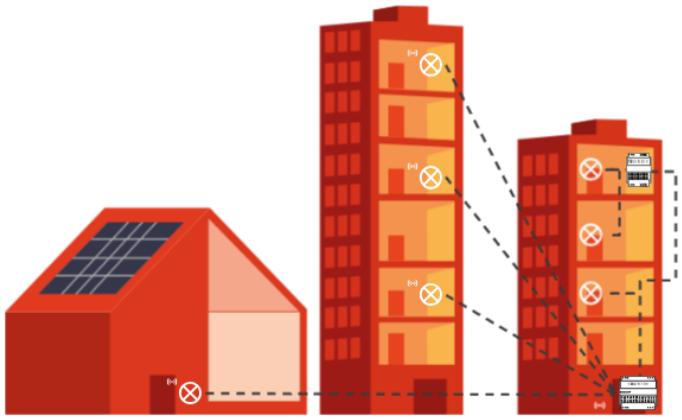
CHALMERS



PiiGAB Flexibility. Simplicity. Enability.

Automatiserad och AI-driven testning för inbyggda system

Ett Robot Framework-baserat testsystem för verifiering av M-Bus- och Modbus-kommunikation i IoT-gateways

Examensarbete inom högskoleprogrammet Datateknik

Richard Emmerstorfer,
Sebastian Nguyen

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg 2026
www.chalmers.se

Examensarbete 2026

Automatiserad och AI-driven testning för inbyggda system

Ett Robot Framework-baserat testsystem för verifiering av M-Bus- och Modbus-kommunikation i IoT-gateways

Richard Emmerstorfer,
Sebastian Nguyen

CHALMERS

Institutionen för data- och informationsteknik
Chalmers Tekniska Högskola
Göteborg 2026

Automatiserad och AI-driven testning för inbyggda system
Ett Robot Framework-baserat testsystem för verifiering av M-Bus- och Modbus-kommunikation i IoT-gateways
Richard Emmerstorfer & Sebastian Nguyen

Handledare: Karl Lindell, Utvecklingsansvarig, PiiGAB
Handledare: Sakib Sistek, Data- och informationsteknik, Chalmers
Examinator: John J. Camilleri, Data- och informationsteknik, Chalmers

Examensarbete 2026
Institutionen för data- och informationsteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Omslagsbild: Visar hur infrastrukturen för PiiGABs gateways kan se ut i verkliga fastigheter.

Skriven i LaTeX
Göteborg 2026

Automatiserad och AI-driven testning för inbyggda system
Ett Robot Framework-baserat testsystem för verifiering av M-Bus- och Modbus-kommunikation i IoT-gateways
Richard Emmerstorfer & Sebastian Nguyen
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola

Abstract

This thesis investigates how a manual testing process for embedded gateway systems can be automated, and how artificial intelligence can be used as support in such a process. The work was carried out in collaboration with PiiGAB and focused on the company's IoT gateways, which are used for collecting and forwarding measurement data through protocols such as M-Bus and Modbus.

The project resulted in a modular automated test environment based on Python and Robot Framework. The system can execute test sequences, handle external configuration files, collect and compare measurement data, and generate summarized test reports. Several tests were implemented to verify gateway communication and functionality.

The modular structure makes it possible to extend the test environment with additional protocols, meters, and test scenarios in the future. However, the work also shows that reliable automated testing depends on controlled test data, correct configuration, and verification of the test system itself.

AI was evaluated as a possible support tool during the project. It proved useful for information gathering, documentation, idea generation, and general reasoning about test cases. However, it was less reliable when detailed protocol-specific functionality had to be implemented, especially regarding M-Bus structure. Therefore, AI was not integrated directly into the final test application. The conclusion is that AI can support the testing process, but it cannot replace human technical review and verification.

Keywords: Embedded Systems, IoT, Gateway, Robot Framework, Automated Testing, M-Bus, Modbus.

Förord

Detta examensarbete har genomförts vid Chalmers tekniska högskola i samarbete med PiiGAB i Mölnlycke. Arbetet har varit mycket lärorikt och har gett oss möjlighet att fördjupa våra kunskaper inom automatiserad testning, inbyggda system och industriell kommunikation.

Vi vill rikta ett stort tack till vår handledare på PiiGAB, Karl Lindell, för teknisk vägledning, återkoppling och stöd under arbetets gång. Vi vill även tacka vår handledare på Chalmers, Sakib Sistek, för värdefulla synpunkter kring rapportens struktur, metodik och akademiska innehåll. Ett tack riktas även till vår examinator John J. Camilleri.

Ett särskilt stort tack riktas till alla på PiiGAB som har varit behjälpliga under projektets gång. Den hjälp, kunskap och tillgänglighet som vi fått ta del av har varit mycket värdefull, både vid förståelsen av företagets produkter och vid lösning av tekniska problem. Vi uppskattar särskilt att vi har fått möjlighet att arbeta på plats hos företaget, vilket har gett oss tillgång till hårdvara, interna verktyg och direkt kontakt med personer med stor teknisk erfarenhet.

Richard Emmerstorfer & Sebastian Nguyen, Göteborg, Maj 2026

Nomenklatur

Nedan presenteras nomenklaturen för de begrepp som används i detta examensarbete.

Akronymer

IoT	(Internet of Things) avser uppkopplade fysiska enheter med sensorer och beräkningsförmåga som samlar in, kommunicerar och utbyter data via kommunikationsnätverk.

Begrepp

Mjukvara	Immateriella instruktioner som styr funktion hos ett system eller en enhet.
Hårdvara	Fysiska komponenter i en enhet.
Master	Styrande enhet som initierar och kontrollerar kommunikation.
Slave	Underordnad enhet som svarar på förfrågningar, exempelvis en mätare.
Bit	Den minsta informationsenheten i ett digitalt system, som kan vara värdet 0 eller 1.
Byte	En grupp om åtta bitar som tillsammans representerar ett tecken eller ett numeriskt värde.
Hexadecimal	Ett talsystem med basen 16 som använder siffrorna 0–9 och bokstäverna A–F för att representera binära värden i kompakt form.
Webbgränssnitt	Grafiskt gränssnitt som nås via webbläsare.

Innehåll

Nomenklatur **ix**

Figurer **xv**

Tabeller **xvii**

1 Inledning **1**
1.1 Bakgrund . . . 1
1.2 Syfte och frågeställning . . . 2
1.3 Mål . . . 2
1.4 Avgränsningar . . . 2

2 Metod **3**
2.1 Arbetssätt . . . 3
2.2 Etiska och ekologiska aspekter . . . 4
2.3 Tidsplan . . . 4
2.4 Val av tekniker . . . 5
2.4.1 Programmeringsspråk . . . 5
2.4.2 Testramverk . . . 5
2.4.3 Versionshantering . . . 6
2.4.4 Kodstruktur . . . 6
2.4.5 AI-modell . . . 6
2.4.6 Hårdvara . . . 7
2.5 Informationsinhämtning . . . 7
2.5.1 Handledning . . . 7
2.6 Metodanalys . . . 8

3 Teknisk bakgrund **9**
3.1 Datakommunikation . . . 9
3.1.1 Applikationslager . . . 10
3.1.2 Transportlager . . . 10
3.1.3 Länklager . . . 11
3.2 TCP . . . 11
3.3 UDP . . . 12
3.4 M-Bus . . . 12
3.4.1 Telegramstruktur . . . 13
3.4.2 Initiering av M-Bus-kommunikation . . . 15

3.4.3 Singel- och multitelegram . . . 16
3.4.4 M-Bus över TCP . . . 17
3.4.5 Wireless M-Bus . . . 18
3.5 Modbus . . . 18
3.6 Gateway . . . 19
3.6.1 PiiGAB 900 . . . 20
3.7 Modifierad HanP1-konverterare . . . 20
3.7.1 Mätarsimulator . . . 21

4 Genomförande **23**
4.1 Intern utbildning . . . 23
4.2 Initial testmiljö . . . 24
4.3 Uppsättning av laborationsmiljö . . . 24
4.3.1 Felsökning av Laborationsmiljön . . . 26
4.4 Implementering av första testet . . . 27
4.4.1 Json2Telegram . . . 27
4.4.2 Data Test . . . 28
4.5 Utökad testning . . . 28
4.5.1 Read Test . . . 28
4.5.2 Package Install/Uninstall . . . 29
4.5.3 Corrupted Test . . . 30
4.5.4 Timeout Test . . . 30
4.5.5 Value Test . . . 30
4.5.6 Load Test . . . 31
4.5.7 Compare Test . . . 32
4.6 Primäradressering . . . 32
4.7 Förenklad testkörning . . . 33
4.7.1 Konfigurering av tester . . . 33
4.7.2 Dokumentation . . . 34
4.8 Verifiering . . . 35
4.8.1 Manuell avläsning . . . 35
4.8.2 Enhetstester . . . 35
4.8.3 Fysisk verifiering . . . 36

5 Resultat **37**
5.1 Programstruktur . . . 37
5.2 Implementerade tester . . . 39
5.2.1 Data Test . . . 40
5.2.2 Read Test . . . 40
5.2.3 Packet Install/Uninstall . . . 41
5.2.4 Corrupted Test . . . 41
5.2.5 Timeout Test . . . 42
5.2.6 Value Test . . . 42
5.2.7 Load Test . . . 43
5.2.8 Compare Test . . . 44

6 Diskussion **45**

6.1 Planerad tidsplan mot faktisk tidsplan 45
6.2 Tekniska val och arkitektur 46
6.3 Testsystemets faktiska nytta 46
6.4 Teststrategier och verifiering 47
6.5 Testdata 47
6.6 Konfiguration, exekvering och rapportering 48
6.7 Användarvänlighet 49
6.8 AI som stöd i testprocessen 49
6.9 Uppfyllelse av krav 50
6.10 Etiska, samhälleliga och ekologiska aspekter 50
6.11 Arbete i verklig miljö 51
6.12 Begränsningar och fortsatt arbete 51
6.13 Sammanfattande diskussion 51

7 Slutsats **53**

Litteraturförteckning **55**

Figurer

2.1 Preliminära tidsplanen i form av ett Gantt-schema. 5

3.1 Exempel på M-Bus-telegram i *long frame*-format där 68_{16}, L1, L2, 68_{16} markerats och även checksumman $3D_{16}$ följt av stoppbyte 16_{16}. . 14
3.2 Produktbild på PiiGAB 900S, [14] 19
3.3 Modifierad HanP1-konverterare kopplad till gateway. 21

4.1 Laborationsmiljön som användes för testningen. 25

5.1 Förenklad systemöversikt över testsystemets komponenter. 38
5.2 Sekvensdiagram över ett REQ_UD2-kommando. 40

Tabeller

3.1 TCP/IP-modellens olika lager,[6]. 9

4.1 Enhetstest för `Verify Ack`-keyword 36
4.2 Enhetstest för verifiering av att telegram tillhör rätt mätare 36

5.1 Verifiering av `RSP_UD` (mätarsvar) 40
5.2 Verifiering av `RSP_UD` (mätarsvar) 40
5.3 Installation av paket med opkg via SSH 41
5.4 Avinstallation av paket med opkg via SSH 41
5.5 Verifiering av korrupt data från mätare 41
5.6 Verifiering av timeout-hantering 42
5.7 Verifiering av värde från mätare 42
5.8 Verifiering av mätares sekundäradress 43
5.9 Jämförelse av värde mellan M-Bus och Modbus 44

1

Inledning

1.1 Bakgrund

PiiGAB är ett företag som ligger i Mölnlycke där företaget designar, utvecklar och slutmonterar mjuk- och hårdvarulösningar för mätvärdesinsamling, kommunikation och systemintegration inom fastighets- och industrisektorn. Företagets lösningar används för att samla in och vidareförmedla mätdata, exempelvis temperatur och elförbrukning från olika typer av sensorer och mätare till molnbaserade tjänster.

Den mest centrala delen av PiiGAB:s hårdvaruprodukter utgörs av gateways, exempelvis deras PiiGAB 900-serie. Dessa gateways fungerar som kommunikationsnoder som kopplar samman mätvärden från mätare för el, temperatur och vatten med molntjänster för mätvärdesinsamling. Kommunikationen sker via flera olika protokoll såsom M-Bus, Wireless M-Bus, Modbus, TCP, UDP och så vidare.

På senare år har mjuk- och hårdvaruutvecklingen på PiiGAB gått mycket fort och allt nytt måste ständigt testas för att garantera produkternas robusthet. I och med att deras gateways använder sig av flera olika kommunikationsprotokoll, leder det till att testning av alla kommunikationsvägar manuellt blir mycket tids- och resurskrävande. Det finns därför ett behov av att automatisera denna testning så den kräver mindre manuellt arbete.

Kommunikationsprotokollen testas idag genom att en testrigg byggs upp där anställda på företaget manuellt verifierar att den data som skickas in i en gateway överensstämmer med det förväntade resultatet. Dessa tester behöver dessutom genomföras i varierande miljöer, i syfte att även hitta de extremt ovanliga utfallen, så kallade *edge cases*. Detta gör testningen ännu mer tids- och resurskrävande. Eftersom testmiljön i sig inte fullt ut kan återskapa en verklig miljö krävs dessutom ett stort antal tester för att kunna garantera produkternas kvalitet och tillförlitlighet. Företaget vill även effektivisera testningen genom att implementera en AI-modell som kan skriva egna tester samt underlätta vid avläsning av testresultat.

1.2 Syfte och frågeställning

Genom arbetet ska det undersökas hur testningen av PiiGAB:s mjukvara kan automatiseras samt vilka möjligheter som finns att integrera en AI-modell i testprocessen. Frågor som kommer att besvaras genom arbetet är:

- På vilket sätt kan automatiserad testning utformas för att passa PiiGAB:s 900-serie?
- Hur kan automatiserad testning implementeras med systemkrav från PiiGAB?
- Hur förhåller sig AI-genererade tester till manuellt skrivna tester i kvalitet vid uppbyggnaden av en automatiserad testmiljö?
- Möter lösningen PiiGAB:s krav på tillförlitlighet eller krävs fortsatt mänsklig kontroll för att säkerställa korrekthet?

1.3 Mål

Målet med projektet är att implementera en väl dokumenterad automatiserad testmiljö för dataöverföring via olika protokoll utifrån PiiGAB:s behov samt framställa en rapport utifrån arbetet. Vid projektets slut förväntas följande mål ha uppnåtts:

- Utveckla ett testsystem för körande av automatiserade tester samt insamling, loggning och sammanställning av testresultat.
- Undersöka hur en AI-modell kan integreras i systemet för analys och bearbetning av insamlad testloggning samt generering av tester.
- Etablera en fysisk testmiljö bestående av relevant hårdvara där tester kan genomföras och verifieras.
- En modulärt utformad testmiljöarkitektur för att möjliggöra enkel vidareutveckling.

1.4 Avgränsningar

Följande avgränsningar har gjorts för att säkerställa projektets relevans för PiiGAB samt för att hålla arbetet inom den givna tidsramen:

- Den automatiserade testmiljön utvecklas enbart för PiiGAB:s produkter och anpassas efter företagets specifika behov.
- Den slutgiltiga produkten kommer inte att publiceras för allmän beskådan.

2

Metod

Det finns många olika sätt att arbeta på och även sätt att välja tekniska specifikationer. Därför kommer detta kapitel att behandla hur projektet tagit ställning till olika arbetssätt, samt val av tekniker.

2.1 Arbetssätt

Projektet har bedrivits med ett agilt inspirerat arbetssätt. Att arbeta agilt innebär enligt M. Comstedt [1] ett förhållningssätt där flexibilitet, kontinuerlig återkoppling och successiv utveckling står i centrum. Istället för att följa en strikt, förutbestämd plan delas arbetet upp i mindre delar som löpande utvärderas och justeras utifrån nya insikter och förändrade förutsättningar.

Arbetet strukturerades genom veckovisa målsättningar i kombination med återkommande avstämningsmöten med handledaren på företaget. Dessa kompletterades med kortfattade veckorapporter som syftade till att tydliggöra genomförda aktiviteter, identifierade hinder samt planerade nästa steg. Detta möjliggjorde ett iterativt arbetssätt med kontinuerlig reflektion och möjlighet till omprioritering vid behov [1].

Istället för att tillämpa ett formellt ramverk såsom Scrum till hundra procent, där arbetet organiseras i tidsbestämda *sprints* med fördefinierad längd och uppgifter, användes ett mer flexibelt arbetssätt utan strikt satta iterationer. Dessutom användes ingen traditionell Kanban-tavla, det vill säga en tavla där arbetsuppgifter delas upp efter status, exempelvis planerade, pågående och färdiga uppgifter [2]. Istället användes en detaljerad checklista för att strukturera och följa upp arbetsuppgifter. Scrum-ramverket sågs mer som ett stöd för arbetet snarare än ett mål att uppnå i sig. Projektgruppen bestod endast av två personer och därför bedömdes ett förenklat och mer anpassningsbart arbetssätt vara mer passande än en strikt tillämpning av Scrum-ramverket.

Arbetet planerades att huvudsakligen genomföras gemensamt på plats vid företagets kontor. Syftet med detta var att underlätta kontinuerlig dialog, snabb återkoppling och gemensam problemlösning, vilket ligger i linje med agila principer om samarbete och anpassningsförmåga [1]. Närvaro på plats var även viktig för att ge tillgång till nödvändig hårdvara, arbetsdatorer och interna system, samt för att möjliggöra di-

rekt kontakt med tekniskt kunnig personal vid behov av vägledning eller felsökning.

2.2 Etiska och ekologiska aspekter

Projektet har genomförts med hänsyn till både etiska och ekologiska aspekter. Eftersom arbetet skedde i nära samarbete med PiiGAB och berörde företagets produkter, testmiljöer och interna arbetssätt, behövde informationshanteringen ske med försiktighet. För att minska risken att företagsintern eller känslig information inkluderades i rapporten delades material löpande med företagets handledare för granskning. På så sätt kunde rapportens innehåll kontrolleras innan det färdigställdes.

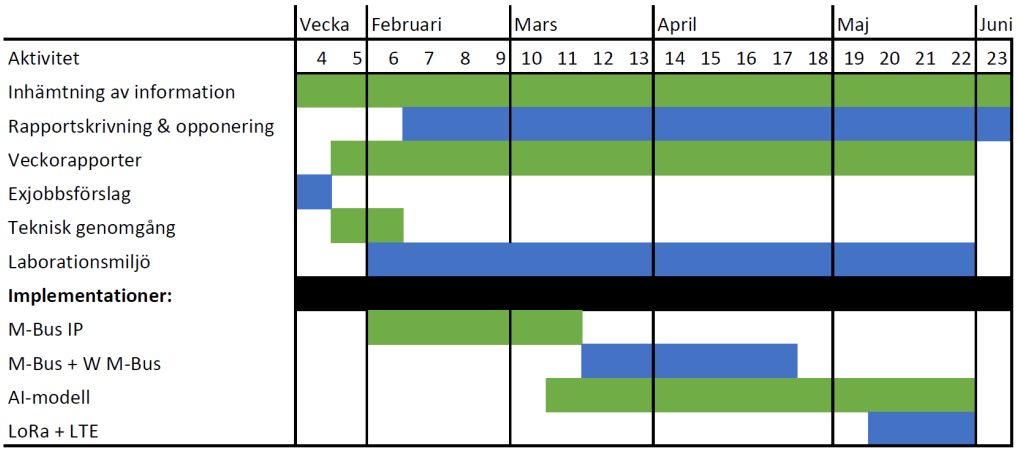
Användningen av AI-verktyg hanterades också med försiktighet. Eftersom en del av projektets syfte var att undersöka hur AI kan stödja testprocessen användes AI i viss utsträckning, exempelvis som stöd vid informationsinhämtning. Samtidigt användes inte AI som ensam grund för tekniska beslut eller implementationer. Kod och tekniska lösningar granskades och verifierades manuellt innan de användes. Endast AI-verktyg som bedömdes lämpliga ur informationssäkerhetsperspektiv användes, i syfte att undvika att företagsintern information lagrades eller spreds utanför projektet.

De ekologiska aspekterna hanterades främst genom att testmiljön hölls så begränsad och resurseffektiv som möjligt. I stället för att använda större servrar eller mer omfattande testutrustning byggdes testmiljön upp med befintlig hårdvara från företaget, exempelvis gateways, mätare och datorer. Simulerade mätare användes även, vilket minskade behovet av ytterligare fysisk utrustning. Detta gjorde det möjligt att genomföra tester utan att bygga upp en större eller mer resurskrävande miljö än nödvändigt.

Projektet hade därmed ingen större direkt miljöpåverkan, men arbetet bedrevs med målet att undvika onödig resursanvändning och att använda befintliga resurser på ett effektivt sätt.

2.3 Tidsplan

En preliminär tidsplan skapades i form av ett Gantt-schema, vilket illustrerade hur projektets olika arbetsmoment planerades att genomföras över tid. Schemat omfattade vecka 4 till 23 och täckte därmed hela projektets genomförande. Gantt-schemat användes som planeringsverktyg då det möjliggjorde en tydlig visualisering av aktiviteternas tidsmässiga placering samt överlapp mellan de, vilket underlättade både planering och uppföljning.



Figur 2.1: Preliminära tidsplanen i form av ett Gantt-schema.

2.4 Val av tekniker

I detta avsnitt presenteras de tekniska val som gjordes för projektets genomförande. Valen omfattar programmeringsspråk, testramverk, versionshantering, kodstruktur, AI-modell och hårdvara. Syftet är att motivera varför dessa tekniker användes och hur de förhåller sig till projektets behov av automatiserad testning, modularitet och kompatibilitet med PiiGAB:s befintliga systemmiljö.

2.4.1 Programmeringsspråk

Projektet bygger på befintliga moduler som har skrivits i Python. Det blev därför ett naturligt val att fortsätta utvecklingen i samma programmeringsspråk för att säkerställa kompatibilitet, minska integrationsarbete och möjliggöra återanvändning av befintlig kod. Python erbjuder dessutom hög läsbarhet och ett omfattande ekosystem av bibliotek, vilket kan underlätta utveckling och automatisering [3]. Detta var särskilt fördelaktigt i ett projekt där både systemintegration och testautomatisering har varit centrala delar.

En möjlig begränsning med Python är att det är ett tolkat språk, vilket generellt kan innebära lägre exekveringsprestanda jämfört med kompilerade språk såsom C++ eller Java. I detta projekt har dock prestanda på exekveringsnivå inte varit avgörande, vilket gör att fördelarna med utvecklingshastighet och läsbarhet bedömdes väga tyngre.

2.4.2 Testramverk

I början av projektet genomfördes en förstudie för att välja lämpligt testramverk. Valet föll på *Robot Framework*, ett etablerat och väldokumenterat ramverk för testautomatisering [4]. Ramverket stödjer *keyword-driven testing*, vilket innebär att testfall kan skrivas på en hög abstraktionsnivå och struktureras på ett sätt som är

lättläst och underhållbart [4]. Detta möjliggör att testfall kan förstås även av personer utan djup programmeringsbakgrund, vilket var fördelaktigt i projektets kontext där flera olika roller på företaget kan komma att behöva ta del av testresultat.

Robot Framework har även god integration med Python och möjliggör att Python-bibliotek kan användas direkt i testmiljön [4]. Ramverket erbjuder dessutom inbyggt stöd för rapportgenerering, loggning och visualisering av testresultat, vilket underlättar felsökning och kvalitetssäkring. En möjlig begränsning är att den högre abstraktionsnivån kan innebära mer omgivande struktur än mer kodnära testverktyg, exempelvis *pytest*. I detta projekt bedömdes dock behovet av tydliga testfall och lättillgängliga rapporter väga tyngre än denna nackdel.

2.4.3 Versionshantering

För versionshantering användes Git, vilket är ett distribuerat versionshanteringssystem med brett stöd inom mjukvaruutveckling [5]. Git möjliggör spårbarhet av kodändringar, parallell utveckling via så kallade branches och återställning till tidigare versioner vid behov. Projektets kodbas var sedan tidigare placerad på GitHub, och därför bedömdes det inte vara motiverat att byta till en annan plattform, exempelvis GitLab. Under projektet användes separata branches för nya funktioner och större förändringar innan dessa fördes in i huvud-branchen. Detta gjorde det möjligt att testa nya lösningar utan att påverka den stabila kodbasen, samtidigt som tidigare versioner och ändringar kunde följas.

2.4.4 Kodstruktur

Kodbasen struktureras modulärt i syfte att separera kommunikationslogik, datahantering och testrelaterad funktionalitet. En modulär arkitektur valdes för att möjliggöra eventuell återanvändning av delar i testsystemet i andra program. Denna metodik underlättar även testning och verifiering, då enskilda moduler kan utvecklas och verifieras oberoende av varandra. Vidare bidrar strukturen till förenklad vidareutveckling i framtiden, exempelvis vid integration av ytterligare protokoll eller funktioner i testsystemet.

2.4.5 AI-modell

Företaget har ett företagskonto hos OpenAI. Den aktuella AI-modellen, GPT-5.5, har använts för att undersöka hur testskrivning och logganalys skulle kunna utföras av modellen. Den har använts för att förklara grundläggande principer för vilka typer av tester som är lämpliga att inkludera i en teststruktur, samt för att generera pseudokod för tester i syfte att utvärdera modellens effektivitet vid testgenerering. Modellen har även använts för att ta emot loggar i syfte att stödja tolkning och felanalys. Resultaten från modellen har använts i varierande grad.

2.4.6 Hårdvara

All hårdvara som har använts består av PiiGAB:s egna produkter eller tillgångar, från datorer till fysiska gateways och mätare. Detta val har gjorts eftersom projektet är exklusivt anpassat till PiiGAB:s systemmiljö. Därigenom etableras en grundförutsättning som säkerställer att projektet under alla förhållanden är kompatibelt med PiiGAB:s egna produkter.

2.5 Informationsinhämtning

Informationsinhämtningen i projektet har genomförts med hjälp av flera kompletterande källor för att säkerställa både bredd och djup i det teoretiska underlaget. Ett centralt inslag i informationsinsamlingen har utgjorts av material från PiiGAB, inklusive interna utbildningar, teknisk dokumentation samt tekniska standarder för relevanta kommunikationsprotokoll. Detta material har varit särskilt värdefullt då det innehåller detaljerad och praktisk information som i många fall inte finns tillgänglig i publicerade källor.

Utöver interna resurser har även digitala källor använts i form av vetenskapliga artiklar. Dessa källor har använts för att studera tekniska principer och för att stärka arbetets vetenskapliga förankring. Samtidigt var detta informationssätt mer tidskrävande, eftersom det inte alltid var enkelt att hitta artiklar som var direkt relevanta för projektets specifika tekniska område. Det krävdes därför tid både för att söka efter lämpligt material och för att läsa igenom artiklarna för att avgöra om de kunde användas som stöd i arbetet.

AI användes även som stöd vid informationsinhämtningen. Verktyget användes främst för att få översiktliga förklaringar av tekniska begrepp, identifiera relevanta områden att undersöka vidare och sammanfatta komplex information. Informationen som togs fram med hjälp av AI användes dock inte som ensam källa, utan behövde kontrolleras mot interna dokument, tekniska standarder eller andra mer tillförlitliga källor innan den användes i rapporten.

2.5.1 Handledning

Under projektets gång har handledning funnits tillgänglig både från PiiGAB och från Chalmers tekniska högskola. Från företagets sida har en utsedd handledare ansvarat för den tekniska vägledningen i projektet. Företagets handledare, Karl Lindell, som är utvecklingschef på PiiGAB, har bidragit med omfattande kunskap om företagets produkter, systemarkitektur och de tekniska krav som dessa system behöver uppfylla. Utöver den formella handledningen har även andra anställda på företaget bidragit med värdefulla insikter, särskilt vid idéutveckling och problemlösning relaterad till projektets tekniska utmaningar.

Från Chalmers sida har projektet haft en akademisk handledare, Sakib Sistek, vars fokus främst har varit på rapportens struktur, metodik och vetenskapliga kvalitet. Den akademiska handledningen har bidragit till att säkerställa att arbetet uppfyller

de krav som ställs på ett examensarbete och har gett vägledning kring hur delar i rapporten, såsom resultat och metod, ska presenteras på ett tydligt och vetenskapligt sätt.

2.6 Metodanalys

I ett projekt av denna typ, som kännetecknas av en begränsad tidsram och en tydlig koppling till ett specifikt företag, tenderar metodvalen att bli företagsinriktade. Detta beror dels på att viss information är konfidentiell och företagspecifik, dels på att arbetet inriktas mot ett fåtal specifika produkter inom samma företag.

Även om den interna kunskapen om företagets produkter är omfattande, blir kunskapen i praktiken specialiserad. Därav, även om metodvalen är välmotiverade och teoretiskt skulle kunna tillämpas i andra projekt, har projektets företagsanknytning ändå utgjort en betydande faktor vid dessa val.

Med detta i åtanke kan det konstateras att vissa metodval hade kunnat se annorlunda ut om projektet haft en mer generell inriktning, större oberoende från företagsmiljön eller en längre tidsram. Exempelvis hade informationsinhämtningen kunnat baseras mer på externa och vetenskapliga källor, medan de tekniska valen hade kunnat utvärderas bredare mot alternativa lösningar. Även valet av programmeringsspråk hade kunnat tänkas om exempelvis ifall exekveringsprestanda hade varit en mer central faktor. I detta projekt bedömdes dock kompatibilitet med befintlig kod, tillgång till intern kunskap och praktisk användbarhet i PiiGAB:s miljö väga tyngre.

3

Teknisk bakgrund

Detta kapitel beskriver den tekniska bakgrund som behövs för att förstå projektets sammanhang och de system som testsystemet utvecklades för. Kapitlet behandlar först grundläggande datakommunikation och de lager som används för att beskriva hur data överförs mellan olika system. Därefter presenteras de kommunikationsprotokoll som är relevanta för arbetet, med särskilt fokus på M-Bus och Modbus. Avslutningsvis beskrivs PiiGAB:s 900-serie och den modifierade HanP1-konverterare som användes i laborationsmiljön.

3.1 Datakommunikation

Datakommunikation handlar om hur information struktureras, överförs och tolkas mellan olika system i ett nätverk [6]. Kommunikationen delas ofta upp i olika lager, där varje lager ansvarar för en specifik uppgift i överföringen av data.

Varje lager tillhandahåller tjänster till lagret ovanför och använder tjänster från lagret under. Genom denna uppdelning kan olika protokoll utvecklas och implementeras oberoende av varandra, samtidigt som de tillsammans möjliggör fullständig kommunikation mellan systemen.

Det finns två vanligt förekommande modeller för att förklara de olika lagren inom datakommunikation. Den första är OSI-modellen (Open Systems Interconnection) som består av sju lager och används ofta för att beskriva hur nätverkskommunikation fungerar. Den andra modellen är TCP/IP-modellen, som består av färre lager.

I detta arbete används TCP/IP-modellen för att beskriva datakommunikation eftersom flera av de protokoll som behandlas i arbetet, exempelvis TCP och UDP, är definierade inom denna modell och används direkt ovanpå IP-nätverk. Tabellen 3.1 nedan visar en förenklad översikt av lagren i TCP/IP-modellen tillsammans med exempel på protokoll som förekommer i respektive lager.

Tabell 3.1: TCP/IP-modellens olika lager,[6].

Lager	Exempel på protokoll	Beskrivning
Applikationslager	M-Bus, Modbus, FTP, SSH	Hur data tolkas och används av applikationer, exempelvis mätvärden eller kommandon.
Transportlager	TCP, UDP	Överföring av data mellan två system samt hantering av exempelvis segmentering och tillförlitlighet.
Internetlager	IP	Adressering och routing av datapaket mellan olika nätverk.
Länklager	Ethernet, RS-485, seriell buss	Hanterar överföring av data mellan direkt anslutna enheter över ett specifikt medium.

Med seriell buss avses här en kommunikationsförbindelse där data överförs bit för bit mellan anslutna enheter över ett gemensamt kommunikationsmedium, exempelvis en trådbunden förbindelse.

3.1.1 Applikationslager

Applikationslagret beskriver hur data tolkas och används av program eller system [6]. På denna nivå definieras betydelsen av de data som skickas, exempelvis vilka värden som representerar mätdata, kommandon eller statusinformation.

Applikationsprotokoll specificerar hur olika typer av meddelanden ska struktureras och hur olika enheter ska interagera med varandra. Dessa protokoll använder i sin tur transportprotokoll för att överföra data mellan system.

Exempel på applikationsprotokoll är M-Bus och Modbus, vilka används i många industriella system för att läsa av mätvärden och styra utrustning [7]. Andra exempel är nätverksbaserade protokoll såsom FTP och SSH, vilka används för filöverföring respektive fjärrkommunikation.

3.1.2 Transportlager

Transportlagret ansvarar för den logiska kommunikationen mellan applikationer som körs på olika enheter i ett nätverk. Lagret tar emot data från applikationslagret och delar vid behov upp informationen i mindre segment för överföring. Vid mottagarsidan återmonteras segmenten till den ursprungliga dataströmmen innan informationen vidarebefordras till rätt applikation.

En central uppgift för transportlagret är att säkerställa att data levereras till korrekt applikation genom användning av portnummer. Lagret kan även hantera funktioner såsom felkontroll, flödeskontroll och omsändning av förlorade datapaket, beroende på vilket transportprotokoll som används. På så sätt fungerar transportlagret som en

länk mellan applikationsprotokollen och de underliggande nätverkslagren, samtidigt som det tillhandahåller tjänster som möjliggör tillförlitlig och effektiv kommunikation mellan två noder i nätverket.

3.1.3 Länklager

Länklagret beskriver hur data överförs mellan direkt anslutna enheter över ett specifikt kommunikationsmedium. Detta lager ansvarar för hur data paketeras i ramar eller telegram samt hur felkontroll och adressering hanteras på den lokala länken [6].

På denna nivå definieras även hur data representeras som elektriska eller andra fysiska signaler i överföringsmediet. Exempel på tekniker i detta lager är Ethernet samt seriella kommunikationsgränssnitt såsom RS-232 och RS-485.

I industriella system används ofta RS-485 för robust seriell kommunikation över längre avstånd [7]. Vissa bussystem definierar dessutom både signalering och strömförsörjning i samma standard. Ett exempel är M-Bus, där en masterenhet kommunicerar med flera slavenheter över en tvåledarbuss samtidigt som slavarna kan få sin strömförsörjning via samma ledningar [8].

3.2 TCP

Transmission Control Protocol (TCP) är ett kommunikationsprotokoll som kännetecknas av tillförlitlig dataöverföring mellan två nätverksenheter [6]. TCP är anslutningsorienterat, vilket innebär att en kommunikationssession etableras mellan två enheter innan data utbyts. Paketförlust, även kallat *packet loss* och paket som mottas i fel ordning hanteras också av protokollet. Dessa egenskaper gör TCP särskilt lämpligt när det är viktigt att all data överförs i rätt ordning.

För att säkerställa tillförlitlig överföring använder TCP flera mekanismer. Varje byte i dataströmmen tilldelas ett sekvensnummer, vilket gör det möjligt för mottagaren att rekonstruera data i korrekt ordning. Mottagaren skickar dessutom kvittensmeddelanden (*acknowledgements*, ACK) för att bekräfta att data har mottagits korrekt. Om en kvittens uteblir inom en viss tidsperiod antar sändaren att data har gått förlorad och skickar om den. TCP implementerar även flödeskontroll och överbelastningskontroll för att anpassa överföringshastigheten efter mottagarens kapacitet och nätverkets aktuella belastning [6]. Dessa mekanismer gör att protokollet kan leverera data på ett robust sätt även i nätverk oavsett nätverkets stabilitet.

Egenskaperna hos TCP gör protokollet särskilt lämpligt för applikationer där korrekt och fullständig dataöverföring är avgörande. Ett exempel är File Transfer Protocol (FTP), där filer överförs mellan klient och server över en TCP-anslutning [6]. Även Secure Shell (SSH) använder TCP som transportprotokoll för att etablera säkra fjärranslutningar mellan system.

3.3 UDP

User Datagram Protocol (UDP) är ett annat kommunikationsprotokoll som används för att överföra data mellan nätverksenheter över ett nätverk [6]. Till skillnad från TCP är UDP anslutningslöst, vilket innebär att ingen explicit anslutning etableras mellan sändare och mottagare innan data skickas. Data överförs istället i form av fristående paket som skickas direkt till mottagaren utan inledande "handskakning".

En viktig egenskap hos UDP är att protokollet är relativt enkelt och inte lika resurskrävande jämfört med TCP [6]. Detta eftersom UDP inte hanterar varken kvittensmeddelanden, omsändning av förlorade paket eller ordning av datapaket. Det innebär att protokollet varken garanterar att all data levereras, ordningen eller att den duplicerats. Om sådana funktioner krävs måste de implementeras i applikationslagret.

Till skillnad från TCP, behandlas varje UDP-paket oberoende av tidigare eller efterföljande paket [6]. Detta gör att mottagaren kan läsa ett komplett paket i taget utan att behöva rekonstruera en hel dataström.

På grund av dess simplicitet används UDP ofta i system där låg fördröjning är viktigare än absolut tillförlitlighet, exempelvis i röst- och videosamtal samt strömning av video och ljud, eller i samband med protokoll där applikationen själv hanterar eventuell felkontroll [6].

3.4 M-Bus

Meter-Bus, allmänt känt under förkortningen M-Bus, är ett kommunikationsprotokoll som utvecklats vid Paderborn Universitet i Tyskland [9]. Det karakteristiska med M-Bus-protokollet är att det är telegrambaserat, kostnadseffektivt och kan färdas långa sträckor. Protokollet skickar alltså hela telegram i form av paket varje gång data skickas mellan enheter. Idag är det ett europeiskt standardiserat kommunikationsprotokoll avsett för fjärravläsning av mätare och sensorer inom områden som energi- och vattenförbrukning, värmemätning samt andra konsumtionsrelaterade mätvärden [10].

Enligt PiiGAB [7] är anledningen till varför M-Bus blivit så populärt att det är mycket kostnadseffektivt och att trådbunden M-bus kan färdas flera kilometer med en robust signal, vilket gör den lämplig för användning i stora byggnader. Protokollet är uppbyggt på master-slave-arkitekturen. PiiGAB har även nämnt att det i praktiken är vanligast att deras gateway är master och mätare är slavar. Det finns även möjlighet för PiiGAB:s gateways att vara master för en slav-gateway om det krävs, vilket gör det möjligt att utöka nätverket. Trådbunden M-Bus har även möjlighet att försörja slav-enheten med elektricitet, vilket gör installationen enkel och ännu mer kostnadseffektiv.

Enligt interna utbildningar på PiiGAB [7] är kommunikationen i ett trådbundet M-

Bus-system halvduplex, vilket innebär att dataöverföring kan ske i båda riktningarna, men inte samtidigt. Överföringen bygger på två olika modulationsprinciper av det fysiska lagret, beroende på vilken enhet som sänder. Master-enheten överför data genom spänningsmodulering, där spänningsnivån på busslingan höjs och sänks med 12 V från viloläget 40 V för att representera ettor och nollor. Slav-enheten kommunicerar istället genom strömmodulering, vilket innebär att den höjer och sänker sitt strömuttag från bussen med 1,5 mA för att signalera binära värden.

Kommunikationshastigheten anges i enheten baud, där 1 baud motsvarar en symbol per sekund. I M-Bus-systemet motsvarar detta 1 bit per sekund eftersom binär signalering används. Standarden definierar flera överföringshastigheter från 300 baud upp till 38 400 baud [11]. I praktiska installationer är 2 400 baud den vanligaste överföringshastigheten, medan högre hastigheter kan användas, men ställer då högre krav på hårdvaran hos mätaren [7].

M-Bus-protokollet har utgjort en central del av detta projekt och kommer därför att beskrivas mer ingående i följande avsnitt.

3.4.1 Telegramstruktur

På länklagernivå definierar EN 13757-2 tre olika ramformat för trådbunden M-Bus [11]. Det enklaste formatet består av en enda byte. Ett exempel är kvittenssignalen `ACK` (acknowledgement), som representeras av byten $\texttt{E5}_{16}$. Detta telegram sänds från slav till master och innehåller varken adressfält eller kontrollsumma, eftersom dess funktion enbart är att indikera lyckad mottagning.

Det andra formatet är ett kort telegram med fast längd, även kallat *short frame*. Strukturen kan beskrivas som `10 C A CS 16`, där $\texttt{10}_{16}$ är startbyte, `C` är kontrollfält (Control field), `A` är adressfält (Address field), `CS` är kontrollsumma (Checksum) och $\texttt{16}_{16}$ är stoppbyte [11]. Kontrollfältet anger telegramtyp samt kommunikationsriktning, exempelvis om telegrammet är en begäran från master eller ett svar från slav. Adressfältet identifierar den avsedda enheten och möjliggör både primär och sekundär adressering.

Kontrollsumman används för att upptäcka fel på länklagernivå [11]. Den beräknas som den aritmetiska summan av värdena i kontrollfältet och adressfältet, reducerad till en byte (det vill säga modulo 256). Mottagaren utför motsvarande beräkning och jämför resultatet med den mottagna kontrollsumman. Om värdena inte överensstämmer, betraktas telegrammet som felaktigt och ignoreras.

Det tredje formatet är ett långt telegram med variabel längd, benämnt *long frame* [11]. Den generella strukturen kan skrivas som

```
68 L1 L2 68 C A CI ... CS 16
```

Notera att startbyten frånskiljer sig från *short frame* format, där den här är $\texttt{68}_{16}$. Därefter följer två längdfält, `L1` och `L2`, som anger antalet byte från kontrollfältet (`C`) till och med den sista databyten före kontrollsumman (`CS`). Enligt standarden

ska dessa längdfält ha identiskt värde (`L1 = L2`), vilket ger en enkel strukturell felkontroll på länklagernivå [11]. Efter längdfälten upprepas startbyten 68_{16}.

Längdfälten `L1` och `L2` är en byte vardera, vilket innebär att den maximala längden för datadelen i ett *long frame*-telegram är 255 byte [11]. Datadelen omfattar samtliga byte från kontrollfältet (`C`) till och med den sista byten före kontrollsumman (`CS`). Detta medför att den maximala mängden applikationsdata är något mindre än 255 byte, eftersom även fälten `C`, `A` och `CI` inkluderas i längdangivelsen.

Den totala telegramstorleken begränsas således av länklagrets längdfält. Om applikationsdatan överstiger denna gräns kan den inte överföras i ett enda telegram. I sådana fall används istället blocköverföring, där datan delas upp i flera efterföljande `RSP_UD`-telegram, vilket kallas multitelegram [11].

Begränsningen i telegramstorlek är en direkt konsekvens av att längdfältet är 8 bitar. Detta utgör en av de centrala designbegränsningarna i trådbunden M-Bus och påverkar hur stora datamängder som kan överföras i en enskild läsoperation.

Efter det andra startbytet följer kontrollfält (`C`), adressfält (`A`) och kontrollinformationsfält (`CI`). Kontrollfältet anger funktion och riktning på länklagernivå, exempelvis `SND_UD` eller `RSP_UD`. CI-fältet specificerar vilken typ av applikationsdata som följer enligt EN 13757-3 [12]. Telegrammet avslutas med en kontrollsumma, beräknad som den aritmetiska summan (modulo 256) av samtliga byte från `C` till sista databyten, samt stoppbyten `16` [11]. Figuren 3.1 nedan visar ett exempel på ett M-Bus-telegram i *long frame*-format.

```
68 95 95 68 08 01 72 01 00 00 00 29 41 01 02 00
00 00 20 0C 78 75 16 81 16 03 74 11 00 00 01 FD
71 B0 04 6D 0B 2B 4A 3A 04 06 E8 03 00 00 04 08
E8 03 00 00 04 FF A1 15 00 00 00 00 04 FF A2 00
05 00 00 00 04 FF A3 15 00 00 00 00 07 FF A6 00
00 00 00 00 00 00 00 00 07 FF A7 00 00 00 00 00
00 00 00 00 07 FF A8 00 00 00 00 00 00 00 00 00
07 FF A9 00 00 00 00 00 00 00 00 00 0D FD 8E 00
07 30 2E 30 33 2E 31 42 0D FF AA 00 0B 30 30 31
2D 33 31 31 20 34 32 42 1F 3D 16
```

Figur 3.1: Exempel på M-Bus-telegram i *long frame*-format där 68_{16}, `L1`, `L2`, 68_{16} markerats och även checksumman $3D_{16}$ följt av stoppbyte 16_{16}.

Detta exempel utgör ett svar från mätare till master, vilket indikeras av kontrollfältets värde som motsvarar funktionen `RSP_UD`. Ett sådant telegram, T, kan beskrivas som

$$T = (CI, H, \{R_n\}) \tag{3.1}$$

där `CI` är *Control Information*-fältet, `H` är ett datahuvud (*Header*). I *headern* definie-

ras information som sekundäradress, tillverkarkod. $\texttt{R}_n$ är en sekvens av dataposter [12]. Varje datapost kan skrivas som

$$R_n = \{\text{DIF}, \textit{DIFE}, \text{VIF}, \textit{VIFE}, \text{Data}\}, \tag{3.2}$$

där DIF anger datalängd och kodning, VIF anger fysisk storhet och enhet samt `Data` är själva mätvärdet [12]. Eventuella DIFE- och VIFE-fält är utökningar. DIF/DIFE bildar ett *Data Information Block* (DIB) och VIF/VIFE bildar ett *Value Information Block* (VIB). Kombinationen gör att telegrammet blir självbeskrivande, vilket innebär att mottagaren kan tolka mätvärden utan förkonfigurerad datastruktur.

Som ett konkret exempel kan dataposten som representerar energivärdet 1000 kWh identifieras i figur 3.1 ovan, detta enligt bytesekvensen

```
04 06 E8 03 00 00
```

Den första byten ($\texttt{04}_{16}$) utgör datapostens DIF och värdet `04` anger att dataposten består av ett 32-bitars heltal (INT32) [12]. Ingen DIFE följer, eftersom utökningsbiten i DIF inte är satt.

Den andra byten ($\texttt{06}_{16}$) utgör VIF och ($\texttt{06}_{16}$) motsvarar energi uttryckt i kilowattimmar (kWh) enligt M-Bus-standarden [12]. Inte heller här följer någon VIFE, eftersom ingen utökningsbit är satt.

Dessa två bytes tillsammans innebär alltså att datan är ett 32-bitars heltal uttryckt i kilowattimmar.

De fyra sista bytena (`E8 03 00 00`) utgör själva datan. M-Bus använder little endian-representation på datavärden och därför tolkas värdet som

$$000003\text{E}8_{16} = 1000_{10}$$

vilket ger energivärdet 1000 kWh.

Detta exempel illustrerar hur kombinationen av DIB och VIB gör telegrammet självbeskrivande, eftersom både datatyp och fysisk enhet kan utläsas direkt ur telegrammet med hjälp av förbestämda tabeller i EN 13757-3 [12] utan extern konfiguration.

3.4.2 Initiering av M-Bus-kommunikation

Som framgått i föregående avsnitt överförs mätdata från slav till master i form av ett `RSP_UD`-telegram i *long frame*-format. Eftersom trådbunden M-Bus är strikt masterinitierad sker all kommunikation på initiativ av mastern [11]. Slavar sänder endast data som svar på explicita kommandon.

När bussen är stabil initierar mastern normalt länklagret genom att sända `SND_NKE` till broadcast-adressen $\texttt{FF}_{16}$. Detta återställer slavarnas länkstatus utan att påverka applikationsdata [11]. Ett exempel är:

```
10 40 FF 3F 16
```

En slav som mottar kommandot korrekt svarar med kvittensbyten $\texttt{E5}_{16}$ (`ACK`).

För att adressera en specifik mätare kan mastern använda antingen primär eller sekundär adressering. Vid primär adressering används slavens primäradress direkt i telegrammets adressfält [11].

Vid sekundär adressering används istället kommandot `SND_UD` med funktion *Slave Select*, där mätarens sekundäradress (identifikationsnummer, tillverkarkod, version och medium) anges i nyttolasten [11]. Endast den slav vars sekundäradress matchar selektionskriteriet svarar då med `ACK` ($\texttt{E5}_{16}$). Detta möjliggör unik identifiering av en mätare även när primäradresser inte är kända.

Efter lyckad adressering begär mastern användardata, vanligtvis med kommandot `REQ_UD2` [11], exempelvis:

```
10 5B 01 5C 16
```

Slaven svarar därefter med ett `RSP_UD`-telegram (exempelvis figur 3.1).

Initierings- och avläsningssekvensen kan därmed sammanfattas enligt följande:

1. Spänningssättning och stabilisering av bussen.
2. Primär eller sekundär adressering av master till slav via `SND_NKE` respektive *Slave Select*.
3. Kvittens från slav (`ACK`).
4. Dataförfrågan från master `REQ_UD2`.
5. Svar från slav med `RSP_UD` i *long frame*-format.

Denna sekvens säkerställer korrekt adressering och synkronisering innan applikationsdata överförs [11].

3.4.3 Singel- och multitelegram

Vid normal avläsning med `REQ_UD2` kan en slav svara med antingen ett enskilt telegram (singeltelegram) eller flera telegram i följd (multitelegram) [11]. Mekanismen regleras på länklagernivå genom användning av FCB/FCV-bitarna i kontrollfältet enligt EN 13757-2 [11].

I ett singeltelegram ryms all applikationsdata i ett enda telegram. Mastern skickar då `REQ_UD2` och slaven svarar direkt med ett `long frame`-telegram. Ingen ytterligare blockhantering krävs.

Om datamängden är större än vad som får plats i ett telegram används istället

multitelegram. I detta fall delas svaret upp i flera efterföljande `RSP_UD`-telegram. För att säkerställa korrekt ordning och undvika dubbletter används `Frame Count Bit` (FCB) tillsammans med `Frame Count Valid` (FCV) i kontrollfältet [11]. FCB fungerar som en sekvensbit som växlas mellan 0 och 1, där $\texttt{1F}_{16}$ indikerar att mer data finns att hämta från mätaren och $\texttt{0F}_{16}$ visar att det finns fler telegram att hämta.

Vid användning av `REQ_UD2` växlar kontrollfältets värde därav mellan $\texttt{5B}_{16}$ och $\texttt{7B}_{16}$. Skillnaden mellan dessa två värden är FCB-biten. När mastern begär nästa block i en multitelegramsekvens skiftar FCB-biten, vilket signalerar till slaven att nästa telegram ska överföras [11]. Om FCB inte skiftas korrekt kommer slaven att repetera föregående block istället för att sända nästa.

Multitelegram används alltså när:

- Applikationsdatan överstiger maximal längd för ett enskilt telegram.
- Flera datalager behöver överföras sekventiellt.
- Eller när blocköverföring krävs för tillförlitlig dataöverföring.

Skillnaden mellan singel- och multitelegram ligger därför inte i telegramformatet i sig, utan i hur länklagret hanterar sekvensering via FCB/FCV-mekanismen. Denna blockhantering är central för robust överföring i flerpunktssystem och möjliggör överföring av stora datamängder utan att bryta mot telegrammets längdbegränsningar [11].

3.4.4 M-Bus över TCP

I vissa system transporteras M-Bus-kommunikation inte direkt över den fysiska M-Bus-bussen, utan kapslas istället in i ett IP-baserat nätverk. Detta möjliggör fjärrkommunikation över lokala nätverk eller internet, där TCP används som transportprotokoll. I ett sådant upplägg fungerar TCP-anslutningen som en tillförlitlig transportkanal där M-Bus-telegram överförs som en kontinuerlig dataström mellan två system.

När M-Bus överförs över TCP används vanligtvis en klient-server-modell. En klient, exempelvis ett överordnat system eller en gateway, etablerar en TCP-anslutning till en server som representerar en M-Bus-enhet eller en M-Bus-gateway. När anslutningen är etablerad kan M-Bus-telegram skickas över anslutningen utan att varje enskilt telegram behöver skapa en ny nätverksanslutning. TCP säkerställer samtidigt att telegrammen levereras i korrekt ordning och utan förlust, vilket är viktigt när telegrammen innehåller strukturerad mätdata.

En viktig skillnad jämfört med traditionell seriell kommunikation är att TCP arbetar med en kontinuerlig byte-ström snarare än diskreta paket. Det innebär att mottagaren måste kunna identifiera var ett M-Bus-telegram börjar och slutar i dataströmmen. Detta görs normalt genom att tolka telegrammets struktur, exempelvis

startbyte och längdfält, för att avgöra när ett komplett telegram har mottagits.

Genom att använda TCP som transportlager kan M-Bus-kommunikation integreras i befintliga IP-baserade nätverk och möjliggöra fjärravläsning av mätare utan direkt fysisk anslutning till M-Bus-bussen. Detta är särskilt användbart i system där mätdata behöver samlas in från geografiskt spridda installationer eller där gateways används för att koppla samman lokala M-Bus-nät med överordnade system.

3.4.5 Wireless M-Bus

Wireless M-Bus (wM-Bus) är en trådlös vidareutveckling av det trådbundna M-Bus-protokollet, avsett för fjärravläsning av mätare såsom vatten-, värme- och elmätare. Protokollet är standardiserat enligt EN 13757 [12] och möjliggör energisnål, långdistanskommunikation via radio, vilket gör det särskilt lämpligt i applikationer där kabeldragning är opraktisk eller kostsam.

En central likhet mellan Wireless M-Bus och traditionell M-Bus är att de delar samma grundläggande datamodell och telegramstruktur. Detta innebär att informationen som överförs, exempelvis mätvärden och enhetsidentifiering, är uppbyggd på ett likartat sätt oberoende av överföringsmedium. Därmed kan system som hanterar M-Bus-data ofta anpassas för att även stödja Wireless M-Bus med relativt begränsade förändringar.

Den huvudsakliga skillnaden mellan protokollen ligger i kommunikationsmediet och hur dataöverföringen sker. Till skillnad från M-Bus, där data endast överförs om mastern förfrågar en mätare, kan Wireless M-Bus däremot arbeta både i sändarinitierade lägen och i mer strukturerade kommunikationslägen beroende på driftläge [7]. Detta innebär att en mätare kan skicka data periodiskt utan att först bli förfrågad.

3.5 Modbus

Modbus är ett registerbaserat kommunikationsprotokoll där data adresseras och hämtas genom åtkomst till specifika register i anslutna enheter, exempelvis mätare eller styrenheter. Protokollet används för datautbyte mellan industriella styrsystem såsom programmerbara logiska styrenheter (PLC), sensorer, ställdon och överordnade system och möjliggör överföring av mätvärden, statusinformation och styrkommandon mellan enheter.

Kommunikationen baseras vanligtvis på en master-slave- eller klient-server-arkitektur, där en central nod initierar förfrågningar och adresserade enheter svarar med begärd information eller utför specificerade åtgärder. Modbus förekommer i flera varianter, där Modbus RTU och Modbus ASCII används över seriella gränssnitt såsom RS-232 eller RS-485, medan Modbus TCP/IP möjliggör kommunikation över ethernet.

PiiGAB stödjer M-Bus i upp till 38 400 baud, men som tidigare nämnt är de flesta installationer av M-Bus inställda på 2400 baud. Detta betyder att i verkliga in-

stallationer går det fortare att nå data genom Modbus jämfört med M-Bus. Detta tillsammans med sin enkla struktur, låga resurskrav och breda hårdvarukompatibilitet, gör att Modbus ofta används inom industriell automation och IoT-baserade system för datainsamling och styrning [13].

3.6 Gateway

Inom teknikområdet Internet of Things (IoT) utgör en gateway en central komponent som agerar brygga mellan fysiska enheter, molnbaserade tjänster och användarutrustning. Gatewayen möjliggör således kommunikation mellan fältmiljön och molnet och kan dessutom möjliggöra realtidsstyrning av enheter i fält.

PiiGABs centrala hårdvaruutbud utgörs främst av företagets olika gateway-modeller, vilka möjliggör transparent fjärravläsning via kommunikationsprotokollen M-Bus och Modbus. Lösningarna möjliggör samtidig avläsning av flera M-Bus- och Modbusmätare från olika åtkomstpunkter, där fjärravläsning kan genomföras via lokala nätverk, stadsnät eller internet [14].

De mest populära gateway-modellerna är 900-serien, som kommer i flera olika installationer beroende på kundens behov, se figur 3.2 nedan.

Figur 3.2: Produktbild på PiiGAB 900S, [14] .

3.6.1 PiiGAB 900

PiiGABs 900-serie omfattar modellerna 900T och 900S, vilka båda fungerar som M-Bus-gateways för datainsamling och kommunikation mellan mätare och överordnade nätverk. Modellen 900S är utrustad med flera hårdvarugränssnitt, däribland seriella portar (RS-232 och RS-485), digitala ingångar samt reläutgångar, vilket möjliggör direkt anslutning till extern utrustning och industriella styrsystem [14]. Modellen 900T saknar portarna RS-232 och RS-485, samt de seriella gränssnitten. Därmed erbjuder den en mer begränsad hårdvarukonfiguration, avsedd för tillämpningar där enbart grundläggande nätverksbaserad M-Bus-kommunikation och fjärravläsning krävs.

Båda modellerna är modulärt uppbyggda, vilket möjliggör utökning av funktionalitet genom tillägg av kompletterande hårdvarumoduler [14]. Exempelvis kan nätverkskort och antenn installeras för att möjliggöra kommunikation via Wireless M-Bus. Modellen 900T kan dessutom utökas med stöd för ytterligare kommunikationsprotokoll, såsom LTE och LoRa [7].

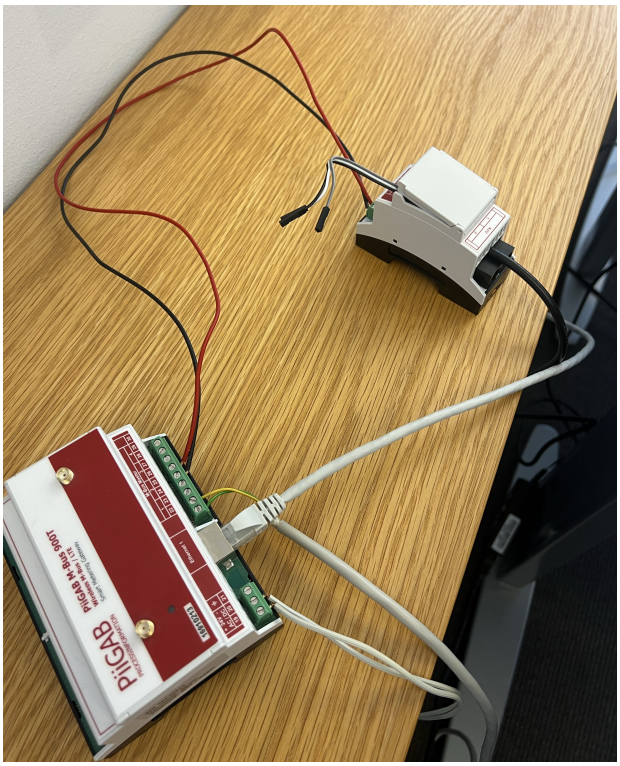
Mjukvaran i dessa gateways är uppbyggd av separata paket, vilket möjliggör flexibel uppdatering och underhåll av systemets funktionalitet. Uppdateringar kan utföras via det inbyggda webbgränssnittet eller via SSH (*Secure Shell*), vilket är ett krypterat kommunikationsprotokoll för säker fjärråtkomst till system. Genom SSH ges användaren tillgång till en terminalmiljö i gatewayen.

Via terminalmiljön kan pakethanteraren `opkg` användas för att installera, uppdatera och ta bort mjukvarupaket. Detta möjliggör versionshantering på komponentnivå, där enskilda delar av systemet kan uppgraderas eller nedgraderas oberoende av varandra.

3.7 Modifierad HanP1-konverterare

På företaget fanns en modifierad HanP1-konverterare som kunde användas för att möjliggöra seriell M-Bus-kommunikation mellan en dator och en gateway. Lösningen baserades på hårdvara från en HanP1-konverterare och därav benämns komponenten genom projektet som modifierad HanP1-konverterare. HanP1 är ett protokoll som inte hanteras i examensarbetet och kommer därför inte hanteras i rapporten.

Den modifierade enheten användes som en brygga mellan datorns USB-port och gatewayens M-Bus-anslutning. På så sätt kunde en dator skicka och ta emot M-Bus-telegram seriellt via gatewayen, vilket var nödvändigt för att kunna använda den mätarsimulator som förklaras senare i rapporten. Figur 3.3 nedan visar HanP1-konverteraren kopplad till en gateway.



Figur 3.3: Modifierad HanP1-konverterare kopplad till gateway.

3.7.1 Mätarsimulator

Ett verktyg utvecklat av PiiGAB som via en värddator kan simulera flera mätare och möjliggöra kommunikation med dessa genom fördefinierade portar och baudhastigheter. Simulatorn är utformad för att ta emot M-Bus förfrågningar och hantera kommunikation både via seriell M-Bus och över IP.

Lösningen bygger på en fördefinierad lista med telegram för respektive mätare. Eftersom telegrammen innehåller både primär- och sekundäradresser kan programmet tolka dessa värden och därigenom skapa möjligheten att etablera och hantera kommunikationen med mätarna.

4

Genomförande

Detta kapitel beskriver genomförandet av projektet och de praktiska steg som ledde fram till den färdiga testmiljön. Kapitlet inleds med den kunskapsinhämtning som genomfördes genom interna utbildningar och fortsätter sedan med en beskrivning av den befintliga testmiljön samt hur en egen laborationsmiljö etablerades.

Därefter redovisas hur testmiljön successivt utvecklades genom nya tester, stödverktyg och funktioner för konfigurering och exekvering. Kapitlet avslutas med hur testmiljön dokumenterades och verifierades för att säkerställa att de implementerade testerna fungerade enligt förväntan.

4.1 Intern utbildning

Under de två första veckorna av examensarbetet genomfördes en introduktionsperiod med utbildningstillfällen organiserade av företaget. Utbildningarna hölls av anställda från olika avdelningar och syftade till att ge en grundläggande förståelse för PiiGAB:s verksamhet, produkter och tekniska system. Varje utbildningstillfälle varade i ungefär en till en och en halv timme.

Introduktionsperioden inleddes med en övergripande genomgång av PiiGAB som företag. Där presenterades företagets bakgrund, kundgrupper och organisatoriska struktur. Syftet var att skapa en förståelse för företagets roll inom branschen och för det sammanhang där testsystemet skulle användas.

Därefter följde tekniska genomgångar av företagets produkter, med särskilt fokus på gateway-modellen PiiGAB 900. Genomgångarna omfattade bland annat hårdvarans uppbyggnad, den modulära arkitekturen och webbgränssnittet som används för konfiguration och administration.

En efterföljande utbildning behandlade mjukvaruarkitekturen i PiiGAB:s system. Under denna genomgång presenterades vilka programmeringsspråk som används i olika delar av systemen, samt hur källkoden är organiserad i företagets GitHub-repositorier. Detta gav en överblick över den befintliga kodbasen och företagets utvecklingsprocess.

Slutligen genomfördes utbildningar om de kommunikationsprotokoll som är relevan-

ta för produkterna. Särskilt fokus lades på M-Bus, Wireless M-Bus och Modbus. Dessa genomgångar behandlade protokollens grundläggande uppbyggnad, användningsområden och betydelse i företagets gateways. Utbildningarna gav därmed den tekniska grund som behövdes för att kunna påbörja arbetet med testmiljön.

4.2 Initial testmiljö

Vid projektets start fanns en påbörjad testmiljö tillgänglig hos företaget. Utvecklingen av denna hade dock av olika skäl pausats och den senaste uppdateringen var från oktober 2024. Den befintliga testmiljön användes som utgångspunkt i arbetet. Utöver testmiljön fanns en implementation av en mätarsimulator som kunde användas för att simulera mätare genom att svara på frågor med förutbestämda telegram.

Den initiala implementationen bestod av ett test som etablerade en UDP-anslutning mot en mätarsimulator. Testet inleddes med att kommandot `Slave Select` skickades till simulatorn, varefter ett kvittenssvar (`ACK`) verifierades. Därefter kontrollerades att svaret kom från en mätare med korrekt förbestämd sekundäradress.

När adresseringen verifierats skickades kommandot `REQ_UD2` för att begära ett telegram från mätaren. Testet verifierade sedan att det mottagna svaret var ett giltigt *long frame*-telegram samt att sekundäradressen i telegrammet överensstämde med den adresserade mätaren.

Den befintliga testmiljön fungerade som en grund för det fortsatta arbetet och bidrog till att snabbt kunna påbörja vidare utveckling och utökning av testfunktionaliteten.

4.3 Uppsättning av laborationsmiljö

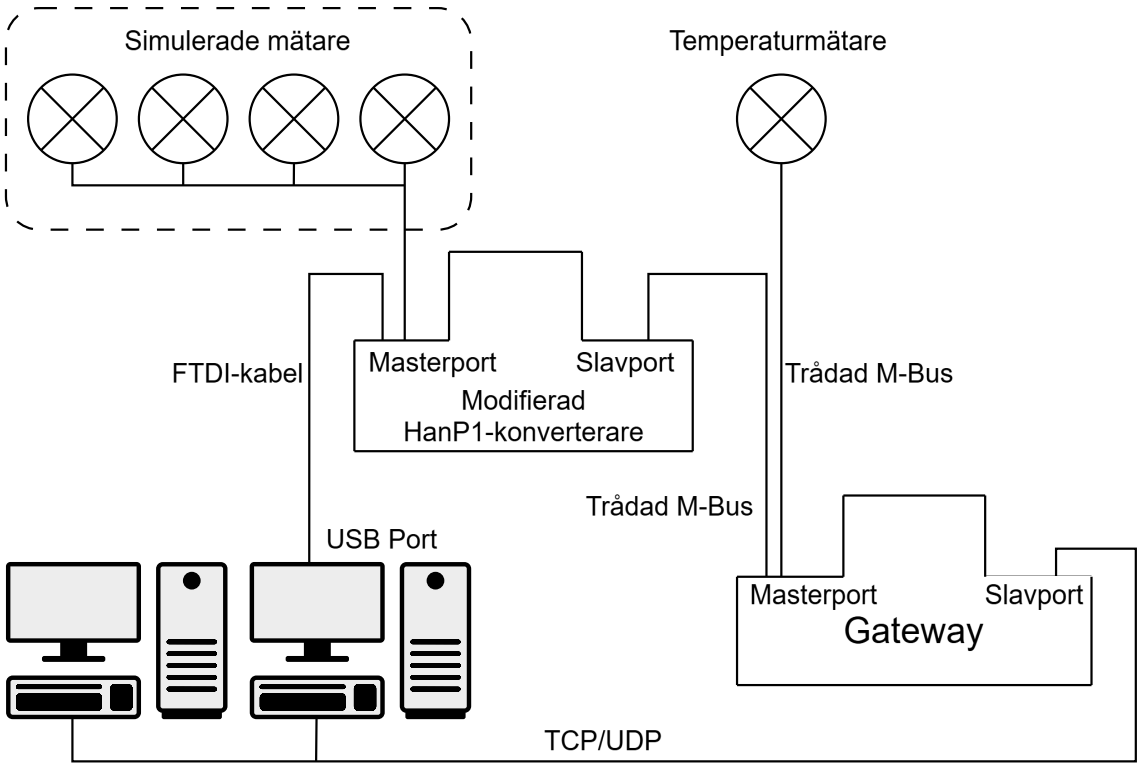
Efter de interna utbildningarna påbörjades arbetet med att etablera en laborationsmiljö som skulle fungera som grund för det framtida testsystemet. Arbetet genomfördes iterativt där den initiala uppsättningen successivt förbättrades för att bättre stödja utveckling och testning.

I den första iterationen användes en befintlig testrigg bestående av en PiiGAB 900S med flera anslutna mätare. Denna uppsättning möjliggjorde praktisk interaktion med systemet och underlättade därmed inlärningen av den aktuella hårdvaran. För att kunna använda en mätarsimulator med transparent m-bus krävdes dock en fysisk anslutning mellan datorn som kör simulatorn och gatewayen. Detta var inte möjligt i den befintliga uppsättningen eftersom testriggen var placerad på en annan våning än arbetsplatsen.

I den andra iterationen etablerades därför en egen laborationsmiljö på kontoret. En gateway av modellen PiiGAB 900T LTE tilldelades för detta ändamål. Valet av modell baserades främst på tillgänglighet, då den specifika varianten av 900-serien inte hade någon avgörande betydelse i den inledande fasen av arbetet.

Gatewayen installerades på kontoret med en temperaturmätare inkopplad i en av gatewayens masterport. För att gatewayen skulle kunna användas fullt ut installerades även nödvändiga licenser. Därefter lades mätaren till i mätarlistan genom gatewayens webbgränssnitt.

När grundläggande funktionalitet hade verifierats, etablerades även en anslutning mellan datorn och gatewayen för att möjliggöra kommunikation med simulerade mätare. Detta genomfördes genom att använda den modifierade HanP1-konverteraren (som förklarats under avsnitt 3.7 Modifierad HanP1-konverterare). Med denna uppsättning kunde en mätarsimulator som kördes på datorn kommunicera med gatewayen via seriell M-Bus. Den andra iterationen av laborationsmiljön illustreras i figur 4.1 nedan.



Figur 4.1: Laborationsmiljön som användes för testningen.

Denna iteration bestod av en dator som körde mätarsimulatorn via HanP1-konverteraren där den var kopplad till gatewayens masterport och en av datorns USB-portar. Temperaturmätaren var ansluten till en av gatewayens masterport. Bilden i sig är något missvisande, då HanP1 egentligen inte har någon direkt master- eller slavport utan snarare är en sammansatt kabel (se figur 3.3).

Tester kunde även exekveras från en extern dator som skapar en anslutning via

antingen UDP eller TCP till gatewayens slavport. Detta möjliggjorde förfrågningar till antingen en simulerad mätare eller en verkligt ansluten mätare.

Detta medför att den externa datorn agerar master mot gatewayen som i sin tur agerar master mot de simulerade mätarna och den verkliga temperatursensorn.

Denna iteration utgjorde även en grund som fortsatte att användas under projektets gång. Vidareutvecklingen av laborationsmiljön framöver handlade snarare om vilka sorters eller antal mätare som behövdes för att genomföra specifika tester.

4.3.1 Felsökning av Laborationsmiljön

Under implementationsfasen av det första testet observerades att testresultaten från temperaturmätaren i laborationsmiljön varierade mellan godkända och misslyckade körningar. Detta beteende var oväntat eftersom samma test kunde godkännas vid en körning, men en ny körning kort därefter kunde resultera i ett misslyckande.

En felsökningsprocess inleddes för att identifiera orsaken till problemet. Genom att visa telegrammen i terminalen på en dator kunde det konstateras tidigt att mätaren vid vissa tillfällen skickade ofullständiga telegram, vilket i sin tur ledde till att testen misslyckades.

Det första steget var att ansluta temperaturmätaren till en annan port på gatewayen för att utesluta eventuella portrelaterade problem. Detta resulterade dock inte i någon förändring i beteendet. Därefter undersöktes temperaturmätaren mer ingående av en anställd på företaget genom att läsa av telegrammen direkt från mätaren. De avlästa telegrammen visade sig vara fullständiga, vilket indikerade att problemet inte låg i själva mätaren.

För att vidare undersöka om problemet kunde bero på testsystemet användes ett program utvecklat av PiiGAB vid namn *M-Bus Wizard* för att läsa av telegram direkt från gatewayen. Resultaten visade att även gatewayen i vissa fall mottog ofullständiga telegram. Detta isolerade problemet till att det uppstod någonstans i kommunikationen mellan mätaren och gatewayen.

Som nästa steg uppdaterades samtliga mjukvarupaket i gatewayen i syfte att utesluta programvarurelaterade fel. Efter uppdateringen genomfördes nya tester, men problemet kvarstod. Därefter ersattes gatewayen helt och hållet med en annan enhet av modellen 900T, vilken konfigurerades på motsvarande sätt som den tidigare enheten. Trots detta observerades samma beteende när temperaturmätaren lästes av med hjälp av *M-Bus Wizard*, där telegrammen fortfarande ibland var ofullständiga.

I detta skede föreslog en anställd att problemet potentiellt kunde bero på ett jordfel i M-Bus-slingan, orsakat av kabelkopplingen mellan datorn, HanP1-konverteraren och gatewayen. Det kopplades då in en isolator mellan datorns USB-port och kabeln till den modifierade HanP1-konverteraren. Problemet kvarstod dock, men i samma sittning kollade en anställd på hur gatewayens portar konfigurerats och märkte då att två masterportar var konfigurerade med transparent M-Bus, vilket resulterat i data-

krockar i gatewayen varje gång ett telegram skulle vidarebefordras till applikationen. Portarna omkonfigurerades och laborationsmiljön fungerade då som förväntat.

4.4 Implementering av första testet

Det första testet bestod av att läsa av data från en mätare med förutbestämt innehåll. Eftersom mätarens data var känd i förväg kunde den mottagna datan enkelt verifieras genom att jämföra den med den förväntade datan.

Tidigt under implementationsprocessen identifierades ett behov av ett mer effektivt sätt att generera M-Bus-telegram med specificerade egenskaper. Att manuellt konstruera telegram genom att redigera hexadecimala värden visade sig vara både tidskrävande och komplicerat, särskilt vid behov av variation i telegrammens struktur och innehåll.

Som en följd av detta implementerades ett verktyg för att generera M-Bus-telegram baserat på en mer läsbar och strukturerad representation av telegrammens innehåll. Denna funktionalitet utvecklades initialt för att stödja implementeringen av det första testet, men vidareutvecklades senare till ett separat verktyg med namnet *Json2Telegram*.

4.4.1 Json2Telegram

Grundidén var att skapa ett program som kan framställa ett korrekt M-Bus-telegram utifrån standarden för protokollet, baserat på en JSON-fil där egenskaper som sekundäradress, tillverkarkod och mätdata enkelt kunde konfigureras. Inledningsvis undersöktes vilken information som en användare rimligen ska ange manuellt och vilken information som istället bör genereras automatiskt av programmet.

Utifrån denna analys skapades en mall i JSONC-format där centrala egenskaper kunde definieras på ett tydligt och lättmodifierat sätt. Istället för vanliga JSON valdes JSONC-formatet, eftersom formatet tillåter rad-kommentarer genom `//` notation. Detta leder till att dokumentation kan föras direkt på samma rad som de olika fälten. Programmet läser in filen, tolkar dess innehåll och översätter sedan värdena till motsvarande representation i ett M-Bus-telegram.

Implementationen delades upp i flera komponenter för att göra koden mer överskådlig och lätt att vidareutveckla. Huvudprogrammet ansvarar för att läsa in JSON-filen, tolka strukturen och initiera skapandet av telegrammet. Därefter byggs telegrammet stegvis genom att olika delar av datan konverteras till byte-sekvenser enligt M-Bus-standardens specifikationer. Programmet genererar slutligen ett komplett *long frame*-telegram, figur 3.1 är faktiskt ett telegram som genererats av verktyget.

För att förenkla konverteringen mellan användarens indata och byte-värdena i telegrammet implementerades en separat modul med tabeller och hjälpfunktioner. Dessa tabeller innehåller exempelvis kopplingar mellan olika enheter och deras mot-

svarande VIF-koder samt mellan datatyper och motsvarande DIF-koder. På så sätt kan användaren ange exempelvis en fysisk enhet i JSONC-filen, medan programmet automatiskt väljer rätt kod enligt standarden.

Själva telegrammet byggs upp genom att först skapa kontrollfält, adressfält, kontrollinformationsfält och dataposter. När det är färdigt beräknas telegrammets längdfält och kontrollsumma automatiskt innan start- och stoppbyte läggs till. Användaren kan alltså mata in bland annat en egen sekundäradress och exempelvis värden som 1000 kWh och programmet kan utifrån hjälpfunktionerna skapa ett korrekt M-Bus-telegram.

Programmet utformades även för att kunna generera flera telegram i följd från en och samma JSONC-fil, vilket möjliggör simulering av exempelvis multitelegram eller olika mätvärden över tid. Detta gör verktyget användbart både för enkel testning av enskilda telegram och för mer omfattande testscenarier.

4.4.2 Data Test

Med utgångspunkt i att testmiljön i grunden ska användas efter att en ny uppdatering har installerats på en given gateway, behöver dess funktionalitet verifieras. Detta innefattar hur gatewayen kommunicerar med de olika mätarna samt med sina slavportar, det vill säga hur kommunikationen över samtliga relevanta protokoll genomförs och att denna sker korrekt.

Ett effektivt tillvägagångssätt för att testa detta var att använda förgenererade telegram skapade med Json2Telegram och därefter verifiera att gatewayen, oavsett protokoll, skickar dessa förgenerarade telegram från sina slavportar som de referenstelegram som definierats i förväg.

Den första varianten av detta test bestod av använda sig av ett telegram som körs i mätarsimulatorn. Applikationen förfrågar gatewayen via UDP eller TCP (beroende på vad som konfigurerats) och förfrågningen vidarebefordras till den simulerade mätaren. Mätaren skickar telegrammet via seriell M-Bus till gatewayen och telegrammet skickas vidare via M-Bus över IP genom UDP eller TCP. Testet kan beskrivas enligt tabell 5.1 under avsnitt 5.2.1 Data Test.

4.5 Utökad testning

Efter att *Data Test* implementerades och verifierades skapades fler tester. Implementeringsprocessen för dessa tester kommer att beskrivas i detta avsnitt.

4.5.1 Read Test

Under implementeringen av *Data Test* insågs det att `access number` ökas med `1` varje gång data hämtas från verkliga mätare. Detta resulterade i att *Data Test* behövde modifieras så att `access number` nollställdes vid hämtning av telegram.

Utöver detta var grundidén med *Data Test* att ett telegram behövde vara känt innan exekvering, vilket inte var något som kunde garanteras för verkliga mätare. Det fanns därför behov av att skapa ett liknande test, men som fungerar mot verkliga mätare med samma syfte, alltså verifiera att förfrågan av telegram, mottagning och avläsning skett korrekt i en gateway.

Read Test implementerades på så vis att det skickar två tätt efterföljande `REQ_UD2` förfrågningar. Testet nollställer access-fält och checksumma och genomför därefter en bytevis jämförelse av telegrammen. Testet kan förklaras enligt tabell 5.2 under avsnitt 5.2.2 Read Test.

4.5.2 Package Install/Uninstall

På företagets önskan skapades ett test för installation och avinstallation av gatewayens paket, vilket skulle användas i kombination med avläsning av data. Detta i syfte att säkerställa att gatewayen läser av telegram på samma sätt innan och efter en uppdatering på ett eller alla installerade paket.

Detta implementerades genom att använda SSH-protokollet, där testmiljön ansluter till gatewayen och genom `opkg` antingen installerar eller avinstallerar de paket användaren valt. Funktionalitet för att hitta tillgängliga paket implementerades även för att kunna installera eller avinstallera alla paket som finns tillgängliga på gatewayen.

Syftet med testet var som tidigare nämnt att endast användas i kombination med andra tester, men en positiv bieffekt utifrån detta test är att även gatewayens pakethantering verifieras eftersom `opkg` svarar med en statusflagga som kan kontrolleras. Genom att kontrollera denna flagga går det att säkerställa att pakethanteringen skett korrekt. Installationsprocessen beskrivas enligt tabell 5.3 under avsnitt 5.2.3 Packet Install/Uninstall.

Ett problem som uppstod vid testkörning var att SSH-anslutningen inte alltid hann stängas korrekt innan nästa test i en testsekvens påbörjades. Detta kunde skapa problem när ett efterföljande test direkt skulle etablera ny kommunikation med gatewayen efter installation eller avinstallation av paket. För att hantera detta implementerades en kort fördröjning efter att SSH-anslutningen stängts. På så sätt fick gatewayen och testmiljön tid att avsluta den pågående anslutningen innan nästa test startades.

Båda testerna måste konfigureras med gatewayens IP-adress, vilket enkelt kan hittas med hjälp av PiiGAB:s verktyg *MBus-Wizard*. Även SSH-lösenordet måste vara känt, vilket kan vara ett hinder om användaren inte har rättigheter till detta. Det var dock inte något hinder i arbetets laborationsmiljö. Vid val av att installera eller avinstallera vissa paket måste även paketnamnet vara känt, vilket kan tas utifrån antingen webbgränssnittet eller direkt ur gatewayen genom SSH.

Verifiering av testerna kan delas upp i två olika kategorier, där ena är att paketen

faktiskt installerats eller avinstallerats och andra att processen genomförts korrekt, alltså att inget paket är korrupt. Den första kategorin har verifierats genom att både kolla paketlistan i webbgränssnittet samt i gatewayens terminal genom SSH-anslutning.

4.5.3 Corrupted Test

För att verifiera gatewayens felhantering implementerades testet *Corrupted Test*. Testet kontrollerade hur gatewayen reagerade när ett korrupt telegram togs emot från en mätare. Syftet var att säkerställa att felaktiga telegram identifierades och inte behandlades som giltig mätdata.

Testet genomfördes med hjälp av en simulerad mätare som konfigurerades att skicka ett korrupt telegram. Ett sådant telegram kunde exempelvis bestå av en felaktig startbyte, där 67_{16} användes i stället för den förväntade startbyten 68_{16}, eller en felaktig stoppbyte, där 15_{16} användes i stället för 16_{16}. På så sätt kunde testet efterlikna situationer där telegrammets struktur inte följde M-Bus-formatet, enligt avsnitt 3.4.1. Mätarens sekundäradress behövde däremot vara korrekt, eftersom gatewayen fortfarande måste kunna adressera mätaren och initiera kommunikationen.

För att även kontrollera att gatewayen inte påverkades av det felaktiga telegrammet genomfördes en ny avläsning direkt efteråt, denna gång mot en mätare med giltig data. Om gatewayen därefter kunde ta emot och hantera ett korrekt telegram visade det att det korrupta telegrammet inte påverkat gatewayen. Det mer detaljerade testflödet beskrivs i tabell 5.5 under avsnitt 5.2.4 Corrupted Test.

4.5.4 Timeout Test

Ett test för verifiering av gatewayens timeout-hantering implementerades även. Detta genom att skicka en `Slave select`-operation mot en icke-existerande sekundäradress. Det beteende som testet avser att säkerställa kan framstå som relativt trivialt, eftersom det förväntade utfallet är att inget svar genereras, men det bedöms ändå vara av värde att detta prövas och verifieras.

Likt `Corrupted Test` så består detta test av två efterföljande `slave select`-operationer, en riktad mot en icke-existerande sekundäradress och en riktad mot en existerande sekundäradress. Denna kombination exekveras tre gånger i följd, men kan vid behov upprepas ytterligare för att stärka resultatens tillförlitlighet. Detta test kan beskrivas enligt tabell 5.6 under avsnitt 5.2.5 Timeout Test.

4.5.5 Value Test

De flesta tidigare testerna i testsystemet byggde på att mottagna telegram kunde jämföras byte för byte med ett förväntat telegram. Detta fungerade väl för simulerade mätare med statisk testdata, men var mindre lämpligt för verkliga mätare där mätvärden kan förändras mellan två avläsningar. För att även kunna testa sådana scenarier implementerades *Value Test*. Testets syfte var att verifiera att ett valt

mätvärde från en verklig mätare ligger inom ett förväntat intervall över tid, i stället för att kräva att hela telegrammet är identiskt mellan två avläsningar.

Utgångspunkten för testet är att användaren väljer vilket mätvärde i telegrammet som ska kontrolleras, samt vilken tolerans som ska tillåtas. På exempelvis en temperaturmätare kan detta innebära att temperaturen får variera ett visst antal grader utan att testet betraktas som misslyckat. För andra typer av mätare kan toleransen behöva definieras på ett annat sätt. Ett exempel på detta är att uppmätt elförbrukning i en elmätare inte bör minska mellan två avläsningar. Därför ansågs det viktigt att toleransen inte hårdkodades i testet, utan kunde anpassas efter mätartyp och testscenario.

För att kunna välja ett specifikt mätvärde behövde testet kunna läsa ut enskilda dataposter ur ett `RSP_UD`-telegram. Eftersom mätvärdena i telegrammet kan hanteras som separata dataposter användes deras index för att identifiera vilket värde som skulle kontrolleras. Användaren kan därmed ange vilket index som motsvarar det mätvärde som är relevant för testet.

Testet utgår från att ett telegram hämtas från mätaren via gatewayen och att ett förutbestämt mätvärde därefter tas ut ur telegrammet. Det första avlästa värdet används som referensvärde för den fortsatta testkörningen. Vid efterföljande avläsningar hämtas samma mätvärde ut från nya telegram och jämförs med referensvärdet. Om avvikelsen mellan värdena överstiger den tolerans som användaren har definierat, betraktas testet som misslyckat. Testet beskrivs mer detaljerat i tabell 5.7 under avsnitt 5.2.6 Value Test.

Med detta tillvägagångssätt kan testet användas för att övervaka att en verklig mätare returnerar rimliga värden över tid. Detta gör testet särskilt användbart vid längre testkörningar mot verkliga mätare, där syftet är att upptäcka avvikande värden eller oregelbundet beteende. Testet är däremot mindre givande för simulerade mätare med helt statiska värden, eftersom dessa normalt returnerar samma mätdata vid varje avläsning.

4.5.6 Load Test

I den initiala testmiljön fanns `Load Test` redan implementerat och användes i början av arbetet för att verifiera att laborationsmiljön var korrekt uppsatt. Testet gav en första bekräftelse på att kommunikationen mellan testsystemet, gatewayen och mätaren fungerade. Eftersom testet redan existerade utvecklades det inte från grunden, utan refaktorerades för att kunna användas i den nya strukturen för konfiguration och exekvering, vilka beskrivs i senare avsnitt.

Den ursprungliga versionen var funktionell, men variabler och testvärden var hårdkodade direkt i Robot Framework. Vid refaktoreringen flyttades dessa parametrar så att de i stället kunde hanteras genom det nya konfigurationsflödet. Därmed kunde `Load Test` användas på samma sätt som de övriga testerna och ingå i valda testsekvenser.

Syftet med `Load Test` var att verifiera att rätt mätare svarade vid initiering av M-Bus-kommunikation. Testet kontrollerade att den sekundäradress som fanns i mätarens svar överensstämde med den adress som användes vid initieringen. Det mer detaljerade testflödet beskrivs i tabell 5.8 under avsnitt 5.2.7 Load Test.

4.5.7 Compare Test

Som tidigare nämnts i den tekniska diskussionen, stödjer PiiGAB:s gateways fler kommunikationsprotokoll än M-Bus. Det var därför relevant att implementera funktionalitet för fler protokoll. I detta arbete valdes Modbus som nästa protokoll att inkludera, eftersom en intern utbildning om protokollet hade genomförts och samtliga PiiGAB 900-modeller stödjer Modbus. Testet som implementerades fick namnet `Compare Test` och syftade till att jämföra ett mätvärde som hämtades via Modbus med motsvarande värde som hämtades via M-Bus.

Testet verifierar alltså att samma mätvärde kunde läsas av genom två olika kommunikationsvägar. Detta är relevant eftersom gatewayen kan fungera som en brygga mellan olika protokoll, där mätdata behöver vara samma oavsett hur den hämtas.

Övergripande fungerar testet genom att applikationen först hämtar ett valt mätvärde från ett Modbus-register. Därefter hämtas motsvarande mätvärde från mätaren via M-Bus. De två värdena extraheras sedan och jämförs med varandra. Om värdena inte överensstämmer betraktas testet som misslyckat. Det mer detaljerade testflödet beskrivs i tabell 5.9 under avsnitt 5.2.8 Compare Test.

4.6 Primäradressering

I den verkliga värden finns det miljöer som istället för sekundäradresser använder sig av primäradresser för att föra kommunikation mellan gateway och mätare. Detta medförde behovet av att implementera tester som just gör detta. Därav implementerades primäradressering för två av testerna, *Load Test* och *Read Test*. Skillnaden jämfört med sekundäradressering är att testet inte inleds med ett `Slave_Select`-kommando baserat på mätarens sekundäradress. I stället skickas ett `SND_NKE`-kommando till den primäradress som mätaren har tilldelats.

En begränsning med primäradressering är att primäradresser inte nödvändigtvis är unika i ett system. Det är därmed möjligt att två mätare har tilldelats samma primäradress, vilket kan leda till otydlighet kring vilken mätare som en förfrågan är för. Implementeringen krävde relativt lite ny funktionalitet i själva testlogiken, men ställde större krav på att laborationsmiljön var korrekt uppsatt. Det behövde säkerställas att de mätare som skulle användas i testerna hade unika primäradresser, samt att gatewayens konfiguration i mätarlistan överensstämde med den fysiska uppsättningen av mätare och de simulerade mätarna.

4.7 Förenklad testkörning

En central del av arbetet var att förenkla exekveringen av testerna samt att göra testmiljön mer flexibel och användaranpassad. I den ursprungliga lösningen krävde testkörningen att användaren hade god kännedom om både testernas interna uppbyggnad och hur parametrar skulle anges. Detta innebar att även mindre förändringar i testerna krävde manuella ändringar direkt i testfilerna och vid testning med flera mätare behövde testet kopieras. För att öka användbarheten hos testsystemet genomfördes därför flera förbättringar av hur tester konfigurerades och exekverades.

4.7.1 Konfigurering av tester

I den tidiga versionen av testsystemet var testvariabler hårdkodade direkt i Robot Framework. Detta var tillräckligt under den inledande utvecklingsfasen, då fokus främst låg på att verifiera att enskilda tester fungerade. När behovet av att köra samma tester mot flera olika mätare denna lösning snabbt en begränsning. För att lösa detta infördes en separat konfigurationsfil i form av `config.py`. Genom att samla parametrar i en särskild konfigurationsfil kunde testernas beteende ändras utan att själva testlogiken behövde modifieras.

I samband med detta ändrades även sättet på vilket testerna exekverades. I stället för att köra testerna direkt från terminalen med Robot Framework infördes ett Python-baserat exekveringsflöde. Denna förändring motiverades av att Python gav större möjligheter att bearbeta konfigurationsdata, välja testfall dynamiskt och automatisera hur parametrar skickades vidare till Robot Framework.

En konsekvens av denna arkitektur var att det behövdes en mekanism som översatte och förmedlade variabler mellan konfigurationsfilen och de enskilda Robot Framework-testerna. Eftersom olika tester använder olika parametrar gjordes valet att inte endast implementera ett mellanlager för alla tester, i stället utvecklades ett mellanlager per test. Dessa mellanlager fick namnet *builders* och hade som uppgift att läsa in relevanta parametrar från konfigurationen, strukturera dem på korrekt sätt och därefter vidarebefordra dem till motsvarande test. Denna uppdelning gjorde implementationen mer modulär, eftersom varje builder kunde anpassas efter kraven för ett specifikt test utan att påverka övriga delar av systemet.

Utöver den grundläggande konfigurationshanteringen utvecklades även funktionalitet för att definiera testsekvenser, det vill säga möjligheten att exekvera flera tester i en förutbestämd ordning. Behovet av detta uppstod när mer verkliga fall skulle testas. Ett exempel är hur gatewayen betedde sig före och efter en viss åtgärd, exempelvis efter en uppdatering. Det som skulle verifieras då var att avläsningen fungerade likadant före och efter uppdatering av gatewayens paket.

En sådan testsekvens kunde definieras enligt följande:

`Delete Packages` → `Read Test` → `Install Packages` → `Read Test`

I denna sekvens avinstallerades först ett eller flera paket, därefter genomfördes ett avläsningstest. Därefter installerades paket på nytt och samma avläsningstest genomfördes igen. Eftersom `Read Test` jämför mottagna svar med fördefinierade telegram blev det möjligt att avgöra om paketuppdateringen hade påverkat gatewayens avläsningsfunktion. På så sätt kunde testsekvensen användas för att verifiera att systemets funktionalitet förblev oförändrad trots förändringar i mjukvarumiljön.

Möjligheten att definiera testsekvenser bidrog också till att minska mängden manuellt arbete vid återkommande verifiering. I stället för att användaren själv behövde starta varje test i rätt ordning kunde sekvensen beskrivas en gång och därefter återanvändas. Detta förbättrade reproducerbarheten i testningen och minskade risken för handhavandefel, exempelvis att ett test kördes i fel ordning. Det lades även till funktionalitet för att köra en testsekvens ett förvalt antal gånger eller oändligt.

Konfigurationsmöjligheterna vidareutvecklades senare ytterligare genom att stöd infördes för flera konfigurationsfiler i stället för en enda gemensam fil. Detta gjordes eftersom olika testscenarier ofta krävde olika uppsättningar parametrar och sekvenser. För att förenkla användningen skapades därför flera fördefinierade konfigurationsfiler, anpassade för olika typer av testfall. Vid exekvering kunde användaren välja önskad konfigurationsfil genom att ange dess filnamn som ett argument. Därmed blev det möjligt att snabbt växla mellan olika testsekvenser utan att behöva ändra innehållet i konfigurationsfilerna manuellt inför varje körning.

4.7.2 Dokumentation

Under projektets inledande fas fanns en enklare `README.md`-fil som huvudsakligen användes för att samla grundläggande information om hur testsystemets exekverades. Detta var tillräckligt så länge systemet var relativt begränsat, men i takt med att funktionaliteten byggdes ut blev det tydligt att dokumentationen behövde utvecklas ytterligare. Särskilt efter att flödet för konfigurering och exekvering av tester förändrades från en direkt Robot Framework-baserad körning till ett Python-baserat exekveringsflöde med externa konfigurationsfiler. Det uppstod då behovet av att både omstrukturera och utöka dokumentationen.

En viktig del av detta arbete var att anpassa dokumentationen efter användarens faktiska arbetsflöde. I stället för att främst beskriva enskilda funktioner omstrukturerades innehållet så att det följde den ordning i vilket testsystemet konfigureras och startas. Detta var nödvändigt eftersom testsystemet inte enbart skulle användas av utvecklare och därför behövde bli mer användarvänligt. Dokumentationen kom därmed att inledas med de förutsättningar som krävs, såsom nödvändig programvara, för att därefter beskriva hur testmiljön sätts upp och slutligen hur tester konfigureras, exekveras och analyseras. Denna struktur gjorde dokumentationen mer som en guide för nya användare. Dokumentation för befintliga *keywords* och dess funktion lades även till som stöd vid fortsatt utveckling av systemet.

För att utvärdera hur väl dokumentationen fungerade i praktiken genomfördes ett användartest under arbetets gång. Efter att en stor del av testsystemet implemente-

rats och en steg-för-steg uppstartsguide skrivits, fick en anställd på företaget, med handledning, försöka konfigurera och exekvera en testsekvens. Detta gav värdefulla insikter om vilka delar av dokumentationen som var otydliga, vilka moment som krävde ytterligare förklaring och vilka antaganden som hade gjorts om användarens installerad programvara. Erfarenheterna från detta tillfälle låg till grund för flera förbättringar av `README.md`-filen, både vad gäller struktur och detaljnivå.

4.8 Verifiering

Vid utvecklingen av ett test arbetades det mot att testet skulle lyckas, eftersom de simulerade mätarna och gatewayen fungerade korrekt. När testerna har verifierat att gatewayen fungerar som förväntat, måste även testerna i sig verifieras. Detta eftersom det annars finns risk för att testerna alltid ger positiva utfall och inte fångar upp fallen då gatewayen misslyckas. Testerna verifierades genom fyra tillvägagångssätt, vilka förklaras nedan.

4.8.1 Manuell avläsning

Under implementationen av testerna användes utskrifter i terminalen som ett stöd för att verifiera vilka värden som lästes in och jämfördes i testfallen. Detta visade sig vara särskilt användbart vid felsökning när tester gav misslyckade resultat, men även för att manuellt kunna validera resultat i de fall där testerna lyckades.

Robot Framework erbjuder, som tidigare nämnts, en inbyggd rapporteringsarkitektur där ett test loggas stegvis under exekvering. Dessa rapporter användes som ett komplement till terminalutskrifterna för att analysera testprocessen och identifiera avvikelser.

Eftersom testerna exekveras genom Python genereras en separat rapport per testkörning. Detta medförde att det blev svårt att få en överblick av hela resultatet efter en testsekvens. För att förbättra läsbarheten implementerades därför en lösning där flera rapporter sammanställdes till en gemensam rapport. Detta möjliggjorde en mer effektiv analys av testresultaten och underlättade jämförelser mellan olika körningar.

4.8.2 Enhetstester

För att verifiera funktionaliteten hos de utvecklade Robot Framework-keywordsen implementerades ett antal enhetstester. Dessa skrevs löpande i samband med att nya keywords lades till i testsystemet. Enhetstesterna utformades genom att varje keyword kördes med både giltig och ogiltig indata, där resultatet därefter jämfördes med ett förväntat utfall.

Ett exempel på detta är verifieringen av `ACK`-meddelanden. Som tidigare nämnt under 3.4.1 består ett M-Bus `ACK`-meddelande av byten $E5_{16}$. För att verifiera detta implementerades ett enhetstest för `Verify Ack`. Testet kontrollerar att *keywordet* returnerar sant när korrekt byte tas emot och falskt när annan eller saknad data tas

emot. Testfallen visas i tabell 4.1.

Tabell 4.1: Enhetstest för `Verify Ack`-keyword

Indata	**Assertion**
$E5_{16}$	`True`
$E4_{16}$	`False`
`NULL`	`False`

Ett ytterligare enhetstest implementerades för ett *keyword* som kontrollerar att ett mottaget telegram tillhör rätt mätare. Detta *keyword* tar emot ett telegram och en sekundäradress som indata och kontrollerar om telegrammet innehåller den angivna adressen. Testet kördes med ett telegram som innehöll rätt sekundäradress och med ett telegram som inte gjorde det. Testfallen visas i tabell 4.2.

Tabell 4.2: Enhetstest för verifiering av att telegram tillhör rätt mätare

Indata	**Assertion**
Sekundäradress & telegram med samma sekundäradress	`True`
Sekundäradress & telegram utan samma sekundäradress	`False`

På motsvarande sätt skrevs enhetstester för flera av de övriga *keywords* som användes i testsystemet. Testerna användes för att kontrollera att funktionerna gav förväntade resultat innan de användes i större testflöden.

4.8.3 Fysisk verifiering

Utöver manuell avläsning och enhetstester genomfördes även fysisk verifiering av testmiljön. Syftet var att kontrollera att testerna reagerade som förväntat när förutsättningarna i laborationsmiljön förändrades. Detta var särskilt relevant för `Value Test`, eftersom testet var avsett att användas mot verkliga mätare där mätvärden kan variera över tid.

För att verifiera `Value Test` placerades temperaturmätaren tillfälligt utanför ett fönster. På så sätt skapades en faktisk temperaturförändring som kunde avläsas av mätaren. Testet användes därefter för att kontrollera att förändringen identifierades när mätvärdet avvek mer än den angivna toleransen. Detta gav en praktisk verifiering av att testet inte enbart fungerade mot statiska eller simulerade värden, utan även kunde reagera på förändringar i en fysisk miljö.

Även andra delar av testmiljön verifierades genom att medvetet skapa fel. Exempelvis stängdes mätarsimulatorn av under testkörning och felaktiga värden skrevs in i testdatafiler. Dessa kontroller användes för att säkerställa att testsystemet kunde hantera avvikande situationer och att testerna misslyckades när förutsättningarna inte längre motsvarade det förväntade beteendet.

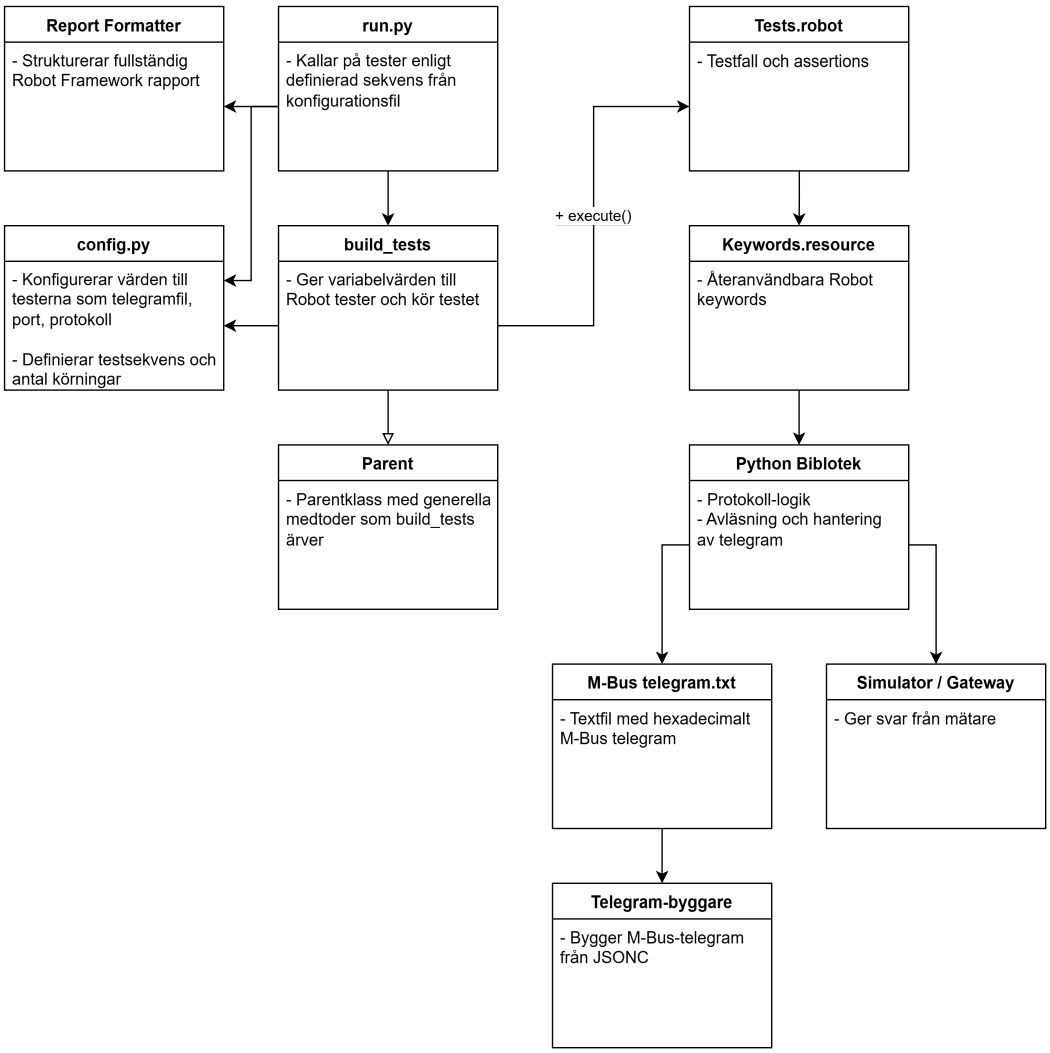
5

Resultat

I detta kapitel beskrivs testsystemets övergripande struktur och de tester som implementerats. Inledningsvis presenteras en förenklad systemöversikt över testsystemets centrala komponenter, följt av en beskrivning av hur dessa samverkar vid konfiguration, exekvering och rapportering av testsekvenser. Därefter presenteras de implementerade testerna med hjälp av tabeller som visar deras huvudsakliga förlopp. För att underlätta förståelsen av tabellerna redovisas även ett sekvensdiagram som visar det grundläggande kommunikationsflödet mellan applikation, gateway och mätare.

5.1 Programstruktur

En förenklad systemöversikt togs fram för att visa hur testsystemets centrala komponenter hänger samman. Figuren visar framför allt sambandet mellan konfiguration, testexekvering, Robot Framework-filer, Python-bibliotek och rapportering. För att göra översikten lättare att tolka har mindre detaljer, hjälpfunktioner och interna beroenden utelämnats.



Figur 5.1: Förenklad systemöversikt över testsystemets komponenter.

`config.py` används som konfigurationsfil för testsystemet. I denna definieras bland annat testsekvensen som ska köras, mätarna som ska ingå i testerna, samt parametrar för respektive test. Exempel på sådana parametrar är toleransvärden i `Value Test` och Modbus-register i `Compare Test`.

`run.py` ansvarar för testkörningen. Programmet läser in den valda konfigurationen och anropar därefter rätt `build_test`-fil utifrån den testsekvens som har definierats. När testsekvensen har slutförts anropas även rapporteringsfunktionen för att sammanställa resultatet.

`build_test`-filerna fungerar som ett mellanlager mellan konfigurationen och Robot Framework-testerna. Dessa filer hämtar relevanta värden från `config.py`, strukturerar parametrarna för det aktuella testet och skickar dem vidare till `Tests.robot`. På så sätt kan samma testlogik återanvändas med olika konfigurationer.

`Tests.robot` innehåller de Robot Framework-testfall som används i testsystemet.

Testfallen bygger på *keywords* som definieras i separata `.resource`-filer. Dessa *keywords* används för att strukturera testflödet och anropa den underliggande funktionaliteten, exempelvis för kommunikation, telegramhantering och verifiering av mottagen data.

Den mer tekniska implementationen av flera *keywords* finns i Python-bibliotek. Dessa filer innehåller logik för exempelvis protokollhantering, telegramtolkning, anslutningar och jämförelser mellan förväntad och mottagen data.

`Report Formatter` ansvarar för sammanställning av rapporter från flera separata Robot Framework-körningar till en gemensam rapport för en testsekvens.

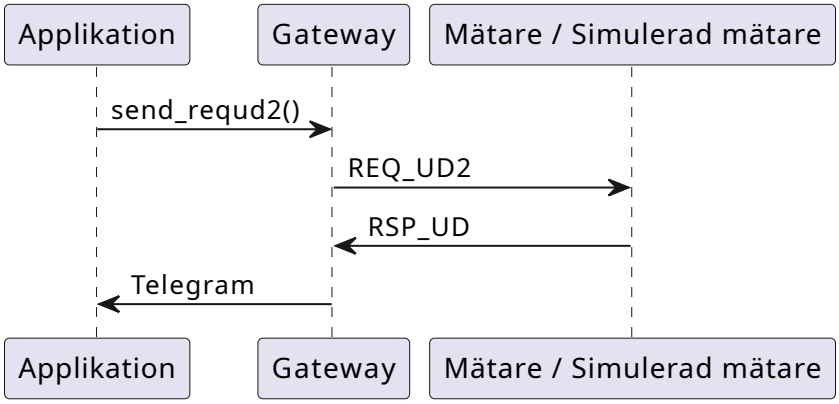
5.2 Implementerade tester

I detta avsnitt presenteras de tester som implementerats i testsystemet. Testerna redovisas med hjälp av tabeller för att ge en tydlig överblick över deras huvudsakliga förlopp och vilka steg som ingår i respektive test. Detta gör det möjligt att beskriva testernas logik på en mer överskådlig nivå utan att gå in på detaljer i implementationen.

Testerna bygger på *assertions*, vilket innebär att ett faktiskt utfall jämförs med ett förväntat utfall. Om jämförelsen stämmer godkänns steget, annars misslyckas testet. När det står `Assert` i tabellerna innebär det därför att testsystemet kontrollerar att ett visst villkor är uppfyllt.

Om ett steg tidigare än `Assert`-steget i testflödet misslyckas, exempelvis om ingen kommunikation kan etableras, kan den avslutande jämförelsen inte genomföras. I sådana fall avbryts testet i förtid. Testerna kan därav inte enbart misslyckas på grund av felaktig telegramdata, utan även på grund av fel i kommunikationen eller systemets uppsättning.

Flera av testerna bygger på samma grundläggande kommunikationsflöde mellan applikationen, gatewayen och mätaren. För att göra de efterföljande tabellerna lättare att tolka visas därför först ett sekvensdiagram i figur 5.2. Diagrammet visar att applikationen skickar en förfrågan till gatewayen, att gatewayen vidarebefordrar förfrågan till mätaren och att svaret sedan skickas tillbaka samma väg. Gatewayen fungerar därmed som en mellanliggande kommunikationsnod.



Figur 5.2: Sekvensdiagram över ett `REQ_UD2`-kommando.

5.2.1 Data Test

Tabell 5.1: Verifiering av `RSP_UD` (mätarsvar)

Steg	Beskrivning
1	`Slave select` skickas från applikationen till gatewayens slavport.
2	Mätaren svarar `ACK` till gatewayen som sedan applikationen läser av.
3	Applikationen skapar en `REQ_UD2` förfrågning som gatewayen skickar till mätaren.
4	Mätaren svarar med `RSP_UD`-telegram till gatewayen som applikationen läser av och sparar, T_1.
5	Applikationen hämtar telegrammet T_2 från simulatorn.
6	Applikationen jämför telegrammen: `Assert` T_1 `==` T_2.

Detta test bygger alltså på två telegram, ett telegram som läses via gatewayen och ett referenstelegram från simulatorn. Resultatet avgörs genom att dessa jämförs med varandra.

5.2.2 Read Test

Tabell 5.2: Verifiering av `RSP_UD` (mätarsvar)

Steg	Beskrivning
1	`Slave select` skickas från applikationen till gatewayens slavport.
2	Mätaren svarar `ACK` till gatewayen som sedan applikationen läser av.
3	Applikationen skapar en `REQ_UD2` förfrågning som gatewayen skickar till mätaren.
4	Mätaren svarar med `RSP_UD`-telegram till gatewayen som applikationen läser av och sparar, T_1.
5	Applikationen nollställer `Access number` och `Checksum`.
6	Applikationen skapar en ny förfrågning till gatewayen och får tillbaka telegrammet T_2.
7	Applikationen jämför telegrammen: `Assert` T_1 `==` T_2.

Även detta test bygger på två telegram, likt *Data Test*, men notera skillnaden att T_2, alltså telegrammet som jämförs inte hämtas från simulatorn och behöver därmed inte vara känt i förtid.

5.2.3 Packet Install/Uninstall

Tabell 5.3: Installation av paket med opkg via SSH

Steg	Beskrivning
1	SSH-anslutning etableras mot gateway.
2	`opkg update` körs.
3	Installation genom `opkg install <paket>`
4	Varje kommando verifieras (`exit_status == 0`).
5	SSH-anslutning stängs.

Testet för avinstallation visas i tabellen 5.4 nedan. Notera att processen skiljer sig något jämfört med installations-testet, eftersom `opkg update` inte är nödvändigt då kommandot uppdaterar lokala listan över paket.

Tabell 5.4: Avinstallation av paket med opkg via SSH

Steg	Beskrivning
1	SSH-anslutning etableras mot gateway.
2	Avinstallation genom `opkg remove <paket>`
3	Varje kommando verifieras (`exit_status == 0`).
4	SSH-anslutningen stängs.

Till skillnad från *Data Test* och *Read Test* verifierar `Packet Install/Uninstall` inte innehållet i M-Bus-telegram. Istället kontrolleras att kommandona som skickas via SSH genomförs korrekt. Detta görs genom att kontrollera kommandonas statuskod, där `exit_status == 0` indikerar att kommandot lyckades.

5.2.4 Corrupted Test

Tabell 5.5: Verifiering av korrupt data från mätare

Steg	Beskrivning
1	`Slave_select` kommando skickas från applikationen till gatewayens slavport mot en mätare som är konfigurerad att ge korrupt data.
2	Mätaren svarar `ACK` till gatewayen som sedan applikationen läser av.
3	Applikationen skapar en `REQ_UD2` förfrågan som gatewayen skickar till mätaren.
4	Mätaren svarar med ett korrupt `RSP_UD` till gatewayen och gatewayen bör kasta det korrupta svaret. `Response timeout` meddelande skickas istället från gatewayen till applikationen.
5	Därefter upprepas testet mot en mätare med korrekt data och applikationen verifierar att gatewayen mottagit ett korrekt `RSP_UD`-telegram.

Testet bygger på två efterföljande avläsningar. Först skickas en förfrågan till en mätare som är konfigurerad att svara med ett korrupt telegram, där gatewayen förväntas ignorera svaret och returnera ett timeout-meddelande. Därefter görs en ny avläsning mot en mätare med giltig data för att verifiera att gatewayen fortfarande kan ta emot ett korrekt `RSP_UD`-telegram.

5.2.5 Timeout Test

Tabell 5.6: Verifiering av timeout-hantering

Steg	Beskrivning
1	`Slave_select` skickas från applikationen till gatewayens slavport mot en icke-existerande sekundäradress.
2	Applikationen verifierar att en timeout uppstår hos gatewayen.
3	Därefter skickar applikationen `slave_select` till gatewayen mot en giltig mätare.
4	Applikationen verifierar att gatewayen fortfarande kan kommunicera korrekt efter timeout-situationen.

Detta test bygger på att en förfrågan först skickas till en sekundäradress som inte tillhör någon mätare, vilket förväntas leda till timeout. Därefter skickas en ny förfrågan till en giltig mätare för att verifiera att gatewayen fortfarande kan kommunicera korrekt efter en timeout-situation.

5.2.6 Value Test

Tabell 5.7: Verifiering av värde från mätare

Steg	Beskrivning
1	Applikationen skapar en `REQ_UD2`-förfrågan som gatewayen skickar till mätaren.
2	Mätaren svarar gatewayen med ett `RSP_UD`-telegram som applikationen läser av och sparar som T_1.
3	Applikationen skapar en ny `REQ_UD2`-förfrågan som gatewayen skickar till mätaren.
4	Mätaren svarar gatewayen med ett nytt `RSP_UD`-telegram som applikationen läser av och sparar som T_2.
5	Applikationen tar ut mätvärdet på index i från T_1 och T_2, vilket ger V_1 respektive V_2.
6	Applikationen jämför värdena mot toleransen: `nedre tolerans` $\leq \lvert V_1 - V_2 \rvert \leq$ `övre tolerans`.
...	Steg 3-6 upprepas enligt nedan.
n-3	Applikationen skapar en `REQ_UD2`-förfrågan som gatewayen skickar till mätaren.
n-2	Mätaren svarar gatewayen med ett `RSP_UD`-telegram som applikationen läser av och sparar som T_n.
n-1	Mätvärdet på index i tas ut från T_1 och T_n, vilket ger V_1 respektive V_n.
n	`nedre tolerans` $\leq \lvert V_1 - V_n \rvert \leq$ `övre tolerans`.

Testet hämtar det specificerade mätvärdet från flera `RSP_UD`-telegram som jämförs med det första avlästa värdet. I stället för att jämföra hela telegram kontrolleras endast mätvärdet på ett angivet index. Testet godkänns om avvikelsen mellan referensvärdet och senare avläsningar ligger inom den definierade toleransen.

5.2.7 Load Test

Tabell 5.8: Verifiering av mätares sekundäradress

Steg	Beskrivning
1	`Slave_select` kommando skickas från applikationen till gatewayens slavport med vald sekundäradress, $Adress_1$.
2	Mätaren svarar `ACK` till gatewayen som sedan applikationen läser av.
3	Applikationen skapar en `REQ_UD2` förfrågan som gatewayen skickar till mätaren.
4	Mätare svarar med `RSP_UD2`, vilket innehåller dess sekundäradress, $Adress_2$
5	Applikationen jämför sekundäradresserna: `Assert` $Adress_1$ `==` $Adress_2$.)

I detta test används sekundäradressen för att kontrollera att svaret kommer från den mätare som adresserades. Testet väljer först en mätare med `Slave_select` och läser därefter ut sekundäradressen ur mätarens svarstelegram. Testet godkänns om den adresserade sekundäradressen överensstämmer med adressen i svaret.

5.2.8 Compare Test

Tabell 5.9: Jämförelse av värde mellan M-Bus och Modbus

Steg	Beskrivning
1	`Slave_select` skickas från applikationen till gatewayens slavport med vald sekundäradress.
2	Mätaren svarar `ACK` till gatewayen som sedan applikationen läser av.
3	Applikationen skapar en `REQ_UD2`-förfrågan som gatewayen skickar till mätaren.
4	Mätaren svarar med ett `RSP_UD`-telegram som applikationen läser av.
5	Applikationen tar ut det valda mätvärdet ur `RSP_UD`-telegrammet, vilket sparas som $V_{\texttt{M-Bus}}$.
6	Applikationen läser motsvarande mätvärde från ett valt Modbus-register, vilket sparas som $V_{\texttt{Modbus}}$.
7	Applikationen jämför mätvärdena: `Assert` $V_{\texttt{M-Bus}}$ `==` $V_{\texttt{Modbus}}$.

`Compare Test` jämför ett mätvärde som läses via Modbus med motsvarande värde som hämtas via M-Bus. Testet använder därmed två olika kommunikationsvägar för samma mätdata. Om värdena överensstämmer godkänns testet, vilket visar att mätvärdet kan läsas konsekvent oavsett protokoll.

6

Diskussion

Detta kapitel diskuterar projektets resultat i förhållande till arbetets mål och frågeställningar. Fokus ligger på hur den automatiserade testmiljön har utformats, vilken praktisk nytta den kan ge PiiGAB och vilka begränsningar som finns i den framtagna lösningen.

Inledningsvis diskuteras hur projektets genomförande påverkades av förändringar i tidsplan och prioriteringar. Därefter behandlas de tekniska valen, testsystemets struktur och hur dessa påverkar möjligheten till vidareutveckling. Kapitlet diskuterar även teststrategier, testdata, verifiering och användarvänlighet, eftersom dessa delar är centrala för att ett automatiserat testsystem ska vara användbart i praktiken.

Vidare diskuteras AI som stöd i testprocessen, uppfyllelse av projektets krav samt etiska, samhälleliga och ekologiska aspekter. Avslutningsvis behandlas erfarenheter från att arbeta i en verklig företagsmiljö, projektets begränsningar och möjliga områden för fortsatt arbete. Kapitlet avslutas med en sammanfattande diskussion där de viktigaste slutsatserna knyts samman.

6.1 Planerad tidsplan mot faktisk tidsplan

Den preliminära tidsplanen, i form av Gantt-schemat i figur 2.1, gav en övergripande bild av hur projektets olika moment var tänkta att genomföras. Under arbetets gång visade det sig dock att flera moment tog längre tid än förväntat. Detta gällde särskilt implementationen av M-Bus-relaterade tester, där arbetet utvecklades från att först omfatta ett verifierat test till att även innefatta fler testfall, konfigurationsstöd och förbättrad modularitet.

Att tidsplanen förändrades berodde inte enbart på underskattad tidsåtgång, utan även på att projektets inriktning anpassades efter nya insikter och företagets behov. Ett exempel är att Modbus inkluderades som ytterligare protokoll, trots att detta inte var ett separat moment i den ursprungliga planeringen. Detta breddade testmiljön och gjorde resultatet mer relevant för PiiGAB:s gateways, men innebar samtidigt att tid behövde omfördelas från andra delar.

Även användarvänlighet fick större betydelse än vad som först planerades. Projektet utgick inledningsvis från att testsystemet främst skulle användas av utvecklare, men under arbetets gång blev det tydligt att även personer med mindre programmeringsvana skulle kunna ha nytta av systemet. Därför lades mer tid på förenklad konfiguration, exekvering och dokumentation.

Tidsplanen följdes alltså inte fullt ut, men avvikelserna bidrog till att arbetet bättre kunde anpassas efter projektets mål om en användbar och vidareutvecklingsbar testmiljö. Samtidigt visar detta att en mer detaljerad preliminär tidsplan hade kunnat underlätta prioriteringar när nya behov uppstod. I praktiken fungerade tidsplanen därför främst som ett övergripande planeringsstöd snarare än som en exakt styrning av projektets genomförande.

6.2 Tekniska val och arkitektur

Valet av Python och Robot Framework visade sig vara väl anpassat till projektets behov. Python möjliggjorde snabb vidareutveckling av befintlig kod, hantering av telegramdata och implementation av kompletterande verktyg. Robot Framework bidrog med en tydlig teststruktur där testfall kan skrivas på en högre abstraktionsnivå än ren programkod. Detta var relevant eftersom testsystemet inte enbart skulle kunna utvecklas av programmerare, utan även kunna förstås av andra roller på företaget.

Den valda arkitekturen bidrog också till att uppfylla målet om en modulär testmiljö. Genom att separera testlogik, kommunikationslogik, konfiguration och rapportering skapades en struktur där nya tester kan läggas till utan att hela systemet behöver byggas om. Detta är en viktig styrka, eftersom arbetet inte endast syftade till att skapa enstaka testfall, utan en grund för fortsatt testautomatisering.

En nackdel med denna struktur är att systemet består av flera lager, exempelvis konfigurationsfiler, builders, Robot Framework-filer, resource-filer och Python-bibliotek. Detta kan göra systemet svårare att förstå för nya utvecklare. Samtidigt var denna uppdelning nödvändig för att skapa den flexibilitet och återanvändbarhet som eftersträvades. Ett enklare system med färre lager hade troligen varit lättare att förstå på kort sikt, men svårare att bygga vidare på när fler tester och protokoll tillkommer.

6.3 Testsystemets faktiska nytta

Testsystemet kan i sin nuvarande form verifiera flera centrala delar av gatewayens kommunikation. Det omfattar framför allt M-Bus-relaterade flöden, hantering av telegram, felhantering, pakethantering samt jämförelse av mätvärden mellan M-Bus och Modbus. Detta innebär att lösningen kan användas som stöd vid återkommande verifiering, exempelvis efter uppdatering av mjukvarupaket eller vid kontroll av att kommunikationsflöden fortfarande fungerar efter förändringar.

Samtidigt är testvariationen fortfarande begränsad. Testsystemet täcker inte alla

protokoll, mätartyper eller funktioner som förekommer i PiiGAB:s gateways. Det ger därför inte en heltäckande verifiering av gatewayens funktionalitet. Däremot har varje nytt test bidragit med funktionalitet som kan återanvändas vid framtida testfall. Exempel på detta är avläsning av Modbus-register, uttag av specifika mätvärden ur M-Bus-telegram och hantering av korrupta telegram.

Detta innebär att arbetets nytta inte enbart ligger i de tester som implementerats, utan även i den struktur och de komponenter som skapats. Även om testtäckningen behöver byggas ut, finns det nu en grund där nya tester kan definieras, konfigureras, exekveras och rapporteras på ett mer enhetligt sätt. Detta gör lösningen användbar redan i sin nuvarande form, men framför allt som en bas för fortsatt utveckling.

För att testsystemet ska få större praktisk nytta bör det i framtiden byggas ut med fler protokoll och fler testscenarier. Detta kan exempelvis omfatta ytterligare testfall för Modbus, Wireless M-Bus, MQTT, LTE och LoRa. Det kan även vara relevant att inkludera scenarier med flera mätare och mer komplexa kommunikationsflöden för att bättre efterlikna verkliga installationer.

6.4 Teststrategier och verifiering

De implementerade testerna visar att olika typer av verifiering behövs för att bedöma ett kommunikationssystems tillförlitlighet. Bytevis jämförelse av telegram är användbart när testdata är känd och statisk, exempelvis vid simulering. Denna metod passar däremot sämre för verkliga mätare där mätvärden kan förändras mellan avläsningar. Därför behövdes även tester där specifika mätvärden jämförs inom en angiven tolerans.

Felhanteringstesterna är också viktiga för att verifiera gatewayens robusthet. Om ett system endast testas med korrekta telegram riskerar testningen att ge en för begränsad bild av systemets funktion. I praktiska installationer kan felaktiga telegram, uteblivna svar eller felaktiga adresser förekomma. Tester som *Corrupted Test* och *Timeout Test* bidrar därför till att verifiera att gatewayen inte bara fungerar i normalfallet, utan även kan hantera avvikande situationer.

Samtidigt behöver även testsystemet i sig verifieras. Eftersom testsystemet används för att bedöma gatewayens funktionalitet kan fel i testlogik, konfiguration eller testdata ge missvisande resultat. Projektet har hanterat detta genom manuell avläsning, enhetstester och fysisk verifiering. Det finns dock utrymme för mer systematisk validering, särskilt av de verktyg som genererar eller tolkar testdata.

6.5 Testdata

Känd, kontrollerbar och reproducerbar testdata visade sig vara en central förutsättning för automatiserad testning av gateways. Det räcker inte att verifiera att data tas emot, utan den mottagna datan måste kunna jämföras med ett förväntat resultat. Detta är särskilt viktigt vid M-Bus-kommunikation, där telegrammens struktur,

kontrollfält och mätvärden påverkar hur datan ska tolkas.

Om testdatan varierar på ett okontrollerat sätt blir det svårare att avgöra om ett misslyckat test beror på gatewayen, testmiljön eller testdatan. Vid manuell modifiering av M-Bus-telegram finns dessutom risk för fel, eftersom exempelvis längdfält och kontrollsumma behöver uppdateras korrekt vid ändringar.

Verktyget *Json2Telegram* utvecklades för att minska denna osäkerhet. Genom att generera telegram från en mer läsbar representation blev det enklare att skapa simulerade mätare med känt innehåll. Detta stärkte möjligheten att skapa reproducerbara testscenarier och gjorde testmiljön mer flexibel.

Samtidigt innebär *Json2Telegram* att ett nytt beroende införs i testsystemet. Om verktyget genererar felaktiga telegram kan samma fel påverka flera tester. Därför behöver även detta verktyg verifieras noggrant. Detta visar att automatiserad testning inte eliminerar behovet av kontroll, utan flyttar en del av ansvaret till testdata och de verktyg som används för att skapa den.

6.6 Konfiguration, exekvering och rapportering

Konfiguration, exekvering och rapportering visade sig vara avgörande för testsystemets praktiska användbarhet. Ett automatiserat testsystem behöver inte bara kunna utföra tester, utan även kunna anpassas till olika scenarier och presentera resultat på ett tydligt sätt. Genom att flytta testvärden från hårdkodade delar av Robot Framework till externa konfigurationsfiler separerades testdata från testlogik.

Denna separation gjorde systemet mer flexibelt. Samma testlogik kan användas med olika mätare, protokoll och parametrar utan att testfilerna behöver ändras. Stödet för flera konfigurationsfiler gjorde det dessutom möjligt att behandla olika testupplägg som återanvändbara scenarier.

Det Python-baserade exekveringsflödet bidrog till att testsekvenser kunde köras automatiskt i en bestämd ordning. Detta är särskilt värdefullt vid återkommande verifiering, exempelvis när funktionalitet ska kontrolleras före och efter en uppdatering av gatewayens paket. På så sätt minskar risken för handhavandefel och testprocessen blir mer reproducerbar.

Rapporteringen är också viktig för att testresultaten ska kunna användas i praktiken. Ett misslyckat test kan bero på flera olika orsaker, exempelvis fel i kommunikation, felaktig konfiguration eller avvikande mätdata. Den sammanställda rapporteringen ger därför ett bättre stöd för att analysera testsekvenser och identifiera var ett fel har uppstått.

6.7 Användarvänlighet

Användarvänlighet är en viktig del av testsystemets faktiska nytta. Även om systemet fungerar tekniskt kan användningen bli begränsad om det är svårt att konfigurera, köra eller tolka resultaten. Detta är särskilt relevant eftersom systemet inte enbart kan komma att användas av utvecklare, utan även av personer med mindre programmeringsvana.

Den praktiska utvärderingen med en anställd på företaget visade att användarvänlighet inte kan bedömas enbart av de som utvecklat systemet. För utvecklarna kan vissa moment framstå som självklara, eftersom de har byggt upp förståelsen för systemets struktur under arbetets gång. För en ny användare kan samma moment däremot vara otydliga.

Arbetet med dokumentation, tydligare konfigurationsstruktur och förenklad exekvering var därför viktigt för att göra systemet mer tillgängligt. Samtidigt kräver användning av testsystemet fortfarande viss teknisk förståelse. Användaren behöver förstå både konfigurationsfilerna och de kommunikationsflöden som testas. Om systemet i framtiden ska användas mer regelbundet av personer med mindre teknisk bakgrund kan ytterligare stöd behövas, exempelvis mallar, validering av konfiguration eller ett mer styrt användargränssnitt.

6.8 AI som stöd i testprocessen

En av projektets frågeställningar rörde hur AI-genererade tester förhåller sig till manuellt skrivna tester och om AI kan användas för att stödja testprocessen. Arbetet visade att AI kan vara användbart i tidiga och mer övergripande delar av processen. AI kunde exempelvis bidra vid informationsinhämtning, dokumentation, idéutveckling och resonemang kring möjliga testfall.

Däremot visade sig AI vara mindre tillförlitligt vid mer detaljerad och protokollnära funktionalitet. Detta märktes särskilt vid arbete med M-Bus-telegram, där korrekt tolkning av telegramstruktur kräver förståelse för protokollets fält, kontrollsumma, längdangivelser och datakodning. AI-genererade förslag kunde därför se rimliga ut på en övergripande nivå, men ändå innehålla tekniska fel.

Detta innebär att AI inte bör användas som en självständig källa för testgenerering eller verifiering i ett sådant system. I ett testsystem som kan påverka bedömningen av produktkvalitet krävs teknisk granskning av personer som förstår både produkten och testmiljön. AI kan därmed fungera som ett kompletterande stöd, men inte ersätta mänsklig kontroll.

Av denna anledning integrerades ingen AI-modell direkt i den färdiga applikationen. Slutsatsen från projektet är att AI kan effektivisera vissa delar av testprocessen, men att den behöver användas med tydliga begränsningar och alltid kombineras med verifiering.

6.9 Uppfyllelse av krav

Projektets mål var att ta fram en automatiserad testmiljö för PiiGAB:s gateways, med stöd för insamling, loggning och sammanställning av testresultat. Detta mål har till stor del uppnåtts. Testsystemet kan exekvera testsekvenser, använda externa konfigurationsfiler och sammanställa resultat från flera körningar.

Målet om en modulär testmiljöarkitektur har också uppfyllts. Systemet är uppbyggt så att olika delar, exempelvis testlogik, kommunikationslogik och konfiguration, kan utvecklas och ändras separat. Detta gör det möjligt att bygga vidare på lösningen med fler tester och protokoll.

Målet om en fysisk testmiljö har också uppnåtts. En laborationsmiljö etablerades med gateway, mätare, mätarsimulator och nödvändig anslutningsutrustning. Denna miljö gjorde det möjligt att verifiera både simulerade och verkliga mätare.

De delar som inte uppfylldes fullt ut gäller AI-integrationen. AI undersöktes som stöd i testprocessen, men integrerades inte direkt i applikationen för automatisk testgenerering eller logganalys. Detta berodde på att AI-stödet fortfarande krävde teknisk granskning för att säkerställa korrekthet. Därmed uppfylldes målet om att undersöka AI:s användbarhet, men inte målet om en fullt integrerad AI-baserad funktionalitet.

6.10 Etiska, samhälleliga och ekologiska aspekter

Projektets samhälleliga relevans är kopplad till att gateways används för insamling och vidareförmedling av mätdata. Tillförlitlig mätdata kan vara viktig i system där värden används för analys, debitering eller styrning. Ett mer systematiskt testsystem kan därför bidra till att minska risken för att felaktiga mätvärden vidareförmedlas.

Ur ett ekologiskt perspektiv kan korrekt mätdata indirekt stödja bättre resursanvändning. Om mätvärden för exempelvis energi eller vatten är tillförlitliga förbättras möjligheten att upptäcka avvikelser och fatta välgrundade beslut. Testsystemet leder inte direkt till minskad resursförbrukning, men kan bidra till mer tillförlitliga system som används för uppföljning av resursanvändning.

De etiska aspekterna handlar framför allt om informationshantering och ansvar. Eftersom arbetet genomfördes i samarbete med PiiGAB förekom företagsintern information om produkter, testmiljöer och tekniska lösningar. Detta ställde krav på att information hanterades försiktigt och att rapportinnehåll granskades innan publicering.

AI-användningen förstärkte denna fråga. När AI används i projekt med företagsintern information behöver det finnas tydliga gränser för vilken information som får delas. Det är också viktigt att AI-genererade förslag inte används utan granskning. Ansvar för testernas korrekthet behöver fortsatt ligga hos människor med teknisk

förståelse för systemet.

6.11 Arbete i verklig miljö

En viktig erfarenhet från projektet är skillnaden mellan att utveckla en lösning i en kursmiljö och att utveckla en lösning för ett verkligt företag. I detta arbete behövde lösningen inte bara fungera vid en demonstration, utan även kunna förstås, användas och vidareutvecklas efter projektets slut.

Detta gjorde att struktur, dokumentation och reproducerbarhet fick stor betydelse. Kod som fungerar för utvecklarna kan ha begränsat värde om den är svår för andra att konfigurera eller felsöka. Därför blev arbetet med konfigurationsfiler, dokumentation och tydligare testflöden en central del av projektets praktiska nytta.

Att arbeta på plats hos företaget var en tydlig fördel. Det gav tillgång till hårdvara, interna verktyg och teknisk kompetens, vilket gjorde felsökning och anpassning enklare. Samtidigt innebar företagsmiljön att lösningen behövde passa befintliga produkter, arbetssätt och begränsningar. Projektet visar därmed att en användbar teknisk lösning inte alltid är den mest avancerade, utan den som är stabil, begriplig och anpassad till sammanhanget.

6.12 Begränsningar och fortsatt arbete

Den största begränsningen är att testsystemet fortfarande är en grund snarare än ett heltäckande testsystem. Det stödjer vissa centrala funktioner och protokoll, men täcker inte hela bredden av PiiGAB:s gateways. Fler protokoll, mätartyper och testscenarier behöver inkluderas för att systemet ska kunna användas mer heltäckande.

En annan begränsning är att laborationsmiljön inte fullt ut motsvarar en verklig installation. Verkliga miljöer kan innehålla fler mätare, längre kablage, störningar och mer varierade konfigurationer. Framtida arbete bör därför inkludera tester i mer realistiska miljöer.

Fortsatt arbete bör främst fokusera på tre områden. Det första är ökad testtäckning genom stöd för fler protokoll och fler scenarier. Det andra är förbättrad användbarhet, exempelvis genom tydligare mallar eller validering av konfigurationsfiler. Det tredje är förbättrad analys av testresultat, så att rapporteringen inte bara visar att ett test har misslyckats utan även ger bättre stöd för att förstå varför.

6.13 Sammanfattande diskussion

Diskussionen visar att projektets huvudsakliga värde ligger i kombinationen av implementerade tester och den struktur som skapats runt dem. Testsystemet kan verifiera flera viktiga delar av kommunikationen i PiiGAB:s gateways och har samtidigt utformats så att det kan byggas vidare på.

Arbetet besvarar därmed frågeställningarna genom att visa att automatiserad testning kan utformas med hjälp av en modulär arkitektur där konfiguration, testlogik, exekvering och rapportering separeras. Lösningen uppfyller flera av PiiGAB:s behov, men kräver fortfarande teknisk förståelse vid konfiguration, vidareutveckling och tolkning av resultat.

Projektet visar också att testdata och verifiering är avgörande delar av ett automatiserat testsystem. Verktyg som *Json2Telegram* kan göra testdata mer reproducerbar och kontrollerbar, men behöver själva verifieras för att inte skapa systematiska fel.

AI-delen visar att AI kan vara ett stöd i testprocessen, men inte en ersättning för teknisk granskning. AI kan användas för dokumentation, idéutveckling och övergripande resonemang, men är mindre tillförlitligt vid detaljerad protokollnära implementation. Därför krävs fortsatt mänsklig kontroll för att säkerställa testernas korrekthet.

Sammanfattningsvis har projektet resulterat i en användbar grund för automatiserad testning av inbyggda kommunikationssystem. Lösningen är inte ett färdigt heltäckande testsystem, men den ger PiiGAB en struktur som kan användas för fortsatt utveckling. Den viktigaste slutsatsen är att ett automatiserat testsystem får sitt verkliga värde först när det inte bara fungerar tekniskt, utan även är konfigurerbart, verifierbart, dokumenterat och möjligt att vidareutveckla i den miljö där det ska användas.

7

Slutsats

Projektet har resulterat i en automatiserad testmiljö för PiiGAB:s gateways. Testmiljön kan användas för att verifiera centrala delar av gatewayens kommunikation, främst inom M-Bus och Modbus, och kan därmed minska behovet av manuell testning vid återkommande verifiering.

Arbetet visar att en modulär struktur är viktig för att testsystemet ska kunna vidareutvecklas. Genom att separera testlogik, konfiguration, exekvering och rapportering har systemet blivit mer anpassningsbart. Funktionalitet som utvecklats för enskilda tester kan dessutom återanvändas vid framtida testfall.

En annan slutsats är att känd och reproducerbar testdata är avgörande vid automatiserad testning. Verktyget *Json2Telegram* gjorde det möjligt att skapa simulerade M-Bus-mätare med fördefinierade telegram, vilket gav bättre kontroll över testdata och stärkte testernas tillförlitlighet.

Målet att ta fram en fungerande och vidareutvecklingsbar testmiljö har till stor del uppnåtts. Däremot uppfylldes inte målet om att integrera AI direkt i applikationen. AI visade sig kunna vara ett användbart stöd vid informationsinhämtning, dokumentation och idéutveckling, men inte tillräckligt tillförlitligt för självständig testgenerering eller verifiering av protokollnära funktionalitet.

Sammanfattningsvis utgör testsystemet en praktiskt användbar grund för fortsatt automatisering. För att bli mer heltäckande behöver det byggas ut med fler protokoll, fler testscenarier och förbättrat användarstöd.

Litteraturförteckning

[1] Comstedt, Martin, "Arbeta agilt - vad innebär det?" 2021, [Online]. Tillgänglig: https://onbird.se/arbeta-agilt-vad-innebar-det/. Hämtad: 2 feb. 2026.

[2] Ungvarsky, Janine, "Scrum." 2023, [Online]. Tillgänglig: https://research.ebsco.com/plink/4a408451-d2e0-3f8a-bf63-21a8b7199046. Hämtad: 2 feb. 2026.

[3] Kte'pi, Bill, MA, "Python." 2026, [Online]. Tillgänglig: https://research.ebsco.com/plink/d530dcb5-2d9e-3e67-bdf1-18374ca48e49. Hämtad: 6 feb. 2026.

[4] , "Robot framework," 2026, [Online]. Tillgänglig: https://robotframework.org/. Hämtad: 3 mars. 2026.

[5] M. Tsitoara, *Git and GitHub Workflow.* Berkeley, CA: Apress, 2024, p. 11. [Online]. Available: https://doi.org/10.1007/979-8-8688-0215-7_19

[6] T. Wilmshurst, R. Toulson, and T. Spink, *Fast and Effective Embedded Systems Design - From Bits and Bytes to IoT, with the Arm Mbed (3rd Edition).* Elsevier, 2025. [Online]. Available: https://app.knovel.com/hotlink/toc/id:kpFEESDFI1/fast-effective-embedded/fast-effective-embedded

[7] PiiGAB, "Intern utbildning," PiiGAB, Tech. Rep., Feb. 2026, opubl. material.

[8] PiiGAB, "M-bus generellt," M-Bus generellt – Support, [Online]. Tillgänglig: https://www.piigab.com/support/m-bus-generellt/. Hämtad: 2 feb. 2026.

[9] "Meter-bus," Wikipedia, The Free Encyclopedia, [Online]. Tillgänglig: https://en.wikipedia.org/wiki/Meter-Bus. Hämtad: 4 feb. 2026.

[10] M-Bus, "M-bus dokumentation," [Online]. Tillgänglig: https://m-bus.com/documentation. Hämtad: 4 feb. 2026.

[11] European Committee for Standardization (CEN), "EN 13757-2:2018 Communication systems for meters – Part 2: Physical and link layer," European Committee for Standardization, Brussels, Belgium, European Standard, 2018.

[12] ——, “EN 13757-2:2018 Communication systems for meters – Part 3: Application layer,” European Committee for Standardization, Brussels, Belgium, European Standard, 2018.

[13] E. Aykut, K. Erdogan, S. N. Benli, M. C. Mumcu, and I. Yavuz, “Remote control and monitoring protocols for plc and hmi systems: A case study of modbus, wmi and vnc,” in *2025 9th International Symposium on Innovative Approaches in Smart Technologies (ISAS)*, 2025, pp. 1–6.

[14] PiiGAB, “Gateways och omvandlare,” [Online]. Tillgänglig: https://www.piigab.com/products/m-bus-omvandlare/. Hämtad: 4 feb. 2026.

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

CHALMERS