förevisas hur man på olika sätt kan bygga upp datastrukturer som kan skickas som en textsträng.

Första taggen ”file_name” är kopplad till texten ”JSON_example” och kan användas för att representera en variabel i maskinen. Tagg nummer två ”file_attributes” är kopplad till ett antal underliggande taggar som är parade med information. Detta kan jämföras med en datastruktur eller ”Struct” som är vanlig i högnivåspråk.



Figur 2.3: JSON exempel.

2.11. Domain Name System (DNS)

För att kommunicera över nätverk används IP-adresser för att identifiera uppkopplade enheter. För att förenkla anslutningen för användaren används ett system för att översätta IP-adresser till domännamn. Till exempel för att ansluta mot IP-adressen 216.58.211.142 kan domännamnet www.google.com användas. För att översättningen ska fungera används en DNS-databas. Databasen är globalt distribuerad och innehåller information om vilken IP-adress som är kopplat till ett domännamn [7].

3. Metod

I metodkapitlet beskrivs de metoder som användes under arbetet. Projektet kunde delas upp i tre delar: etablera anslutning mellan PLC och Azure, konstruera kommunikationsbibliotek och slutligen implementering av resultatet som en konceptvalidering.

3.1. Anslutning

För att etablera en stabil anslutningsmetod testades anslutningen hos PLC och Azure med hjälp av en MQTT klient. Anslutning till en MQTT broker underlättade framtagning av de nätverksinställningar som krävdes för kommunikation med internet.

För att underlätta felsökning delades anslutningstesterna upp i steg enligt följande:

1. Anslutning till broker på samma dator från en MQTT klient.
2. Anslutningen till broker som ligger på en annan dator i det lokala nätverket med en MQTT klient.
3. Anslutning till broker från PLC.
4. Anslutning till broker på en molnserver från MQTT klient.
5. Anslutning till broker på molnserver från PLC
6. Anslutning till Azure från MQTT klient med TLS.
7. Anslutning till Azure från PLC med TLS.

Under varje steg testades publicering och mottagning av meddelanden innan nästa steg påbörjades. När anslutning mellan PLC och Azure etablerats strukturerades meddelanden i JSON-format enligt Acobias standard för att REST-API ska kunna tolka informationen. När skarp kommunikation etablerades kunde konstruktionen av biblioteket påbörjas.

3.2. Konstruktion

Kommunikationsbiblioteket byggdes upp av tre funktionsblockskategorier efter vad MQTT protokollet kräver för att fungera. Första kategorin som behövs är en uppkopplingstyp där anslutning mot Azure hanteras. Sedan för att möjliggöra skrivning till molnet behövs en publiceringstyp samt en prenumerationstyp för att ta emot information.

Biblioteket strukturerades efter tillverkarnas standard samt Acobias önskemål om att anpassa efter deras API. Funktionsblocken skrevs med strukturerad text och anpassades så implementering ska kunna göras i de vanligaste PLC-programspråken. Eftersom det utvecklade API kräver JSON format med vissa specifika värden skapades JSON metoder som kan läsa och skriva enligt dessa.

Under utvecklingen av biblioteket användes Team foundation server, TFS, backup i Visual studio. TFS användes för att säkra backups mellan olika iterationer av projektet.

3.3. Konceptvalidering

Konceptvalidering eller proof of concept, POC, förevisade kommunikationsbibliotekets olika funktioner och gav en enkel bild av den potential som finns i bibliotekets olika funktioner. Valideringen gjordes genom ett programexempel med hårdvara implementerad utöver PLC:n för att på ett enkelt sätt dra paralleller till dagens industrisystem. Konceptet inkluderade bibliotekets tre huvudfunktioner tillsammans med hårdvara som integrerades i Acobias IoT-infrastruktur för att verifiera att målen med projektet uppfyllts.

4. Genomförande

Detta kapitel beskriver processen för att uppnå projektmålen. Som beskrivet i kapitel 3 ovan delades projektet upp i tre huvudsteg som krävde att de tidigare stegen behövde vara avklarat innan nästa steg kunde påbörjas.

4.1. Anslutning

Första steget i projektet var att med MQTT-protokollet ansluta till en broker samt att sätta upp klienter för prenumeration och publicering. Detta testades lokalt på en dator med MQTT-mjukvaran Mosquitto som kunde användas både som broker och klient. För att skicka och ta emot meddelanden från den lokala brokern krävdes endast datorns lokala IP-adress samt vilken port som brokern ställdes in på. Inga lösenord eller kryptering användes här för att underlätta anslutningen.

Mosquitto körs lokalt via Windows command prompt, CMD där operativsystemets enklaste former av operationer inmatas. Ett enkelt exempel på att publicera till den lokala brokern skrivs enligt följande i CMD:

mosquitto_pub -h “192.168.0.20” -p 1883 -t “topic” -m “message”

De tre återkommande parametrarna som krävs vid anslutning var domännamnet (-h) som i detta fall var en IP-adress. Kommunikationsporten till datorn som brokern var kopplad till (-p) och ämnet eller ”topic” (-t) där meddelanden skickades under eller enheter prenumererade till. För att prenumerera på ett ämne användes ett annat kommando men med samma tre anslutningsparametrar enligt följande:

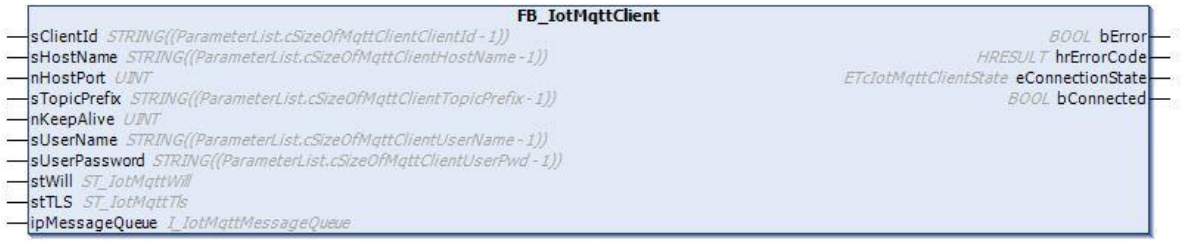
mosquitto_sub -h “192.168.0.20” -p 1883 -t “topic”

För anslutning mellan två datorer på samma lokala nätverk användes samma uppsättning av kommandon där IP-adressen är till den dator som brokern var aktiv på. Här behövdes portar i brandväggen på datorerna öppnas så meddelanden kunde passera.

För anslutning och konfiguration av PLC:n till brokern via det lokala nätverket användes funktionsblocket FB_IotMqttClient från Beckhoffs bibliotek Tc3_IotBase. Funktionsblockets samtliga in- och ut parametrar visas grafiskt i Figur 4.1. Samma anslutningsparametrar som användes tidigare matades in i funktionsblocket via inputparametrarna enligt följande:

sHostName = ” 192.168.0.20”
nHostPort = 1883

För att kontakten skulle bibehållas samt att publicering/prenumeration skulle fungera behövdes exekveringsmetoden ”Execute” cykliskt aktiveras så händelser mellan brokern och PLC:n kan hanteras. Flaggan bConnected från funktionsblocket meddelade om kontakt med brokern hade etablerats.



Figur 4.1: Grafisk representation av funktionsblocket Fb_IotMqttClient från TwinCAT:s utvecklingsmiljö.

Publicering av meddelanden gjordes med metoden ”Publish” som likt ”Execute” är en submetod av FB_IotMqttClient och använder sig av etablerad anslutning för att skicka information i så kallade ”payloads” till specificerat topic. Utöver topic och meddelande så kräver metoden ett antal konfigureringsparametrar [8, p. 26] för att specificera hur meddelandet ska hanteras av brokern.

För att ta emot meddelanden från brokern så användes submetoden ”Subscribe”. Funktionsblocket etablerade då en prenumeration till önskat topic. Metoden användes vid initiering av programmet och returnerar en flagga för att meddela om den lyckats. För att hantera meddelanden från brokern använder sig funktionsblocket av en först in först ut kö, FIFO-kö, där meddelandena placeras i ankommandeordning.

Efter att samtliga funktioner hade testats på lokalt nätverk gjordes övergången till uppkoppling över internet. På ett globalt nätverk är kravet på säker anslutning viktigt och flera säkerhetsfunktioner som lösenordsskydd och TLS behövde implementeras.

Innan uppkoppling till Azure provades alla funktionerna på den Mosquitto-baserade webbservern cloudMQTT.com. På webbservern kunde en MQTT broker sättas upp för enkel hantering av MQTT kommunikation. Webb-brokern gjorde det möjligt att få direkt feedback på valda parametrar som behövdes för kommunikation över internet.

Kontakt mellan webb-brokern och Mosquitto-klienten placerad på en PC krävde att inloggning med hjälp av användarnamn och lösenord som bestäms på webbservern. En enkel publicering till webb-brokern från en Mosquitto-client via CMD kunde se ut enligt följande:

> **mosquitto_pub -h “xx.cloudmqtt.com” -p 8883 -u “username” -P “password” -t “topic” -m “message”**

De stora skillnaderna mellan interna och externa uppkopplingarna är adressen till brokern. Istället för en IP-adress så använder man domännamnet till serverenheten. För kontakt via domännamn krävs det att klienten är kopplad till en DNS-server vilket översätter domännamnet till en IP-adress.

Uppkoppling med PLC:n till webb-brokern byggde på samma princip som Mosquitto-clienten använde sig av. Den viktigaste skillnaden är att PLC:n inte alltid har en DNS-server kopplad vilket kan leda till problem om man ska ansluta via domännamn. En lösning på problemet som användes i projektet är att manuellt ansluta PLC:n till en DNS-server via nätverksinställningar på dess inbyggda operativsystem.

När PLC:n är inställd för DNS-adresser kan samma parametrar som användes för uppkoppling från PC brukas. Samtliga parametrar som matades in i funktionsblocket blev då:

sHostName = ”xx.cloudmqtt.com”
nHostPort = 8883,
sUserName = ”userName”
sUserPassword = ”password”

Komplett kod för anslutning till webb-brokern finns i bilaga A.

Fram hit så hade all kommunikation varit okrypterad men för att kunna ansluta med Microsoft Azure krävdes säkerhetsprotokollet TLS. Både Mosquitto och funktionsblocket för TwinCAT har TLS integrerat vilket gör det möjligt att konfigurera mjukvaran för detta protokoll.

För att få etablerad kommunikation med Azure via MQTT-protokollet krävdes det specifika värden på inloggningsparametrarna vars struktur kunde hittas på Microsofts online dokumentation för Azure [4]. Till skillnad från en typisk MQTT-broker så är Azures IoT-hubb en samling av ”devices” som istället för att distribuera meddelanden mellan klienter tar emot dem och skickar vidare meddelandena till så kallade ”endpoints” där andra Azure-strukturer hanterar informationen. På samma sätt skickas meddelanden från Azure via IoT-hubben till klienterna. För att koppla upp sig till en device krävs domännamnet till IoT-hubben (ExjobbAcobiaGBG.azure-devices.net), namnet på enheten device-id (device_admin) samt ett SAS-token genererad för den aktuella devicen.

För att publicera ett meddelande till IoT-hubben via enheten ”device_admin” med hjälp av Mosquitto krävdes ett antal nya parametrar som i ett CMD-kommando såg ut enligt följande:

mosquitto_pub
-h " ExjobbAcobiaGBG.azure-devices.net"
-i "device_admin"
-u " ExjobbAcobiaGBG.azure-devices.net/device_admin/?api-version=2018-06-30"
-P "SharedAccessSignature sr=ExjobbAcobiaGBG.azure-devices.net%2Fdevices%2Fdevice_admin&sig=8gMzf%2BNSxppvj6pYeP6q0Fq6sQiwpKmVDyqN85pGvEE%3D&se=1588323133"
-t "devices/device_admin/messages/events/"
--cafile C:\TwinCAT\3.1\Config\Certificates\BaltimoreCyberTrust.crt
-p 8883
-V mqttv311
-m "hello azure fom local Mosquitto client"

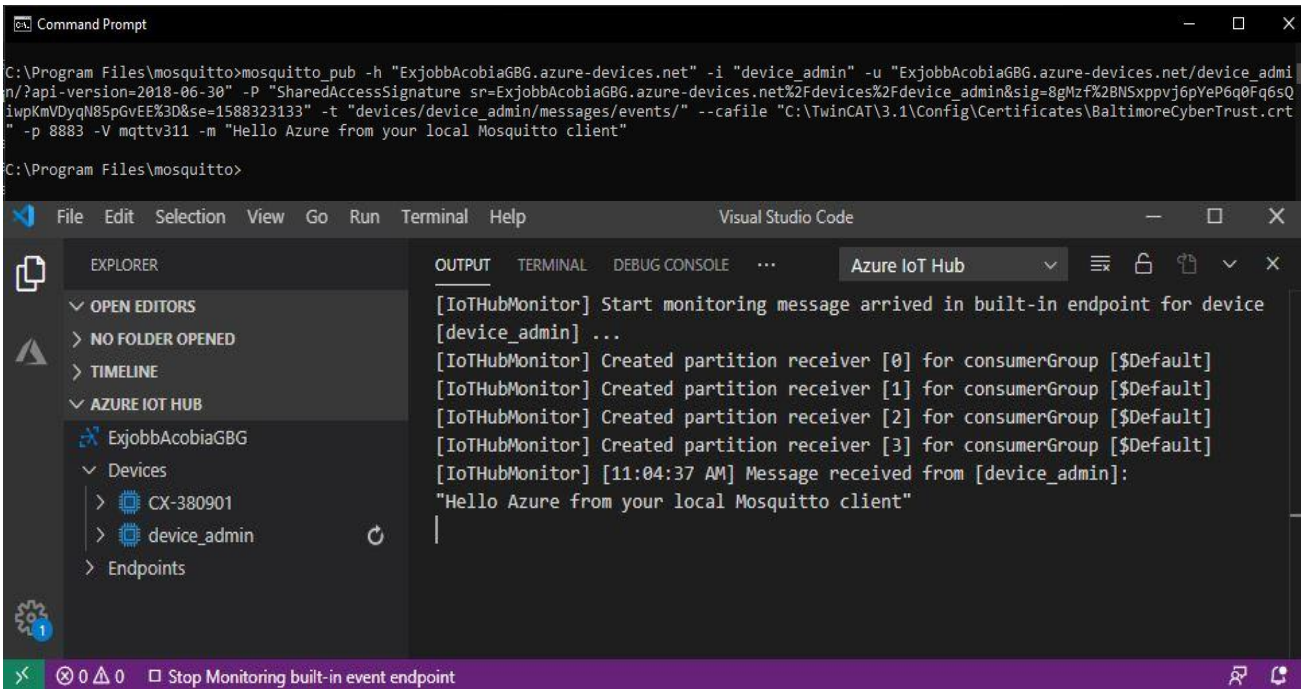
Den device som skulle anslutas till angavs via -i parametern, användarnamnet på -u var en kombination av domännamn, device-id samt api-version som används. Lösenordet bestod av SAS-token på formen:

> **”SharedAccessSignature sig={signature-string}&se={expiry}&sr={URL-encoded-resourceURI}”**

Meddelanden som skulle skickas till devicen skulle gå under ett specifikt topic: **”devices/{device_id}/messages/events/”**. För TLS-protokollet krävdes att sökvägen till ett anslutningscertifikat på datorn, som specificeras med kommandot –cafile. Slutligen behövdes

anslutningsport samt vilken version av MQTT-protokollet som användes, Azure använder för tillfället 3.11. Ska man prenumerera på samma device krävs samtliga parametrar som tidigare förutom topic, som istället använder en sökväg enligt följande **”devices/{device_id}/messages/devicebound/#”**.

Eftersom inga övriga Azure resurser var kopplade till den testdomän som användes under anslutningsfasen användes Visual Studio Code för att läsa meddelanden till IoT-hubben. Ett exempel på ett meddelande som visas i VSC ges i Figur 4.2 nedanför.



Figur 4.2: Resultat D2C- meddelande från Mosquitto klient som visas i VSC.

Efter kontakt hade etablerats och kommunikation åt båda hållen hade lyckats skulle dessa inställningar tillämpas i PLC:n. Istället för att överföra alla parametrar på samma sätt som i tidigare steg så användes parameterstrukturen ST_IotMqttTLS i anslutningsblocket där det fanns en specifik parameter för anslutning via SAS-token.

Eftersom all information som krävdes för att ansluta till IoT-hubben redan fanns i SAS-token så har Beckhoff skapat en funktion där endast ett SAS-token och sökvägen till TLS-certifikatet krävs för att ansluta. I detta fall är det viktigt att notera att den sökväg till certifikatet är i PLC:s minne. Efter lyckad anslutning användes samma publicerings och prenumerings metoder som tidigare för kommunikation. För att verifiera kommunikationen en extra gång användes även här VSC. PLC-kod på hur användning av Beckhoffs färdiga anslutningsfunktion finns i bilaga B.

4.2. Konstruktion av funktionsblocksbibliotek

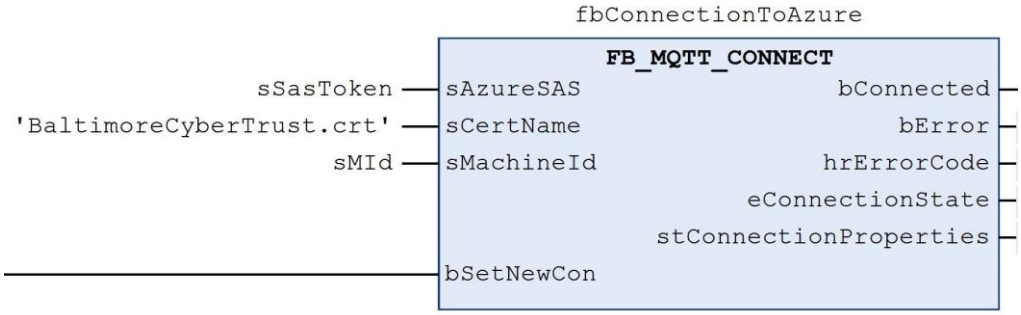
Efter att en fungerande anslutningsmetod var skapad kunde konstruktionen av funktionsblocksbiblioteket påbörjas. Eftersom målet med biblioteket är att underlätta implementering av kommunikation mellan PLC och Azure så delades de olika momenten av kommunikationen upp i tre funktionsblockstyper beskrivna nedan.

4.2.1. Anslutningsblock

Första funktionsblockstypen byggde på den metod som tagits fram under anslutningsdelen av projektet. Anslutningsblocket eller ”Connection”-blocket har uppgiften att med endast ett SAS-token och namnet på ett TLS-certifikat skapa en stabil anslutning till en Azure IoT-hubb.

För att minimera den input som behövs för anslutning har en standardsökväg implementerats i blocket så endast namnet på certifikatet i form av en textsträng behövs. Utöver anslutningsvariablerna ska även ett maskin-id ges för att all information som skickas via blocket ska ha samma id-tagg. För att blocket ska kunna initieras och ändra anslutning kopierar blocket all information från ingångarna när en positiv flank ges på en startingång.

Funktionsblocket har två flaggor som signalerar anslutning respektive fel. Vid fel så presenteras felmeddelanden och felkoder på två separata utgångar. För vidare användning av biblioteket skapas en anslutningsstruct av anslutningsblocket. En struct är en datastrucktur av ett bestämt antal variabler som länkas samman. Anslutningsstructen används för att förenkla sammankopplingen mellan funktionsblocken. Structen innehåller information om vilket anslutningsblock och enhet som ska användas vid skrivning och läsning till den specifika IoT-huben. Den grafiska layouten av ”Connection”-blocket kan ses i Figur 4.3 som är tagen ur det testprogram som användes vid kontroll av blockets funktioner. Det kompletta testprogrammet finns i bilaga C.

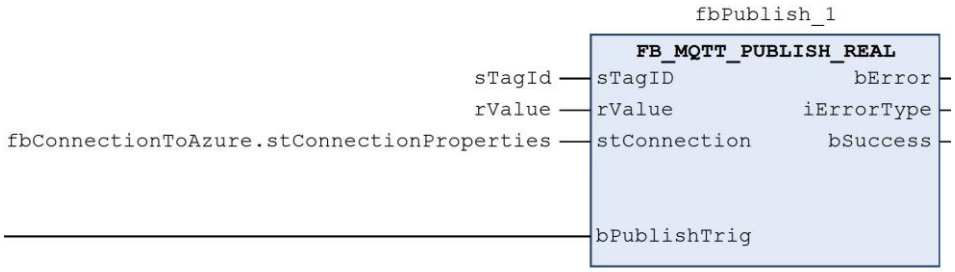


Figur 4.3: Anslutningsblocket taget från testprogram, se bilaga C.

4.2.2. Publiceringsblock

Eftersom Acobias API krävde en specifik JSON-struktur för att tillåta skrivning till taggar utvecklades funktionsblockstyp för publicering av meddelanden. Funktionsblockets huvuduppgift blev att ta in ett datapar som representerar ett tagg-namn och ett tagg-värde. Dessa variabler omvandlas sedan till ett JSON meddelande som accepteras av API. För att kunna publicera meddelandet kräver blocket en koppling till anslutningsblocket som etableras genom den anslutningsstruct som ges från blocket. Blocket har en triggervariabel som på positiv flank konverterar och skickar dataparet.

På önskemål från Acobia hade ett publiceringsblock skapats för varje datatyp, till exempel REAL, INT, BOOL med flera. Ett grafiskt exempel av ett publiceringsblock av typen REAL kan ses i Figur 4.4. Konstruktionen är gjord så att flera publiceringsblock kan anslutas till samma anslutningsblock.



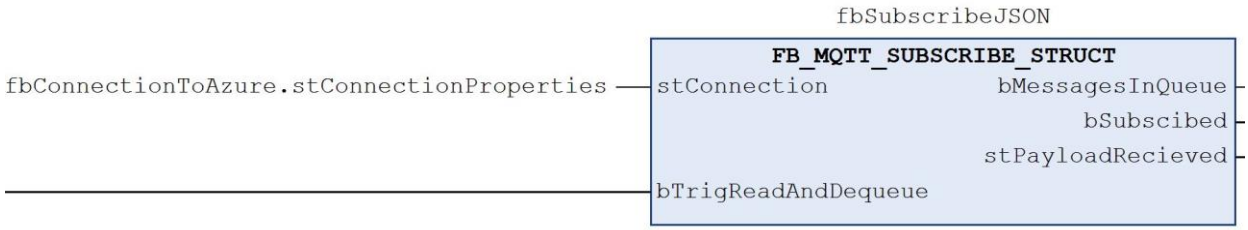
Figur 4.4: Publiceringsblock för datatypen REAL från testprogram, se bilaga C.

4.2.3. Prenumerationsblock

För att underlätta hantering av inkommande meddelanden från IoT-hubben konstruerades ett funktionsblock för att hantera meddelandena. ”Subscribe”-blocket har som uppgift att skapa en prenumeration till den anslutningen som den kopplas till. Eftersom blocket kopplas till den FIFO-kö som existerar i FB_IotMqttClient inuti anslutningsblocket så kan endast ett ”Subscribe”-block kopplas per anslutningsblock.

För att information ur MQTT-kommunikationen skulle kunna hanteras behövde JSON-meddelanden tolkas. För att ta ut information ur ett JSON objekt användes ett funktionsblock FB_JsonDomParser ur Beckhoffs bibliotek TC3_JsonXml. Blocket använder sig av metoden FindMember för att hitta positionen på önskad taggs värden. Sedan används undermetoden GetString för att plocka ur värdet från taggen. En variabelstruktur och en avläsningsfunktion skapades för att hantera de specifika kommunikationstaggarna som används i det befintliga API. Avläsningsfunktionen undersöker om taggar finns, läser ur värdet och skriver in det i önskad struktur för vidare användning i prenumerationsblocken.

När prenumerationsblocket initieras görs försök för att skapa en aktiv prenumeration till IoT-hubben. Subscribe-blocket kopplas till anslutningsblocket via den tidigare nämnda anslutningsstructen. När prenumeration lyckas kommer detta indikeras med en flagga. En andra flagga används för att indikera att det ligger meddelanden FIFO-kön. När ett meddelande finns i kön kan detta hämtas genom en positiv flank på blockets andra ingång, då placeras det nya meddelandet på ”payload”-utgången. För att kunna anpassa informationen som tas emot finns det tre olika block av ”Subscribe”-typen. Blocken hanterar de mottagna meddelanden i olika former, i form av en variabelstruktur (STRUCT), separata variabler för varje tagg i JSON meddelandet (SPLIT) eller bara hela JSON meddelandet som en textsträng (JSON). För att konvertera inkommande JSON i SPLIT och STRUCT används den skapade avläsnings-funktionen. I Figur 4.5 kan ett exempel av prenumerationsblocket av typen STRUCT.



Figur 4.5: Subscribeblocket taget från testprogram, se bilaga C.

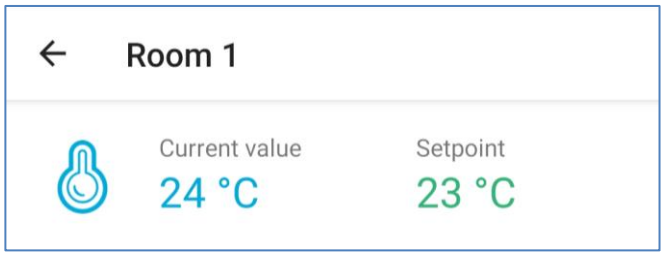
4.3. Konceptvalidering (POC)

För att demonstrera funktion och påvisa att målen med projektet är uppfyllda skulle en tillämpning av funktionsblocksbiblioteket konstrueras. Till detta användes den PLC som använts under utveckling tillsammans med en input-modul, en temperatursensor från Siemens, en HMI panel från Beijer samt Acobias egenutvecklade mobilapplikation.

Grunden av PLC programmet bestod av ett anslutningsblock, ett prenumerationsblock och ett publiceringsblock, samtliga från kommunikationsbiblioteket. En variabel som representerade givarens värde länkades till en av de fysiska ingångarna i input-modulen.

Siemens temperaturgivaren kopplades till en analog ingång i PLC:n och konfigurerades för intervallet -50 till 50 °C vilket motsvarades av signalen 4–20 mA. För att tolka signalen i PLC:n används en inbyggd A/D-omvandlare.

Vid initieringen av programmet sattes uppkopplingsvariablerna till anslutningsblocket och skapade anslutningen till IoT-hubben. Hantering av kommunikationen gjordes med hjälp av de övriga två blocken som kopplades till anslutningsblocket. Beräkningen av temperaturen gjordes med metoden från ekvation (2.1). När temperaturen avviker från senaste skickade värde med 0.5 grader samt att en uppdatering inte gjorts på 5 sekunder, skickas ett nytt värde via publiceringsblocket. Meddelandet går till Acobias applikation via Azure som sedan presentera under taggen ”Current value”, se Figur 4.6.

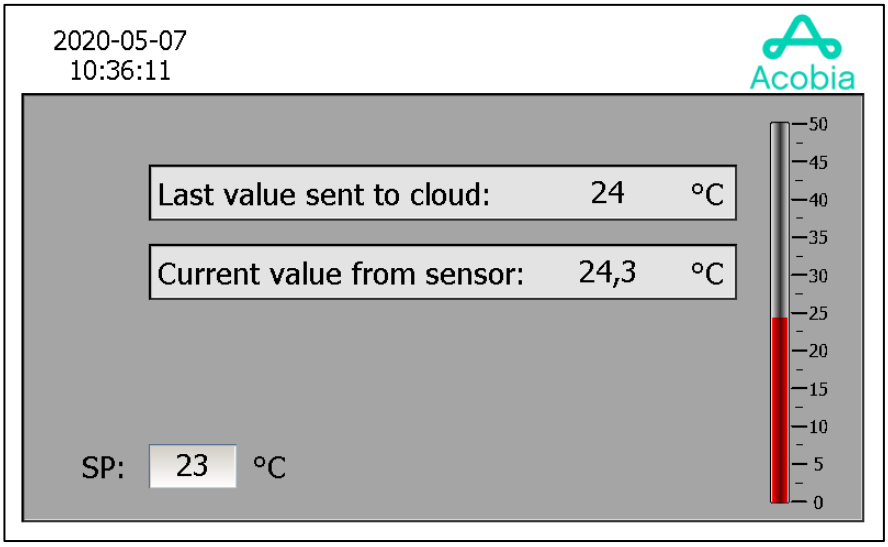


Figur 4.6: Skärmbild från Acobias smartphoneapplikation.

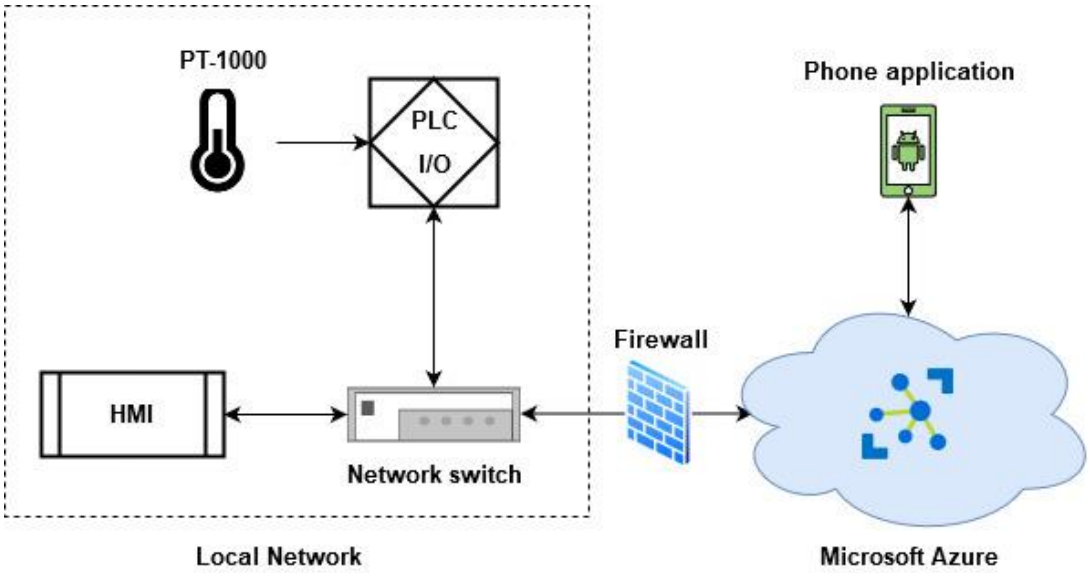
Applikationen kan ändra värde på taggen ”Setpoint” som då skickar ett meddelande genom Azure till PLC:n. Meddelandet från Azure tas emot och hanteras av subscribe-blocket som är av typen SPLIT. Värdet och taggnamnet från meddelandet skickas till publiceringsblocket som svarar applikationen för att konfirmera att meddelandet mottagits. Nya setpoint-värdet sparas även i PLC:n för att presenteras på HMI.

Utöver applikationen visas alla värden på ett HMI som kopplades till PLC:n via det lokala nätverket. På HMI:t skapades en bild som presenterar flera parametrar, se Figur 4.7 för HMI-

layout. På HMI-bilden visas aktuellt sensorvärde, senast uppdaterad värde till hubben, en grafisk representation av aktuell temperatur samt aktuell setpoint. Rutan för setpoint tillåter manuell inmatning via pekskärmen. Om setpoint värdet är skilt från aktuellt värde skickas en uppdatering till applikationen via publishblocket. En helhetsbild av systemet presenteras grafiskt i Figur 4.8, koden kan observeras i bilaga D.



Figur 4.7: HMI layout för konceptvalidering.



Figur 4.8: Grafisk representation av POC

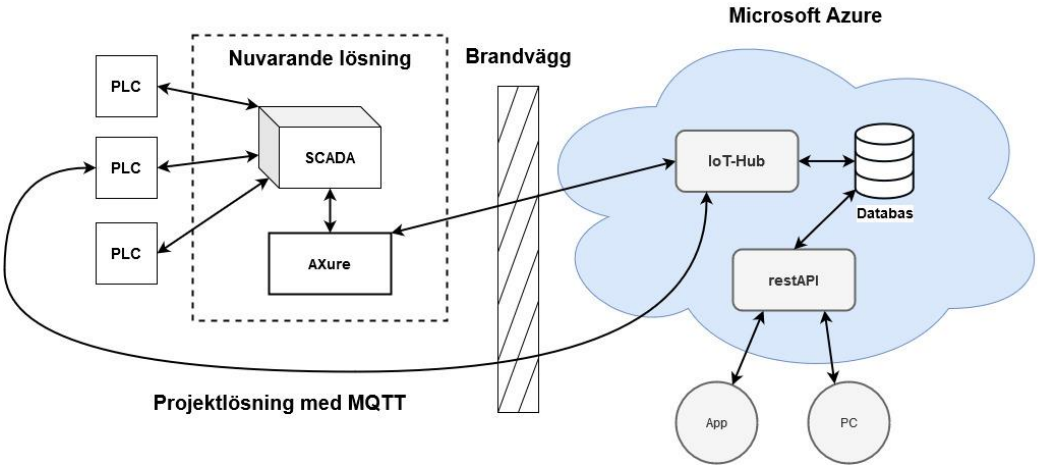
5. Resultat

Syftet med projektet var att skapa möjligheten till en enkel och funktionell lösning mellan PLC, IoT och Azure-applikationer. Projektet har testat fram konfigurationer för att ansluta Beckhoffs PLC mot molntjänsten, skapat ett funktionsblocksbibliotek för MQTT-kommunikation samt skapat en testrigg för validering av hela konceptet.

Kommunikationsbiblioteket byggdes upp av tre olika typer av block. Ett anslutningsblock vilket enkelt ansluter mot önskad IoT-hub med en genererad SAS-nyckel, och ger felkoder för olika anslutningsfel. Anslutningsblocket vidarebefordrar även önskad information till de övriga funktionsblocken. Andra typen av block är publishblock där det skapades ett block för varje datatyp, så att publicering av alla datatyper skulle bli möjlig. Tredje typen som utvecklades var tre olika subscribeblock, vilket möjliggör en enklare hantering av den lästa informationen.

Testriggen för validering av projektet baserades på en mobilapplikation, en temperaturgivare, en PLC, en HMI samt en IoT-hub. Med testriggen kunde varje del av kommunikationen och biblioteket testas, ändringar i temperatur vid PLC uppdaterades automatiskt i applikationen samt HMI: t. Uppdateringar av önskad temperatur från applikationen besvarades korrekt av PLC och presenterades i HMI.

Resultatet av projektet uppfyllde samtliga delmål som framställts från Acobias ursprungliga krav. Verktygen i kommunikationsbiblioteket gör det möjligt att skapa en anslutning och hantera tvåvägskommunikation med Acobias Azure-applikationer. Konceptvalideringen visade även att kommunikationen endast krävde tre funktionsblock för att fungera, vilket påvisar att implementering kan göras utan större insats. Genom konstruktionen av tillämpningsexemplet kunde kommunikationsbiblioteket presenteras och användas som ett exempel för den potential som IoT baserad kommunikation har för industri- och fastighetsautomation. I Figur 5.1 kan den tidigare lösningen jämföras med projektlösningen, vilket kan påvisa möjligheten till billigare och enklare lösningar.



Figur 5.1: Jämförelse mellan nuvarande lösning och projektlösning.

Både Acobia samt vi som projektgrupp är nöjda med åstadkommit resultat och ser möjligheter att utveckla fler applikationsområden för arbetet. Den säkerhetsgrad som etablerades i biblioteket ansågs tillräcklig för Acobia.

6. Slutsats & diskussion

I detta kapitel diskuterar vi projektet, dess resultat, säkerhet samt MQTT-konceptets framtida potential från en hållbar aspekt.

6.1 Tillämpningar av resultat

Projektet lyckades med att skapa en verklig lösning av det koncept som Acobia hade presenterat under projektets uppstart. Eftersom MQTT-protokollet skapades för enkel kommunikation behövdes det ingen specifik datastruktur hos den meddelandet som skickas. Eftersom JSON användes som meddelandestruktur kan det nuvarande programmet utvidgas med nya datapar för tillämpning i andra applikationer.

En tillämpning kan enkelt utvecklas direkt ur testriggen. Konceptvalideringen presenterar endast nuvarande temperatur och en setpoint i en kontorsyta men applikationen är konstruerad för att simulera en hel kontorsbyggnad. Med en regulator i PLC:n skulle det skapa möjligheten att mobilt styra temperatur och klimat i flera ytor. Flera kontorsytor skulle kunna läggas till i webbapplikationen och kopplas till flera givare, vilket enkelt kan implementeras med olika taggar i projektlösningen. Den stora nackdelen med uppkopplingen är uppdateringstider då all kommunikation går via internet. Kommunikation över internet kräver mer information i meddelanden vilket gör att fördröjningen blir betydligt större än en lokal uppkoppling. Projektlösningen har en snittuppdateringstid på två sekunder vilket kan göra systemet för långsamt i tidkänsliga system. Klimatstyrning i kontorslokaler passar projektlösningen då snabbheten i systemet inte är lika viktig. Projektlösningen har möjlighet att enkelt koppla ihop många givare, vilket gör att den anpassas bättre till stora men långsamma system.

6.2 Säkerhet och utvecklingsområden

Ett tydligt utvecklingsområde är i hur SAS-token i systemen hanteras. I projektlösningen har token manuellt skrivits in i en variabel vilket är osmidigt då SAS-token är tidsbegränsade. Ett protokoll på hur tokenet kan uppdateras utan att koppla ner hela systemet och öppna koden behövs. Med vår lösning har tokenet genererats för att hålla en väldigt lång tid, om någon skulle få tag i denna skulle information kunna komma i fel händer. En lösning är att uppdatera denna oftare och på så sätt endast ha korta giltighetstider. För att kringgå problemet med att redigera koden varje gång borde ett program utvecklas som skriver in nyckeln automatiskt. Ett annat problem om SAS-token ska uppdateras automatiskt kan vara övervakning. Om uppkopplingen är övervakad kan nyckeln ändå hamna i fel händer, en lösning på detta skulle vara SAS servrar. Genom att fördela delar av SAS-token på flera serverar och sedan plocka krypterad information från dessa i olika tillfällen, blir det svårare att ta upp informationen vid övervakning [9]. För att säkerställa säkerheten bör nästa steg vid utveckling av vårt system vara att implementera någon form av metod för nyckeluppdatering likt det som nämnts ovan, vilken nuvarande lösning har en stor sårbarhet utan [10].

I projektlösningen är anslutningen mellan endpoints och moln säkrat med TLS vilket skapar en säkrad tunnel för informationen. Nästa steg för att säkra själva informationen är att kryptera meddelandena. En möjlighet är att använda JSON Web Token, JWT, vilket är en teknik skapad specifikt för att kryptera JSON meddelanden. JWT krypterar meddelandet efter ett certifikat likt en översättningsnyckel. Genom att ge varje enhet ett certifikat som är

signerat specifikt för den enheten, skickas ett oläsbart meddelande över nätverket som sedan översätts av enheterna. Det går även likt SAS-servar att ha en säkerhetsserver som enheterna kontrollerar några kontrollbitar i det krypterade meddelandet mot för att se legitimitet innan dekryptering börjar [11]. Nackdelen med denna lösning är att den kräver att varje enhet har processorkraft nog att översätta meddelandena, samt att dekryptering och mellanhänder gör systemet långsammare. Anledningen till att JWT inte användes under projektet är att vi anser att det tappar huvudfördelen med IoT lösningen, att enkelt kunna koppla upp alla enheter oavsett processorkraft. Arbetet avgränsades även av att använda befintliga webapplikationer och API som inte är utvecklade för JWT.

6.3 Hållbarhet

Ett problem med utvecklingen av IoT-baserade enheter är att med ökad kommunikation kommer ökad energianvändning. Energiförbrukningen ökar avsevärt när enheter som tidigare endast styrts manuellt anpassas för att kopplas upp mot en IoT-hub. Ett exempel är intelligenta lampor där den enda energianvändningen tidigare var lampans förbrukning, som efter anpassning behöver en processor och kommunicera med en nätverksenhet för styrning. Fördelen kan vara att man på ett bättre sätt kan styra när lampan används, vilket kan ge en minskning i förbrukning av energi. Trots att lampans energiförbrukning kan minskas med styrningstider så har energianvändningen ökat i både produktionen och vid användning [12].

Vid användning av IoT-lösningen i en anläggning där styrning redan används, till exempel flera fastigheter, kan systemet istället minska energiåtgång och kostnader. Istället för att använda en styrenhet i varje fastighet, så kan intelligenta givare installeras i varje yta och kopplas ihop mot en central styrenhet via befintligt nätverk. På så sätt kan antalet styrenheter minskas vilket leder till lägre inköpskostnader. Administrationskostnader minskar även av att alla system kan styras och underhållas utan att vara på plats vid enheterna.

Ett annat problem är hur kommunikationen hanteras. All information inom IoT går via internet och riskerna för intrång blir större än en lokal lösning. Eftersom antalet internetanslutna enheter ökar, ökar också antalet angreppspunkter, samt flödet av information mot hubben. Detta gör också att övervakning av oönskad information blir svårare [10]. Det är viktigt att ha ett säkerhetstänk vid utveckling av större IoT-lösningar. En övervägning bör göras om vad som är viktigast för den specifika lösningen, enkelheten och ekonomin eller säkerheten i att informationen inte hamnar i fel händer.

Referenser

[1] Beckhoff Automation GmbH & Co., ”Beckhoff - TwinCAT,” Beckhoff Automation GmbH & Co., 15 Januari 2019. [Online]. Available: https://www.beckhoff.com/twincat/. [Använd 13 Februari 2020].

[2] X. Liu, T. Zhang, N. Hu, P. Zhang och Y. Zhang, ”The method of Internet of Things access and network communication based,” *Computer Communications,* vol. 153, pp. 169-176, 2020.

[3] Microsoft, ”What is Azure IoT Hub?,” Microsoft, 8 Augusti 2019. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-hub/about-iot-hub. [Använd 18 Februari 2020].

[4] Microsoft, ”Communicate with your IoT hub using the MQTT protocol,” Microsoft, 10 December 2018. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-mqtt-support. [Använd 18 Februari 2020].

[5] L. C. Paulson, ”Inductive Analysis of the Internet Protocol TLS,” University of Cambridge, Cambridge, 2019.

[6] Microsoft, ”Install and use Azure IoT explorer,” Microsoft, 27 December 2019. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-pnp/howto-install-iot-explorer. [Använd 3 Juni 2020].

[7] A. Kabelova och L. Dostalek, DNS in Action, Packt Publishing Limited, 2006.

[8] Beckhoff Automation GmbH & Co., *TC3 IoT Communication (MQTT),* Verl, 2019.

[9] L. Noe, ”Best Practices in SAS® 9 Security Configurations,” SAS Institute Inc., Cary, North Carolina, 2008.

[10] N. Miloslavskaya och A. Tolstoy, ”Internet of Things: information security challenges and solutions.,” *Cluster Computing,* vol. 22, nr 1, pp. 103-119, 2019.

[11] K. Shingala, ”JSON Web Token (JWT) based client authentication in Message Queuing Telemetry Transport (MQTT),” arxiv.org, Trondheim, 2018.

[12] A. S. H. Abdul-Qawy, N. M. S. Almurisi och S. Tadisetty, ”Classification of Energy Saving Techniques for IoT-basedHeterogeneous Wireless Nodes,” *Third International Conference on Computing and Network Communications (CoCoNet’19),* vol. 171, pp. 2590-2599, 2020.

Bilagor

A. Anslutningskod mellan PLC och cloudMqtt-server

Observera att domännamn, användarnamn och lösenord är exempel.

```
PROGRAM MQTT_CloudMQTT
VAR
    fbMqttClient    : FB_IotMqttClient;
    bSetParameter   : BOOL := TRUE;
    bConnect        : BOOL := TRUE;
END_VAR
 //Connection setup for MQTT hub
IF bSetParameter THEN
    bSetParameter := FALSE;
    fbMqttClient.sHostName      := 'xx.cloudmqtt.com';
    fbMqttClient.nHostPort      := 8883;
    fbMqttClient.sUserName      := 'userNamne';
    fbMqttClient.sUserPassword  := 'password';

END_IF;
//Test conection evry cycle
fbMqttClient.Execute(bConnect);
```

B. Anslutningskod mellan PLC och Azure

```
PROGRAM MQTT_Azure
VAR
    fbMqttClient    : FB_IotMqttClient;
    bSetParameter   : BOOL := TRUE;
    bConnect        : BOOL := TRUE;
END_VAR
//Connection setup for MqttClient
IF bSetParameter THEN
    bSetParameter := FALSE;
    fbMqttClient.stTLS.sCA          :=
'\Hard Disk\TwinCAT\3.1\Config\Certificates\BaltimoreCyberTrust.crt';
    fbMqttClient.stTLS.sAzureSas    := 'HostName=ExjobbAcobiaGBG.azure-
devices.net;DeviceId=device_admin;SharedAccessSignature=
SharedAccessSignature sr=ExjobbAcobiaGBG.azure-
devices.net%2Fdevices%2Fdevice_admin&sig=8gMzf%2BNSxppvj6pYeP6q0Fq6sQiwpKmVDy
qN85pGvEE%3D&se=1588323133';
END_IF;

//Test conection every cycle
fbMqttClient.Execute(bConnect);
```

C. Testprogram för funktionsblocksbibliotek

Blockuppkoppling i testprogrammet som användes när funktionsblockens olika funktioner testades.

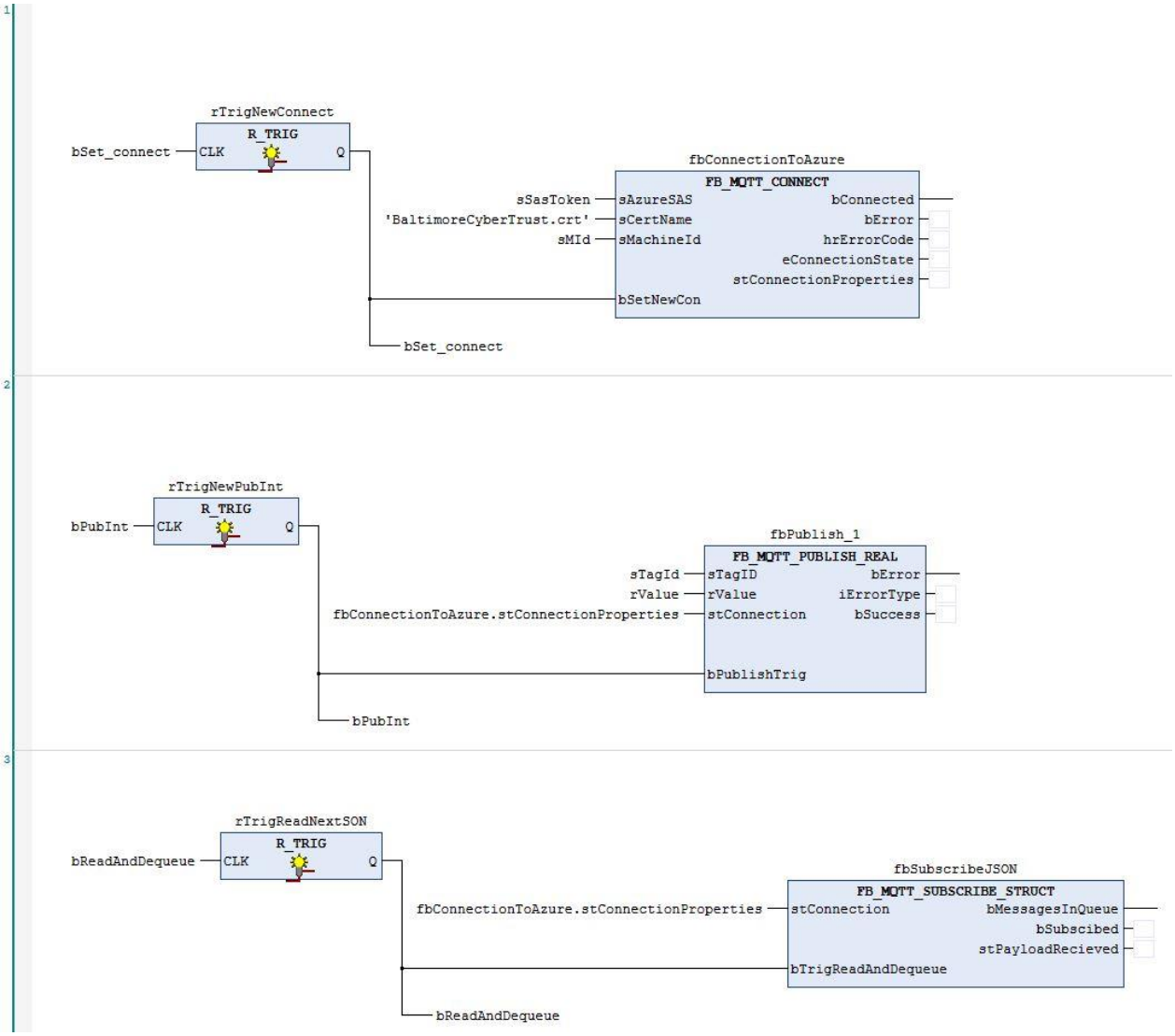


Bild C.1: Blocksammankoppling från testprogram

D. PLC kod för POC

Acobias interna taggar och SAS är reviderade.

```
VAR_GLOBAL
    I_TempSensor    AT %I*: INT;
END_VAR

PROGRAM MAIN
VAR
//  Initvalues for display
    dCurrentTemp        : REAL:=-50;
    dTempDiff           : REAL:=0.5;
    iLastTempSent       : INT:=-50;
//  Setpoint vars.
    iLastSP             : INT:= 0;
    iCurrentSP          : INT;

    bReadMessage        : BOOL := FALSE;
//  Connection vars.
    bInit               : BOOL := TRUE;
    sSAS                : STRING(511) := 'SAS';
    sTag1               : STRING(255) := 'PV';
    sTag2               : STRING(255) := 'SP';
//  Function blocks
    fb_mqttConnect      : FB_MQTT_CONNECT;
    fb_mqttPub          : FB_MQTT_PUBLISH_INT;
    fb_mqttSub          : FB_MQTT_SUBSCRIBE_SPLIT;
    tpMinSendInterval   : TP;
END_VAR

IF bInit THEN
    // Initiate connection
    fb_mqttConnect(sAzureSAS := sSAS, sCertName :=
'BaltimoreCyberTrust.crt',sMachineId := 'MId', bSetNewCon := TRUE);
    bInit := FALSE;
    // Set publish time limit
    tpMinSendInterval.PT := T#5S;

ELSE

    fb_mqttConnect(bSetNewCon := FALSE);
    // Establish subscription
    fb_mqttSub(stConnection := fb_mqttConnect.stConnectionProperties);
END_IF
// SP recieve from cloud
IF fb_mqttSub.bMessagesInQueue THEN
    fb_mqttSub.bTrigReadAndDequeue := TRUE;
    bReadMessage := TRUE;
ELSIF bReadMessage THEN
    // Control that the message is valid
    IF fb_mqttSub.sId = sTag2 AND fb_mqttSub.sSequence = '' AND fb_mqttSub.sQuality
= 'true' THEN
        iCurrentSP := STRING_TO_INT(fb_mqttSub.sValue);
        fb_mqttSub.bTrigReadAndDequeue := FALSE;
    END_IF
    bReadMessage := FALSE;
END_IF
// Run timer every cycle
tpMinSendInterval();
// Convert current sensor value
dCurrentTemp := METH_transformTemp(GVL.I_TempSensor);

// Send to cloud functions
IF fb_mqttConnect.bConnected THEN
    IF ABS(dCurrentTemp-iLastTempSent) > dTempDiff AND NOT tpMinSendInterval.Q THEN
        iLastTempSent := REAL_TO_INT(dCurrentTemp);
```

```
        fb_mqttPub(sTagId := sTag1, iValue := iLastTempSent, stConnection :=
fb_mqttConnect.stConnectionProperties, bPublishTrig:=TRUE);
        tpMinSendInterval.IN := TRUE;
    // Compare current and old sensor value. Wait 5 sec
    ELSIF iCurrentSP <> iLastSP AND NOT fb_mqttPub.bPublishTrig THEN
    fb_mqttPub(sTagId := sTag2, iValue := iCurrentSP, stConnection :=
fb_mqttConnect.stConnectionProperties, bPublishTrig:=TRUE);
    iLastSP := iCurrentSP;
    ELSE
        fb_mqttPub(bPublishTrig := FALSE);
        tpMinSendInterval.IN := FALSE;
    END_IF
END_IF;


METHOD METH_transformTemp : REAL
VAR_INPUT
    iSensorValue    : INT;
END_VAR
VAR
    iMaxInputSize   : INT := 32767;         // Max value 16 bit resolution
    dRatioForTemp   : REAL := 0.003059;     // degrees per step
END_VAR

METH_transformTemp := (iSensorValue-(iMaxInputSize/2))*dRatioForTemp;
```