

**STUDENT INDUSTRIAL PROJECT
REPORT**

**Implementation of a Rust-Built Soft PLC on Real-Time
Linux with EtherCAT, Modbus/TCP, and
OPC UA Integration for IIoT Capability**

JANUARY 2025 – AUGUST 2025

BBS AUTOMATION PENANG SDN BHD


VOLINTINE ANDER JILOH JR.

21001524

ELECTRICAL & ELECTRONICS ENGINEERING

VERIFICATON STATEMENT

I hereby verify that this report was written by
VOLINTINE ANDER JILOH JR. (Student's Name)
and all information regarding this company and the projects involved are NOT
CONFIDENTIAL / ~~CONFIDENTIAL~~ (strikethrough not relevant).

Host Company Supervisor's Signature & Stamp	
Name:	CHUAN TEONG KHOO
Designation:	HOD ELECTRICAL DESIGN & SYSTEM CONTROL
Host Company's:	BBS AUTOMATION PENANG SDN BHD
Date:	1 Aug 2015

Abstract

This project addresses the feasibility of implementing a commercial-grade Building Automation System (BAS) using a fully Free and Open Source Software (FOSS) stack, in response to the growing demand for cost-effective, transparent, vendor-agnostic, and interoperable automation solutions. The adoption of FOSS in commercial-grade automation remains limited, with key challenges encompassing reliability concerns, security, and standards compliance. The goal of the project is to evaluate the feasibility of utilizing FOSS in commercial-grade automation by building a functional prototype mock BAS. This was achieved by implementing a soft PLC running on real-time Linux (PREEMPT_RT) acting as an EtherCAT MainDevice and Modbus/TCP client for interfacing with remote I/O. The soft PLC is hosted on a Raspberry Pi 5 2GB. The reliability of the deployed system was quantified by measuring jitter, latency, uptime, system resource utilization, and CPU temperature. Real-time performance of the soft PLC under continuous heavy stress load over 48 hours resulted in maximum jitter measurement of 210 μ s, with CPU temperature hovering around 60°C. Typical maximum jitters measured vary from 22-210 μ s from no stress load to maximum stress load. Testing of the Industrial Internet of Things (IIoT) functionality of the integrated system was performed qualitatively, posing no impact on real-time performance as maximum jitter recorded stayed <2% of the 10ms cycle time as the maximum jitter recorded was 35 μ s, with 100% of the API responses having returned status code 200 (OK) throughout testing.

Acknowledgements

I owe gratitude to my host company supervisor, Chuah Teong Khoey, for giving me the inspiration to explore this topic, and for lending the Beckhoff EK1100 and BK1120 couplers, as well as the corresponding terminals and EnOcean master and transceiver units. Without access to the necessary hardware, this project would have been impossible. The support and affirmation I have received from him has also proven indispensable in my effort to make the most out of the learning opportunity and my time at BBS Automation Penang.

I would like to thank the global open source community who are the oft-unsung heroes behind the dependencies used in the project. Specifically, I would love to express great gratitude to James Waples for ethercrab, an EtherCAT MainDevice implementation in Rust; Philipp Schmechel of QiTech GmbH for the codebase I used as a starting point; Jeff Anderson for GatorCAT, an EtherCAT MainDevice implementation in Zig; Einar Omang of async-opcua, a fully async implementation of OPC UA in Rust; Umberto Nocelli of frangoteam for FUXA, a FOSS HMI/SCADA platform, and Mathias Kraus, Christian Eltzchig, and Jeff Ithier of ekxide IO GmbH for iceoryx2, a zero-copy lock-free ultra-low latency inter-process communication middleware. I would also like to thank Davy Demeyer of Acceleer for extending a warm invite to the SASE Slack.

To the countless other amazing people who have directly and indirectly contributed to open source code and documentation, I have learned so much thanks to their efforts, and nothing can repay the debt that I owe but to give my own learnings back to the community. Despite being in disparate timezones and each thousands of miles away, they have never failed in giving me assistance, encouragement, and most importantly a genuine sense of community.

List of Abbreviations

ANSI	American National Standards Institute
API	Application Programmer Interface
ASIC	Application-Specific Integrated Circuit
BAS	Building Automation System
DCN	Distributed Control Node
DIN	<i>Deutsches Institut für Normung</i>
DMA	Direct Memory Access
EEPROM	Electrically Erasable Programmable Read-only Memory
ERP	Enterprise Resource Planning
ESC	EtherCAT SubDevice Controller
ESI	EtherCAT SubDevice Information
ESM	EtherCAT State Machine
ETG	EtherCAT Technology Group
FOSS	Free and Open Source Software
FPGA	Field Programmable Gate Array
FSoE	Functional Safety over EtherCAT (Failsafe over EtherCAT)
CoE	CANOpen over EtherCAT
CRUD	Create, Read, Update, Delete
GPOS	General Purpose Operating System
GNU	GNU's Not Unix!
GUI	Graphical User Interface
HMI	Human-Machine Interface
HTTPS	Hypertext Transfer Protocol Secure
IIoT	Industrial Internet of Things
IDE	Integrated Development Environment
IDF	Integrated Digital Factory
IEC	International Electrotechnical Commission
IP	Internet Protocol
IPC	Inter-Process Communication / Industrial PC
IR 4.0	Industrial Revolution 4.0
LAN	Local Area Network
LLVM	Low Level Virtual Machine

LLVM IR	LLVM Intermediate Representation
MES	Manufacturing Execution System
NIC	Network Interface Controller
OOP	Object-Oriented Programming
OPA	Open Process Automation
OPC UA	Open Platforms Communications Unified Architecture
PDO	Process Data Object
PLC	Programmable Logic Controller
RAII	Resource Acquisition is Initialization
REST	Representational State Transfer
RTOS	Real-Time Operating System
SCADA	Supervisory Control and Data Acquisition
SCP	Secure Copy
SDO	Service Data Object
SIL	Safety Integrity Level
SSH	Secure Shell
TCP	Transmission Control Protocol
UB	Undefined Behavior
UI	User Interface
UL	Underwriters' Laboratory
WAN	Wide Area Network

Table of Contents

	Verification Statement	i
	Abstract	ii
	Acknowledgements	iii
	List of Abbreviations	iv-v
1.0	Introduction	1-2
1.1	Background	1
1.2	Problem Statement	1
1.3	Scope of Work	2
2.0	Objectives	3
3.0	Literature Review	4
3.1	Real-Time Computing	4-5
3.2	Soft, Virtual, and Hard PLCs	5-7
3.3	Real-Time Communications	7-8
3.4	Preemptive Real-Time Multitasking	9
3.5	Memory Safety	9-10
4.0	Methodology	11-32
4.1	Methods and Tools	11
4.1.1	Programming Languages	11
4.1.2	Graphics	11
4.2	Project Activities	13-31
4.2.1	Hardware Setup	12-14
4.2.2	Development Toolchain Setup	14-16
4.2.3	System Architecture – Hardware and Software Stack	17-18
4.2.4	Implementation Details	19-22
4.2.5	Test Methodology	23-31
4.3	Gantt Chart and Milestones	32
5.0	Results and Discussion	33-50
5.1	Results	33-40
5.1.1	<i>cyclictest</i> Latency Results	34
5.1.2	Worst-Case 48-Hour Soak Test	34
5.1.3	12-Hour Soak Test	34-35
5.1.4	Proxy Soak Test	35

5.1.5	Light Load Tests	36
5.1.6	Floating above Idle Test	37
5.1.7	12-Hour Test: Floating above Idle	37
5.1.8	Testing of Heuristic: 12-Hour Light Soak Test	38
5.1.9	Jitter Distribution	38-39
5.1.10	Real-Time Performance Test Caveats	39
5.1.11	IIoT Integration Test	40
5.2	Discussion	41-50
5.2.1	Sub-Millisecond Real-Time Performance	42
5.2.2	Comparison with Equivalent Benchmarks	43
5.2.3	Functional Safety	44
5.2.4	The Pi vs. Other Host Hardware	45
5.2.5	Improving Operational Security	45
5.2.6	Extending Support for IEC 61131-3	46
5.2.7	Lock-Free Design	46
5.2.8	Relevance to Open Process Automation (OPA)	47
5.3	Sustainability	48-50
5.3.1	Environmental	48-49
5.3.2	Economic	49
5.3.3	Social	50
6.0	Conclusion	51
	References and Citations	52-56
	Appendix A: Links to Project GitHub Repositories	57
	Appendix B: Screenshots of Parts of the System	58-64
	Appendix C: Pictures of the Physical Test Bench	65-66
	Appendix D: Linux Kernel Build and Boot Configuration	67
	Appendix E: Equipment Bill of Materials	68

1.0 Introduction

1.1 Background

The Fourth Industrial Revolution (IR 4.0) has ushered in an era of unprecedented connectivity and automation, driven by advancements in the Industrial Internet of Things (IIoT). Traditional automation systems often rely on closed-source proprietary software and hardware, which are often unmodular, costly, inflexible, and have poor interoperability. However, with the rise of Free and Open Source Software (FOSS), there now exists a compelling alternative. FOSS offers the potential for reduced costs, increased customization, and greater transparency. This study aims to explore the feasibility of using FOSS in commercial-grade automation by developing a mock Building Automation System (BAS) using open-source tools and technologies.

1.2 Problem Statement

Despite the potential benefits of FOSS, its adoption in commercial-grade automation remains limited. Key challenges include concerns about reliability, security, integration with existing systems, and compliance with common standards. This project seeks to address these challenges by evaluating the performance, reliability, and security of a mock BAS built entirely with FOSS. The goal is to determine whether FOSS can meet the stringent requirements of commercial-grade automation and provide a viable alternative to proprietary solutions.

1.3 Scope of Work

This project seeks to establish the degree of feasibility of utilizing a fully open-source stack in building and deploying a commercial-grade automation system. In this report, a distinction is made between commercial-grade and hobbyist-grade (non-commercial) systems. Commercial-grade automation adheres to industry norms, expectations, and standards at a level reasonable enough for commercial customers; whether that would be industrial process automation, or building automation. Hobbyist-grade automation is distinctive in its use of ultra-low-cost hardware, lower power electronics, voltages lower than 24Vdc. In contrast, this project considers a setup consisting of hardware targeted at industrial applications to be ‘commercial-grade’. The scope is delineated more explicitly as below:

- i. This project does not control nor automate any physically-actuated systems, as such systems typically require functional safety, which albeit possible on the black channel, is currently unimplemented by the EtherCAT MainDevice used.
- ii. This project does not control nor automate any systems involved in the preservation of life, such as but not limited to smoke alarms, gas leak detectors, and fire sprinkler systems.
- iii. All electronics with the exception of the input side of the AC-DC power supply, run on 24Vdc or lower, with the maximum current in any part of the system limited to 1A, 5A, or 10A by fuses.
- iv. ‘Deployment’ of the system refers to execution of compiled program binaries on a Raspberry Pi 5, and the execution of apps/services on the development laptop to host the HMI/SCADA, and optionally, web services running on remote servers for IIoT functionality.

2.0 Objectives

This project aims to evaluate the feasibility of a fully open-source based commercial-grade automation system by emulating a building automation system. The project's main objectives are summarized below:

- i. To create a working prototype of a building automation system (BAS) using a fully Free and Open Source (FOSS) stack to demonstrate feasibility of using open source tools.
- ii. To demonstrate HMI/SCADA functionality and IIoT integration (e.g. alarms and events relayed by RESTful APIs for client mobile apps).
- iii. To quantify deployment reliability via metrics such as determinism, jitter, latency, uptime, resource utilization and CPU temperature.
- iv. To evaluate the extent of possible future expansion into more critical applications, such as (but not limited to) process automation involving motion controls; functional safety; high throughput, low-latency data acquisition, and deterministic plant-level orchestration.

3.0 Literature Review

Following latest guidelines from Item 1.5 of the *EtherCAT Technologi Group* FAQ (2025) all instances henceforth of MainDevice and SubDevice respectively have the same meaning as ‘Master’ and ‘Slave’ in pre-existing materials.

3.1 Real-Time Computing

“Real-time” is used in many different colloquial senses, and is often conflated with “live”. Real-time is defined by the amount of guarantee possible for meeting the deadlines of specific tasks, backed by the degree to which system behavior is deterministic (Rostedt, 2016). The phrase is further subdivided into three classifications: hard, firm, and soft (Brown & Martin, 2010; VanderLeest, 2022; Intel Corporation, 2022). Failure to accomplish a task within a specified deadline can mean the loss of life or great financial loss.

A major example of hard real-time is in aerospace systems, where precious cargo or data may be lost if certain critical tasks could not be accomplished within the deadline allocated. Additionally, a major differentiator between hard real-time and firm or soft real-time is the formal verification required in order to obtain mathematical certainty, where system behavior is modelled mathematically hence formally verified. Such formal verification allows for quantifiable metrics to be established (e.g., known values for latency bounds etc.) (Feo-Arenis et al., 2016). Firm real-time sits between hard and soft real-time, where there is a reasonable expectation of failure that does not have a particularly catastrophic result. Missed deadlines in a firm real-time system does not constitute total unrecoverable failure. On the other hand, soft real-time systems can tolerate missed deadlines which although undesirable, pose no risk or threat to safety (Utande, 2025).

Real-time performance can be characterized by latency and jitter. The exact definition of these quantities depend on context, but latency and jitter are generally defined as follows. In the context of the Linux scheduler, latency can be defined as the time elapsed after the invocation of a task until the beginning of the task execution. (Huang & Yang, 2017). Hence a general definition of latency may be construed as the amount of time elapsed between when an event is ordered to happen until the time that

the event actually starts to happen. The ‘event’ being measured varies by context, hence, the exact quantity being measured also varies.

Jitter may be defined as the deviation of the cycle time for a task. In the context of a PLC cyclic task, it may be further divided into ‘periodic jitter’ and ‘release jitter’. Following the definitions set in *Definitions of Jitter and Latency* (CODESYS Group, 2024), periodic jitter is simply how much the actual cycle time deviates from the desired cycle time. Whereas release jitter is the difference between the largest and smallest recorded latency that occurred within the measurement duration.

3.2 Soft, Virtual, and Hard PLCs

The terms “soft PLC”, “virtual PLC”, and “hard PLC” are being increasingly used in reference to paradigm shifts in how PLCs are implemented. Before the advent of PLCs, system controls were implemented with electromechanical relay logic (Wayand, 2020). The first PLC was invented in 1969 by Bedford Associates in response to a proposals request from General Motors, to replace relay-based control systems in the GM Hydramatic automatic transmission with electronic controls. The idea itself was first envisaged by Dick Morley. (Dunn, 2009). Traditional ‘hard’ PLCs are relatively architecturally simple microcontroller-based systems built to withstand industrial environments. PLC programs in traditional PLCs sit closer to metal with a simpler abstraction layer to interface with hardware.

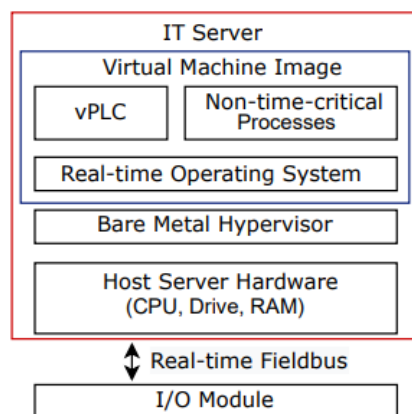


Figure 3.2-1(a): Example implementation architecture of a virtual PLC. Adapted from Perez et al.

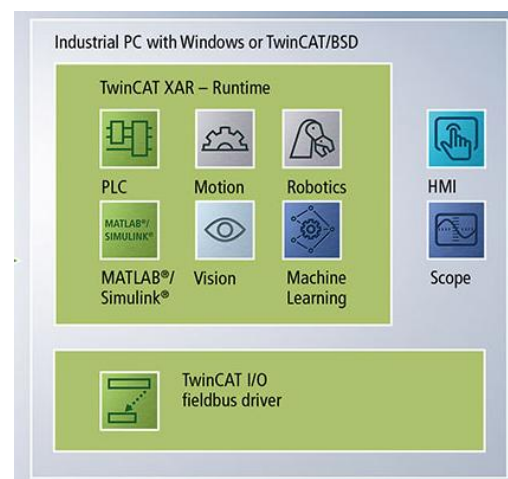


Figure 3.2-1(b): Beckhoff TwinCAT Runtime Architecture. Adapted from Beckhoff Automation (n.d.-e).

‘Soft’ PLCs on the other hand are characterized by a more complex architecture, with more layers of abstractions away from bare metal and relies on an operating system to interface with the hardware. Hence, the *programmable logic* itself is implemented purely as software hosted by an operating system. Perez et al. (2022) distinguishes soft PLCs and virtual PLCs respectively as a software that emulates a standard PLC for the former, and a soft PLC running in a virtual machine for the latter. “PC-based” controllers fall within the definition of soft PLCs, first pioneered by Beckhoff in 1986 (Beckhoff Automation, 2021).

Notable vendors of PC-based controllers are Beckhoff, B&R, and Rexroth. These vendors offer their own host hardware for their soft PLC runtimes. However, one vendor, CODESYS, exclusively focuses on the soft PLC runtime software itself and does not make any specific host hardware. CODESYS runtimes can run on any host hardware that runs Windows, Linux, or VxWorks (CODESYS Group, n.d.-b).

Since soft PLCs are essentially full computers, the low-level software and firmware are specialized to guarantee real-time. Real-time is defined by known bounded worst-case latencies which guarantee that control tasks can be executed whenever necessary in order to meet deadlines. This can only be satisfied by some means of system resource control to ensure timely execution of real-time control tasks. The approach taken by TwinCAT on Windows NT is to have a separate kernel extension that can safely override the behavior of the Windows NT kernel so that real-time tasks always have access to system resources.

Despite itself being not real-time capable, the hybrid architecture of Windows NT allows for this setup, thus enabling real-time (Beckhoff Automation, n.d.-a). The now-discontinued Windows CE on the other hand, is a preemptible kernel, and is readily soft real-time capable. Real-time applications on Windows CE utilize kernel preemption in order to grant access to system resources for real-time tasks (Microsoft, 2006). This is a similar approach with Linux PREEMPT_RT. Being a monolithic kernel, Linux strictly segregates user-space and kernel-space processes. In order to support real-time, the kernel must be built with PREEMPT_RT and enabled. This allows high-priority user-space processes to preempt the kernel itself, guaranteeing access to system resources for real-time tasks (McKenney, 2005; Brown & Martin, 2010; Madden, 2019; *Understanding Linux Real-Time*, 2025).

‘Virtual’ PLCs take the abstraction in a radical direction by completely liberating the control logic from a dedicated host hardware that is traditionally installed at specific parts of the production line. Instead, the PLC itself is merely software running in a container or a virtual machine, whose host hardware may be significantly further from the physical processes it controls. Although radical, there are pioneering examples of the architecture being deployed in production. For example, in a recent project by Audi, the TÜV-certified SIMATIC S7-1500V virtual PLC was used to control the axle assembly line for the Audi e-tron GT in Audi’s Böllinger Höfe factory in Neckarsulm (Siemens, n.d.).

3.3 Real-Time Communications

Real-time communications is achieved via EtherCAT, a summation-frame fieldbus protocol based on Ethernet specially designed for low latency and jitter, which are required in precise, time-sensitive applications (Wu & Xie, 2019). Originally developed by Beckhoff, the protocol has been standardized by the International Electrotechnical Commission (IEC) as IEC 61158. The specification is now maintained by the ETG as an open technology (*EtherCAT Technology Group FAQ*, 2025).

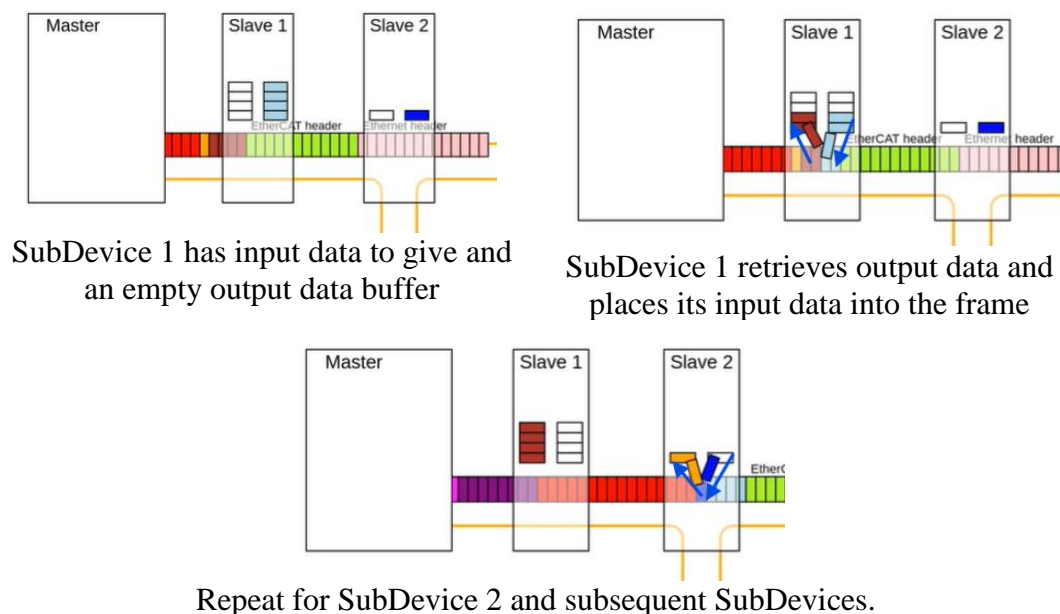


Figure 3.3-1: Visualization of on-the-fly processing. Adapted from Broling (2007).

Time determinism is achieved by utilizing FPGA/ASICs for on-the-fly processing of telegrams. As a telegram passes through a SubDevice, the ESC inserts input data into

the telegram, and retrieves (if any) output data from the telegram. The on-the-fly processing allows the entire network to be addressed with just a single frame. An EtherCAT SubDevice is governed by the EtherCAT State Machine. This state machine is composed of the states Init (INIT), Pre-Operational (PRE-OP), Safe-Operational (SAFE-OP), Operational (OP), and Boot (BOOT) (Beckhoff Automation, n.d.-f).

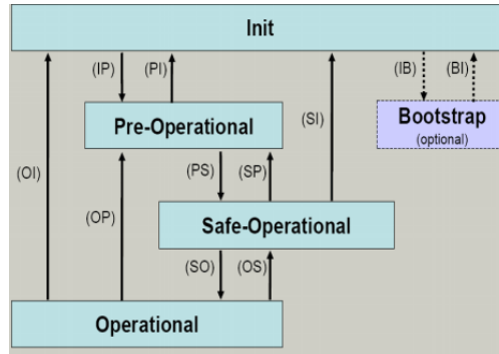


Figure 3.3-2: EtherCAT State Machine. Adapted from Beckhoff Automation (n.d.-f).

As the protocol relies on frame summation, the real-time performance of any specific network is characterized by the slowest SubDevice. Each SubDevice incurs additional processing delays, and this necessitates the use of FPGA/ASICs to implement the ESC for hard real-time guarantees and minimum latencies. Data is transmitted in two forms in EtherCAT: as PDOs (Process Data Objects) or SDOs (Service Data Objects). PDOs are used to transmit cyclic process data during the OP state, whereas SDOs are often used to parameterize a SubDevice during acyclic communication (during PRE-OP). The most common way SubDevices are parameterized is via CoE (CANOpen over EtherCAT) SDOs, among other protocols.

The address space of an EtherCAT network is 16 bits. EtherCAT also supports redundant cabling (ring topology) and cable breakage detection, though these features are optional and are defined in the specifications as feature packs (FP). In any EtherCAT network, there exists only one MainDevice, which handles SubDevice management, distributed clocking, and inter-SubDevice communication (which enables FSoE) among other functions. The bulk of the processing occurs within each SubDevice themselves, thus the MainDevice itself can be simply implemented in software so long as an Ethernet NIC exists (*“Industrial Communication Networks - Fieldbus Specifications - Part I”*, 2023).

3.4 Preemptive Real-Time Multitasking

By default, the Linux kernel employs preemptive multitasking, where tasks are preemptible. Preemption is the interruption of the execution of a task by a scheduler, based on the scheduling policy. This allows system resources to be distributed according to the priority of tasks and whether or not they are CPU-bound (mainly requires CPU resources) or I/O-bound (blocks on waiting for I/O) (Baeldung, 2023). PREEMPT_RT enables the application of preemptive multitasking in a real-time setting. However, the implementation is far from trivial.

When a low-priority task holds a lock on a resource that is needed by a high-priority task, a problem known as priority inversion may occur. When a medium-priority task preempts the low-priority task and holds the lock on the resource needed by the high-priority task, the high-priority task is starved and is effectively being preempted by lower-priority tasks, despite having the higher priority. This is solved in the Linux kernel using priority inheritance. The low-priority task is temporarily boosted to the same level as the high-priority task (thus inheriting the priority of the high-priority task) so that when it yields, the original high-priority task is able to immediately acquire the lock to the shared resource (*RT-mutex Subsystem*, n.d.).

3.5 Memory Safety

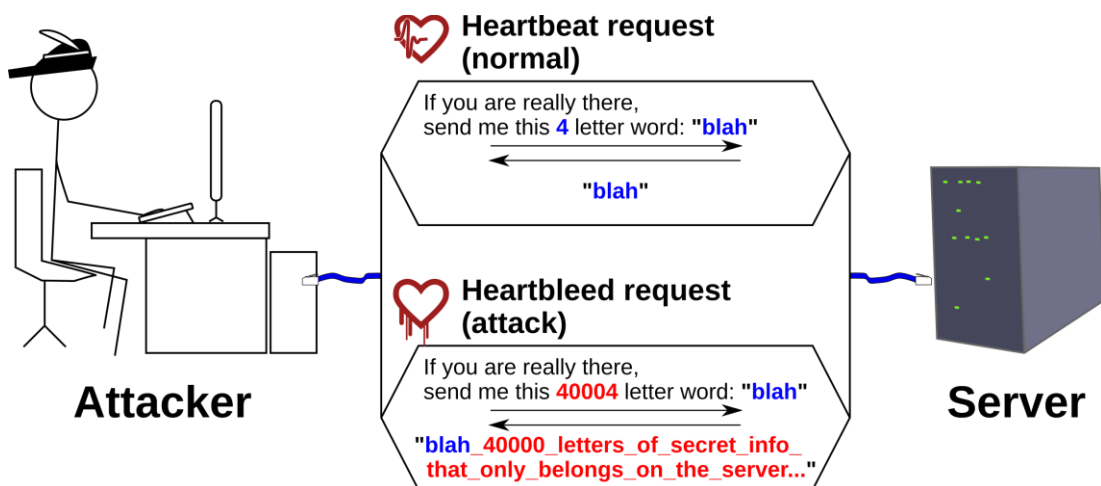


Figure 3.5-1: Illustration of the Heartbleed bug (CVE-2014-0160) caused by lack of input sanitization and bounds-checking. By Patrick87 (2014).

Many popular memory-safe languages achieve memory safety by using garbage collection (Heller, 2023). However, the presence of a garbage collector is undesirable

in many circumstances, such as real-time applications. Garbage collection is inherently indeterministic, and the programmer has limited control over its behavior. Programming languages that offer manual memory management exposes complete control over memory to the programmer. This is needed in applications that require deterministic guarantees, such as PLCs, communications, and embedded devices (Henriksson, 1998).

Traditionally, programming languages with manual memory management are often vulnerable to undefined behavior (UB). UB is simply program constructs that are left undefined by the programming language specification. Examples of UB (at least in C) are division by zero, oversized shifts, integer overflows, null pointer dereference, and reading uninitialized values (Wang et al., 2012). ISO/IEC 9899:2024 defines UB in C as “behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements”. Thus, UB in a program can result in the *program* doing absolutely anything, with no regards to whether this is expected or not by the programmer.

UB can manifest as a security vulnerability, such as in Heartbleed (CVE-2014-0160) as illustrated in Figure 3.5-1. In Heartbleed, the server program does not sanitize inputs from the client, and accesses data in memory beyond the correct bounds. Such data accesses allow malicious actors to arbitrarily read data stored in the server, which may be sensitive and confidential (Durumeric et al., 2014). Memory safety violations affect real-world products used daily. Microsoft and The Chromium Projects each have reported that memory safety related bugs constitute 70% of discovered security vulnerabilities in their respective products (Microsoft, 2019; The Chromium Projects, n.d.).

Rust is a notable exception, as it is currently the only mainstream programming language that is both memory safe while lacking a garbage collector. Rust code is ‘safe’ by default (UB is made impossible due to static checks by the compiler), but is still possible in regions marked with the `unsafe` keyword. However, Rust lacks an official specification such as ISO/IEC9899 for C. Hence, the claim that UB is completely eliminated in safe Rust is not airtight. Despite that, the static checks enforced by the compiler by default eliminates entire classes of memory bugs that are commonly and trivially introduced when writing in C (Ballo, Ballo, & James, 2022).

4.0 Methodology

Throughout the development of the project, the methodology most strongly adheres to the iterative and modular method, where the system is first broken up into simpler, smaller pieces that are eventually integrated. The process of iteration involves codebase refactoring, removal of redundancies, and eventually where necessary, growing in the quantity, interdependency, and complexity of features.

4.1 Methods and Tools

4.1.1 Programming Languages

The main programming language used is Rust, a high-performance, memory-safe programming language that enforces rigorous compile-time guarantees via a static type system and borrow checking, without the runtime overhead of a garbage collector. OOP in Rust is achieved via composition as opposed to inheritance with algebraic data types (specifically ‘algebraic’ to the extent that the type system forms a semiring up to isomorphism and inhabitation¹).

JavaScript is used for additional UI functionality within the FUXA HMI platform. Python is used for performance benchmarking analysis, as well as a gateway for IIoT functionality. Zig and C are used to implement a Telegram bot server. Bash scripting is used in the development and deployment environment assisting in deploying and running the compiled binaries. Dart is used to create the mobile client app, utilizing the Flutter UI framework. Among the programming languages used, C, Rust, and Zig, are not garbage-collected.

4.1.2 Graphics

Inkscape, an open source vector graphics editor was used to design the concept UI of the mobile IIoT client, in addition to graphical assets used for the project documentation. Draw.io was used to create the topology, architecture, and the EnOcean state machine diagram.

¹See Milewski (2015) for a discussion on category theory and algebraic data types.

4.2 Project Activities

This section describes the activities undertaken to realize the project objectives.

4.2.1 Hardware Setup

4.2.1(a) Electrical Connections

Wiring was done closely referencing datasheets provided by manufacturers. Additionally, all metal components were grounded to earth with oversized gauge wires. To avoid ground loops, there is only a single path to earth. For connection with the I/O components, ferrules were used. The IRIV IOC is not grounded as it does not have a dedicated earthing terminal. Additionally, at the time of writing, Cytron claims in its datasheet that the IRIV IOC does not require a connection to protective earth.

Per Beckhoff recommendations, the EK1100 and BK1120 couplers each are wired with 10A and 1A fuses. The 10A fuses protect the E-bus and K-bus terminal power contacts, and the 1A fuses protect the power supply for the EK1100 and BK1120 couplers themselves. The Raspberry Pi 5 was powered separately by a dedicated, 5A-fused AC-DC 5V UK power supply via USB-C. The Phoenix Contact FL SFNB 5TX network switch connects to earth via its metal DIN rail clip. While no effort is spent on creating a fully standards-compliant test bench, UL 508A recommends that all metal parts are earthed, and this is heeded wherever possible. The AC-DC power supply is fed via clip-on terminal blocks, and the hot (live) wire is connected through a breaker. This is especially important so that the system is truly de-energized and does not float live while “off”.

An alternative consumer-grade network switch, the TP-LINK LS1005 was also used, as the industrial-grade Phoenix Contact switch was on ephemeral loan. The LS1005 runs on 5V DC from the supplied, 5A-fused AC wall plug. Since the LS1005 is not earthed, the RJ45 cables float at 0Vdc (relative to 0Vdc of the power supply) for the IRIV IOC, and approximately -0.02V relative to earth for the RJ45 segment between the LS1005 and the UGREEN CR111 USB-A NIC. There should be no risk of ground loops in this configuration as this segment of the network is floating and is not earthed. The IRIV IOC is powered directly from the 24Vdc rail through a 10A fuse, with built-in surge and overcurrent protection.

4.2.1(b) Network Topology

Test Bench Topology

2025-07-17 T14:11 +08:00

Ander

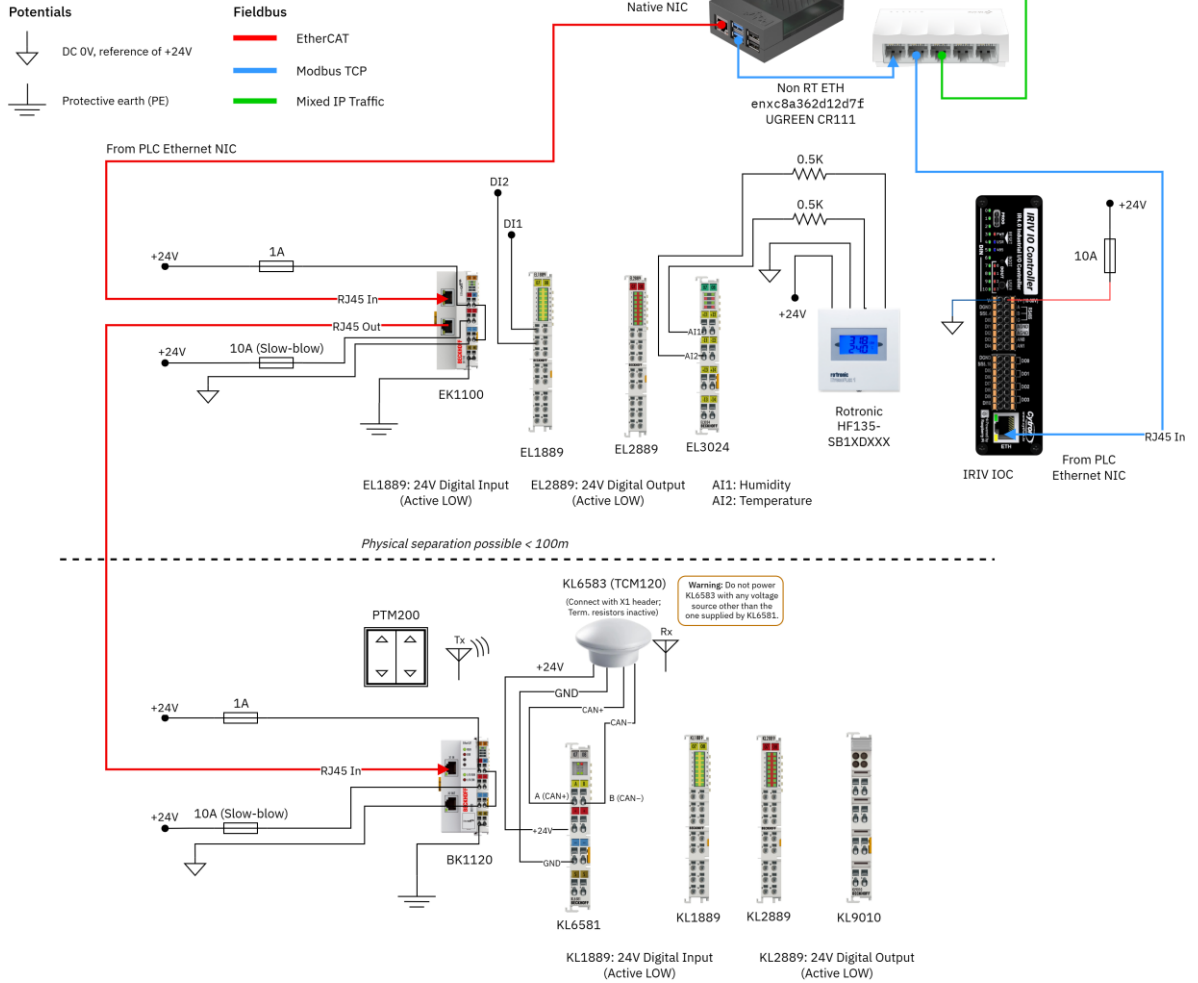


Figure 4.2.1(b)-1: Test bench network topology.

In Figure 4.2.1(b)-1, I/O terminal connections may arbitrarily be changed, but their coupler-relative positions are fixed. The EK1100 must be the first SubDevice in the EtherCAT network. The blue line only serves to clarify the path of Modbus/TCP packets though it piggybacks on the same Mixed IP network.

All wired communication is done via twisted pair Ethernet, with both ends terminated with RJ45. For programming of the IRIV IOC, the manufacturer-provided CircuitPython firmware is flashed via its USB-C port. Different NICs are used to transport different protocols and data. EtherCAT data is transported in its own dedicated network, whereas SCADA, Modbus, SSH, and other non-real-time data are transported via a separate USB-A NIC.

4.2.1(c) Networking Tools

Configuration and setup of the network involved extensive use of command line networking tools to verify and troubleshoot connections. Namely, the tools *arp-scan*, *nmcli*, *ping* in particular were absolutely critical.

4.2.1(d) Deployment Interfacing

The Raspberry Pi 5 is used in a headless (without dedicated display, mouse, and keyboard) configuration, and was likewise set up in the same manner. The Raspberry Pi OS image was flashed onto a 32GB microSD card using the Raspberry Pi Imager tool on the development laptop. Interfacing with the Pi is done via SSH, and files are transferred via SCP, both SSH and SCP have been set up to use certificate-based authentication, with password login disabled. The GUI service is disabled using *raspi-config* to dedicate maximum resources to the soft PLC runtime. Cross-compiled binaries are transferred to the Pi over SCP.

4.2.2 Development Toolchain Setup

4.2.2(a) Operating Systems

The only operating systems used for development are Pop!_OS 22.04 LTS and Raspberry Pi OS (Debian 12 Bookworm). The former is installed on the development laptop, while the latter on the Raspberry Pi 5 for deployment purposes. Both operating systems are Debian-based, with Pop!_OS in particular being based on Ubuntu.

The packaged kernel on the official Raspberry Pi OS image was not real-time capable, despite being version 6.12. Hence, the Linux kernel version 6.12 on the Raspberry Pi OS was recompiled with the PREEMPT_RT patch, to enable preemptive real-time multitasking, a critical requirement for soft PLCs. Additionally, in the spirit of open source, Windows is used only to prepare this report to comply with the academic convention of using Microsoft Word.

4.2.2(b) Integrated Development Environment (IDE)

Visual Studio Code (VSCode) was initially used early on in the project, but due to language server incompatibility and performance overhead, VSCodium is used instead. VSCodium is a community-distributed version of VSCode that is compiled solely from the open-source codebase, without additional closed-source add-ons from Microsoft. Cargo is also installed to handle dependencies, linkers, targets, and to conveniently interact with the Rust compiler.

The language server implementation for Rust, *rust-analyzer*, is installed as an extension, providing syntax highlighting, type checking, and documentation popups. Python is used with *venv* to provide reproducible virtual environments, eliminating issues with dependency management and multiple Python installation directories. *venv* is integrated with the VSCodium IDE, though *venv* itself is a runtime environment independent of any IDE. As for development of the mobile client app, the Dart and Flutter extensions were installed for ease of building and flashing the .apk to an Android smartphone, debugging, packaging, managing dependencies, syntax highlighting and refactoring.

Support for Zig in VSCodium was enabled by installing the Zig language extension, which also comes with the ZLS, the language server implementation for Zig. As is the case with *rust-analyzer*, ZLS provides syntax highlighting, type checking, and popups. However, unlike *rust-analyzer*, ZLS does not show all compiler errors. On save, *rust-analyzer* highlights all compiler errors.

4.2.2(c) Source Control

Source control is achieved using Git, the most ubiquitous, open-source, industry-leading distributed source control system. All source files for the core runtime are stored in a monorepo architecture, using GitHub to host the remote repository. Core Git features such as branches, merges, reflog, rebase, were all used at least once throughout development to assist with structured tracking of the source code. The source code for minor additional non-real-time applications (mobile app client, PLC OPC UA-Supabase gateway) are stored in separate repositories.

4.2.2(d) Cross-Compilation

The ARM toolchain is used to cross-compile for the Raspberry Pi 5's ARMv7 BCM2712 SoC. Initially during development, cross compilation was done directly on the host environment, i.e. without a container. However, with the goal of ensuring no undefined behavior and to increase performance, inter-process communication between the PLC and HMI/SCADA was reimplemented using *iceoryx2*, whose *clang* build scripts could not resolve the required ARM GNU C libraries.

Hence, the most convenient and reliable way to perform cross compilation was to use *cross*, a wrapper for Cargo with Docker as the backend. The containerized approach streamlines dependency management by separating the host environment and emulating the target more closely, thereby eliminating “it works on my PC”-type problems. In implementing the Telegram bot in Zig, however, CMake was unfortunately unavoidable due to a critical dependency, *iceoryx2*, not having official Zig bindings. Hence, its C bindings were used instead.

As the Telegram bot consumes PLC data via inter-process communication (using *iceoryx2*), the C bindings for *iceoryx2* still need to be cross-compiled using CMake, while the resulting Zig application was cross-compiled with Zig itself. Due to the *glibc* shared library dependency, the ARM compiler toolchain was version-matched with the installed *glibc* version on the Raspberry Pi.

4.2.3 System Architecture – Hardware and Software Stack

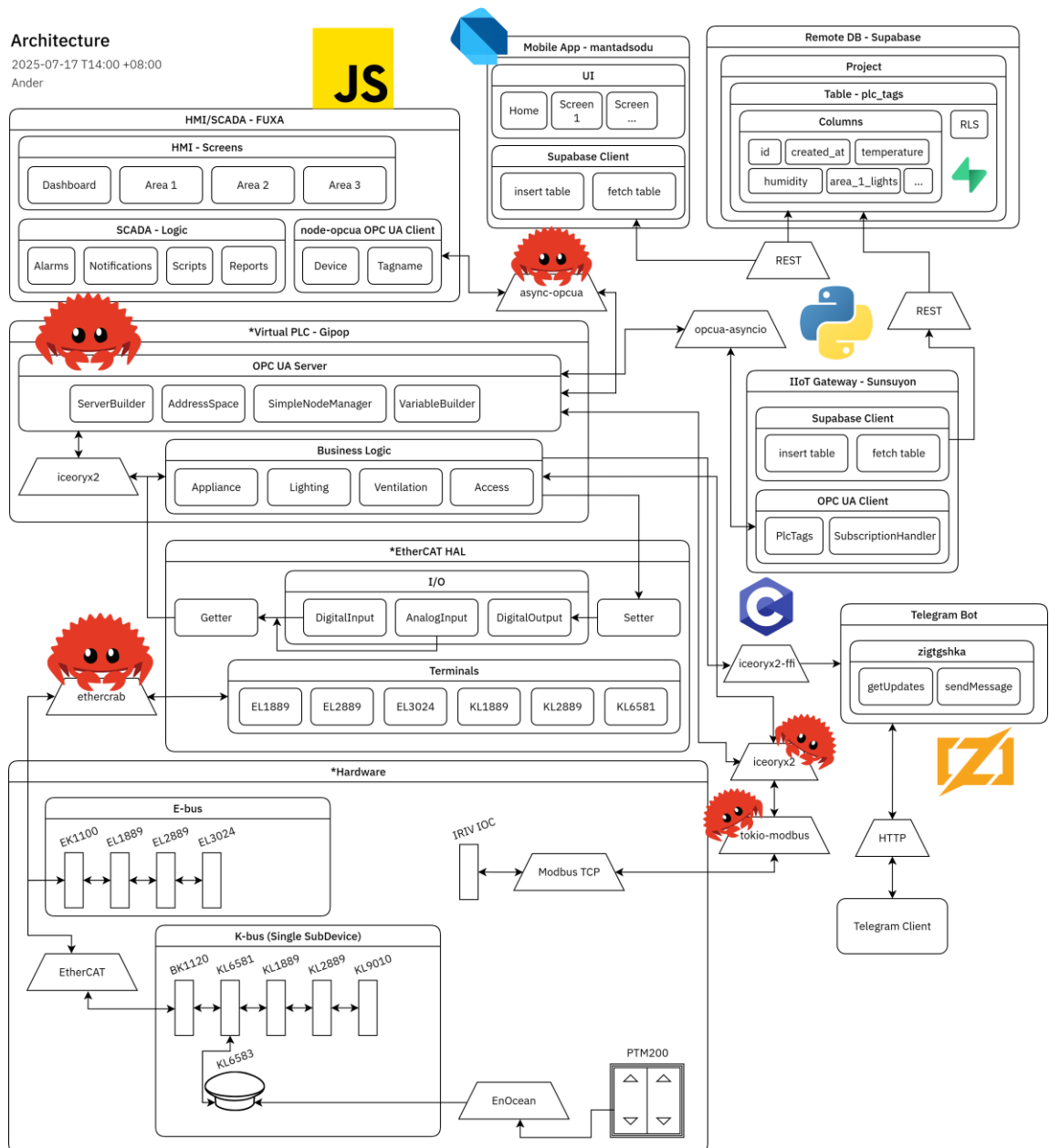


Figure 4.2.3-1: System software and hardware architecture. Parts marked with Ferris the Crab are written in Rust.

Architectural design decisions are motivated by ease of interpretability and experimentation. Major components of the software stack are not production-ready, though they may have been used in production. Maintainability and documentation quality is prioritized over performance in non-critical components. The control loop is implemented in Rust as it is the most critical part of the code requiring performance and deterministic behavior, whereas the non-critical IIoT gateway that links between the remote database and the OPC UA server is a simple Python script. Additional HMI

functionality is achieved using scripts written in JavaScript, while the mobile app client is fully written in Dart using the Flutter framework.

Not all architectural design decisions are motivated solely by technicalities; *ethercrab* was chosen as the EtherCAT MainDevice because of a small, tightly-knit group of about four dozen users that regularly and casually interact with the developer. The soft ‘PLC’ is hence built around the MainDevice, in Rust. The memory safety and fearless concurrency advantages that come with using Rust were an unintended bonus. In addition to its active status, it is also a performant user-space application, simplifying deployment and troubleshooting.

However, this specific decision does not shun other MainDevice implementations. GatorCAT, a Zig implementation of the EtherCAT MainDevice was heavily used as a close reference in understanding the EtherCAT standard itself, despite not being an inseparable dependency of the soft ‘PLC’. GatorCAT’s command line tool and use of Zenoh to publish/subscribe process data as topics makes it highly modular and architecturally elegant, with minimal overhead. Despite that, Zig is an unproven language still under development, and is far from version 1.0; though the language does have an active and wholesomely helpful community of users. Zig was used to implement the Telegram bot, using the C FFI bindings for *iceoryx2* to subscribe to data published by the PLC.

A standard feature of most modern PLCs is online variable read/writes. This is relatively straightforward to implement, as any standard debugger can be used to change variable values of a running process. However, ensuring that such changes are done safely is far from a trivial task. In addition to manipulation of variables at runtime, PLCs require online code changes (also known as hot swapping or hot reloading). That is, recompilation of parts of the control logic and swapping the new binary while the PLC program is running. This feature is difficult at face value, and even harder to implement safely and correctly.

Such features were not implemented due to the sheer complexity. A feature-complete runtime will require additional upstream dependencies. Additionally, there are many ways UB can manifest in such a system, which can have negative real-world consequences. such work is left for future exploration and are hence not within the scope of this project.

4.2.4 Implementation Details

4.2.4(a) Concurrency and Multithreading

tokio and *smol* are used for asynchronous programming. The EtherCAT MainDevice, *ethercrab*, performs best using a lightweight *smol* runtime, whereas *tokio* is used in the OPC UA server and the Modbus/TCP processes. Transport of real-time EtherCAT process data out of the sensitive hot path and into the OPC UA server leveraged inter-process communications, implemented with *iceoryx2*. The OPC UA server is a blocking await, so the IPC polling task is a separate thread.

Thread-safe access of shared data is implemented using *RwLocks* and *Mutexes*. These locks utilize *RAII* in order to release locks to shared data; however, they do not provide any runtime or static warnings for deadlocks, but a deadlock is immediately noticeable as the EtherCAT couplers will immediately go into PRE-OP due to timeout.

4.2.4(b) Pre-Faulting

Page fault counts increases within the first few seconds of program startup and remains constant afterwards. During program initialization at PRE-OP, the process image buffer of the EtherCAT network is configured. This buffer is allocated on the heap using *Vec<T, A>*, which is dynamically-expanding. This effectively (validly) faults the allocated pages needed. Since no additional shrinking/expansion of heap-allocated data structures occur past PRE-OP, the program has access to all of the heap memory it needs; hence page fault counts remain constant. For real-time applications, the added overhead from page faults can negatively affect real-time performance. Hence, allocated memory must be pre-faulted before critical sections.

One caveat of this is minor page faults due to IPC. Since the PLC process is also a subscriber to inter-process data, when publisher processes get initialized, a small number (usually 3) of minor page faults are generated by the PLC process. This is negligible and will not adversely effect real-time performance, as long as publisher processes to the PLC process do not get frequently spawned and killed. The following command can be used to monitor the minor and major page fault counts for a specific process:

```
$ ps -o min_flt,maj_flt [PID]
```

4.2.4(c) EnOcean Driver

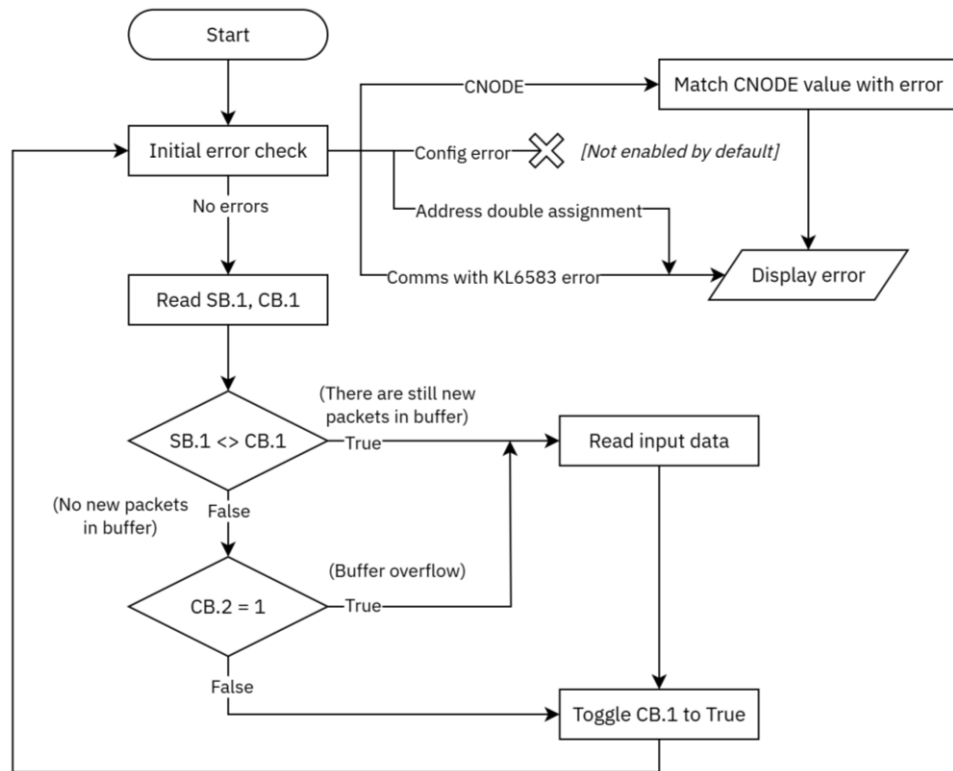


Figure 4.2.4(c)-1: KL6583 EnOcean State Machine.

Constructed based on *Documentation / EN KL6581 and KL6583* (2023) and *EnOcean Equipment Profiles* (2017).

The KL6583 and KL6581 form the transceiver unit that receives input data from the PTM200 switch. EnOcean is a batteryless, wireless protocol. The EnOcean driver was written as part of the PLC application, based on the datasheet of the Beckhoff KL6581/6583 (*Documentation / EN KL6581 and KL6583*, 2023) and the EnOcean Equipment Profiles document, version 2.6.7 published by the EnOcean Alliance (*EnOcean Equipment Profiles*, 2017).

The implementation is a simple state machine, though some trial and error was necessary as the KL6581/6583 datasheet contained confusing language and severe typos. In addition, example TwinCAT programs provided by Beckhoff depend on closed-source libraries, providing little insight to implementation.

4.2.4(d) IIoT Backend

Supabase is an open-source development platform for storage backends. Every project is a Postgres database, and the API is implemented using PostgREST. It can be self-hosted, hosted on a third-party cloud provider, or first-party hosted. For this project, the first-party hosted solution is used with the Free tier. The Free tier offers unlimited API requests, 50,000 monthly active users, a 500MB database size, 5GB bandwidth and 1GB file storage. For the purposes of this project, the Free tier is already more than enough. However, after seven days of inactivity, the Supabase project is automatically paused, and the number of active projects on Free tier is limited to two.

The Postgres tables used for the project are protected using Row Level Security (RLS) to prevent unauthorized read/writes. The source of truth at the plant level is the OPC UA server hosted on the PLC hardware (Raspberry Pi 5 for this project). The PLC is responsible for transforming fieldbus data into OPC UA tags, which are then used by HMI and SCADA. The Supabase database itself acts as a tag historian.

For urgent alerts, the PLC also hosts a Telegram bot. Predefined alert messages in the PLC program are communicated via IPC with the Telegram bot, bypassing the OPC UA server. It is assumed that IPC is more reliable than an OPC UA client connection, hence, critical alerts are triggered in this manner.

4.2.4(e) IIoT Frontend

The UI/UX layout of the mobile client app was initially designed in Inkscape. UI elements such as buttons and icons come from the Material design suite, with the exception of the typeface, Work Sans, which was sourced from Google Fonts. *fl_charts* was used to render line graphs and *icons_launcher* was used to handle app icon generation and configuration.

The mobile client app implements three features: Tag View, Chart View, and Admiral. Tag View simply shows a scrollable sample of the latest records in the database. Chart View simply plots the latest analog input data stored in the database. Admiral is a feature that allows the user to send predefined commands to the PLC remotely, however such commands are only executed if the remote command interlock is satisfied. In addition to the mobile client app, any Telegram client app acts as the frontend for the Telegram bot hosted on the PLC.

4.2.4(f) Debugging Methodology

A major bug relating to the EtherCAT MainDevice used was initially sidestepped. The bug involved an out of bounds array access to an array structure storing SyncManager types for a particular SubDevice; the BK1120 K-bus coupler. Due to the out of bounds access, the runtime panic prevented the ESM of the BK1120 to transition from PRE-OP to SAFE-OP. The bug was sidestepped by forcing an index check and breaking a specific loop early. The coupler was connected to a Windows laptop running TwinCAT 3 and the startup CoE parameters were inspected.

Other users of a separate MainDevice, the IgH EtherCAT Master reported previously ESM state transition failures due to a firmware bug on certain units. Ultimately, it was finally discovered that the MainDevice read two additional Sync Managers that were completely undocumented. This was cross-referenced with the BK1120 datasheet itself as well as the TwinCAT ESI file. The datasheet also did not document specific CoE parameters necessary for transition into SAFE-OP and OP. The bug was fixed by removing discovered Sync Managers that did not correspond to SyncManager types specified in the EtherCAT standard.

4.2.5 Test Methodology

4.2.5(a) Real-Time Performance Testing

All CPU cores are pinned to ‘performance’ to disable frequency scheduling. The command-line tools used for testing are *cyclicttest* and *stress-ng*. The latter is used to add stress to the system while the soft PLC is running. *cyclicttest* is used to gauge the baseline real-time performance of the system by measuring thread latencies. This is particularly useful to quickly verify system real-time performance optimizations. In addition, *stress-ng* may be used with *taskset* in order to specifically stress a specific CPU core. During test runs, *htop* is also used to monitor system resource utilization.

The following *cyclicttest* command was used to specify exact test run parameters:

```
$ sudo cyclicttest -D 10m -i 200u --default-system -a 0-3 -t 8 --  
mainaffinity 2-3 -p 95 --policy fifo --mlockall --priospread
```

Since CPU cores 2 and 3 were isolated for executing real-time tasks, *stress-ng* was run on the isolated CPU cores using *taskset*:

```
$ taskset -c 2 stress-ng --cpu 8 --cpu-method fft  
$ taskset -c 3 stress-ng --cpu 8 --cpu-method fft
```

stress-ng was also used to stress virtual memory and disk I/O as below:

```
$ stress-ng --vm 8 --vm-bytes 900M --timeout 20m --iomix 8
```

4.2.5(b) Jitter Distribution Analysis

Real-time performance relies on the maximum jitter recorded. However, it is still useful to analyze the distribution of recorded jitter values. These test runs are short, ranging from 5-20 minutes, hence they cannot be used to gauge the long-term real-time performance of the PLC. The PLC measures the jitter between each cycle, appends it to a Vec<T, A> type that is then written out to a .csv after a specified amount of time. The .csv is then copied over into the development laptop where it is analyzed with Python to compute the mean, standard deviation, as well as a histogram plot.

4.2.5(c) Integration Testing Methodology

In order to test the fully integrated system as a whole, the mock BAS application is created to demonstrate the maximum extent of functionality for each part of the system, as well as their interactions. This was conducted with the Pi 5 connected to the internet over WiFi. The PLC logic is programmed to conditionally trigger events involving full-duplex transmission and processing of data between all nodes. The nodes in the integrated system are as follows:

Node	Function	Actual Host/Platform
PLC	Run EtherCAT MainDevice, Modbus/TCP client, and control logic	Raspberry Pi 5B
HMI	Host FUXA UI to visualize and read/write I/O data	Acer Aspire A315-57G
Remote Database	Store fieldbus I/O history and commands from Mobile Client	Supabase on AWS t4g.nano in Singapore
Mobile Client	Mobile HMI, send remote commands	Samsung SM-A528B/DS

Table 4.2.5(c)-1: Nodes in Integrated System.

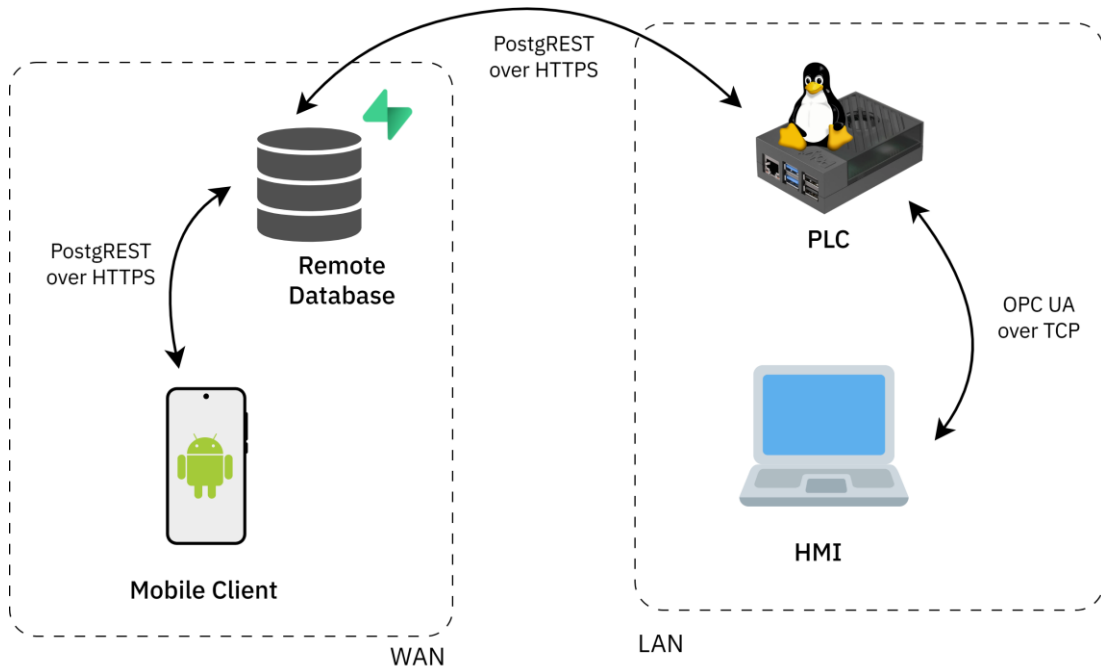


Figure 4.2.5(c)-1: Visualization of Nodes in the Integrated System.

In Figure 4.2.5(c)-1, “LAN” refers to the local TCP/IP network, which excludes the EtherCAT fieldbus, but not Modbus/TCP. However, data from Modbus/TCP is not directly accessed by nodes other than the PLC. The source of truth at the floor level is the OPC UA server, which the PLC hosts.

The integration test is fully qualitative. As the overall integrated system is not fully real-time (with the exception of the PLC logic and the EtherCAT fieldbus), temporal measurements were not measured as the WAN and LAN are inherently non-deterministic (all data transmitted over TCP, internet connection provided through WiFi, etc.), hence are unbounded. Latency between nodes in the integrated system also depends on the specific internet connection, service provider, time of day, etc. Therefore, no useful interpretation of temporal measurements can be made with respect to the integrated system.

4.2.5(d) Mock BAS Control Narrative

The mock BAS controls a fictional building composed of three arbitrary areas, presumably within 100m of each other. One can imagine them to be bedrooms, meeting rooms, garages, or a yard. Nevertheless, the effect of inputs on outputs are not limited based on the ‘area’ where the input device is located. The three arbitrary areas are served by three separate I/O devices: The E-bus terminals (EK1100 coupler cards); the K-bus terminals (BK1120 coupler cards), and the IRIV IO Controller. All fieldbus communication occurs over twisted pair Ethernet.

Input Device	Type	Channels	Area Served	Active Level	Protocol
KL1889	Digital	Channel 6	1	LOW	EtherCAT
KL6581 +KL6583	Digital (or Analog, data scheme is arbitrary)	Rocker A Rocker B	All	-	
EL1889	Digital	Channel 1-2	2	LOW	
EL3024	Analog, 4-20mA	Channel 1	2	-	
IRIV IO	Digital+Analog (0- 10V or 4-20mA)	AN0 DI0	3	HIGH	Modbus /TCP

Table 4.2.5(d)-1: Mock BAS fieldbus inputs.

The KL6581+KL6583 combo constitutes the EnOcean transceiver unit that receives digital data wirelessly from the batteryless PTM200 transmitter. Since the transceiver receives EnOcean datagrams and exposes the raw datagram in its process image, the form of the data (whether it is digital/analog) depends on the transmitter. In this case, the PTM200 is a simple digital switch.

Output Device	Type	Channels	Area Served	Active Level	Protocol
EL2889	Digital	Channels 1-16	1	LOW	EtherCAT
KL2889	Digital	Channels 1-16	2		

Table 4.2.5(d)-2: Mock BAS fieldbus outputs.

Transducer	Type	Type
PTM200	EnOcean Pushbutton Transmitter Module	Input
Limit Switch	Switch	
Selector Switch	Switch	
E-stop Switch	E-stop	
Rotronic Hygroflex HF135-SB1XDXXX	Humidity and Temperature Sensor	
Q-Light ST45B-3-24-RAG	Tower Lights	Output

Table 4.2.5(d)-3: Mock BAS transducers.

Alarms and logs reside on the HMI/SCADA running on a separate computer. The PLC can also send notifications via a Telegram bot. There are two HMIs. The local HMI reads and writes tag values to/from the PLC via OPC UA over the LAN. The remote HMI is a mobile app reads and writes tag values to/from the PLC via a remote database. The remote HMI writes tag values using its ‘Admiral’ feature.

The following flow charts describe the PLC logic.

EnOcean Routine

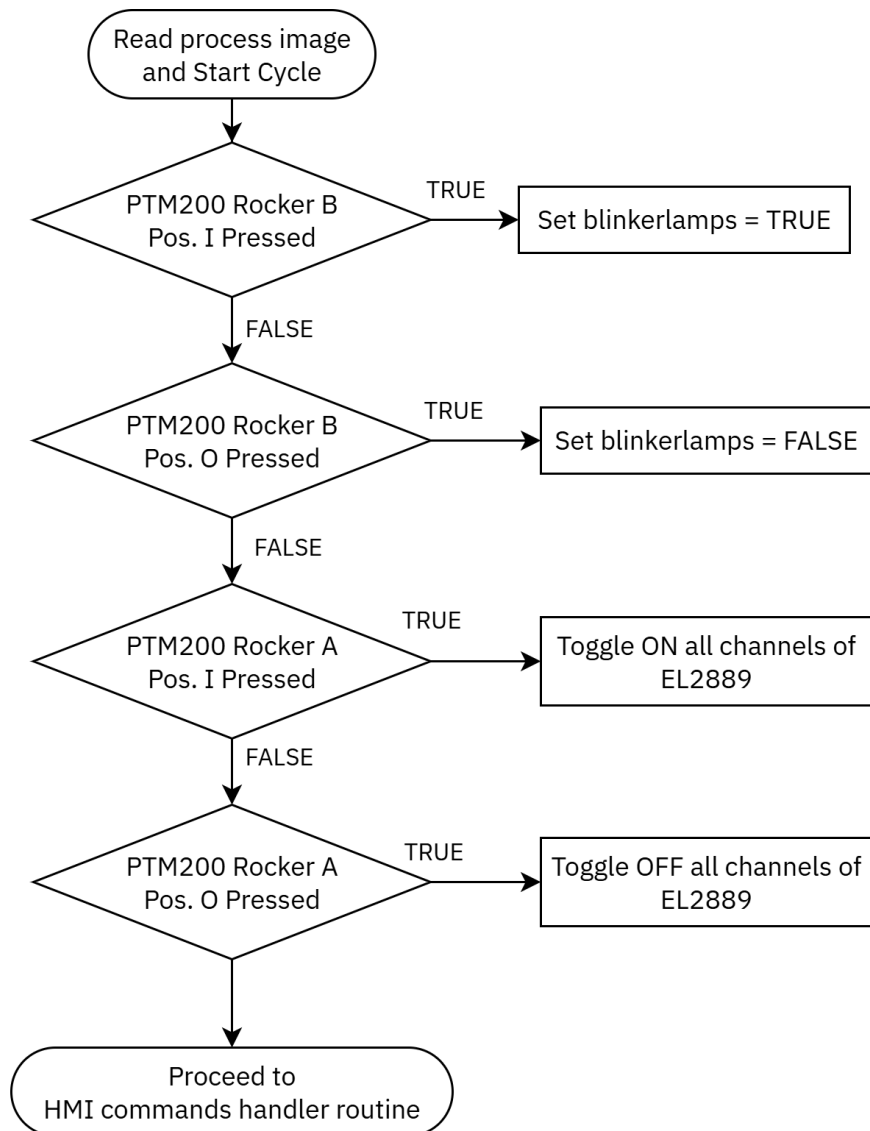


Figure 4.2.5(d)-1: EnOcean Routine in PLC program.

The control logic starts with the EnOcean routine, which handles inputs from the PTM200 and controls digital outputs and program flow via the blinkerlamps variable, which if TRUE, will blink all channels of the KL2889.

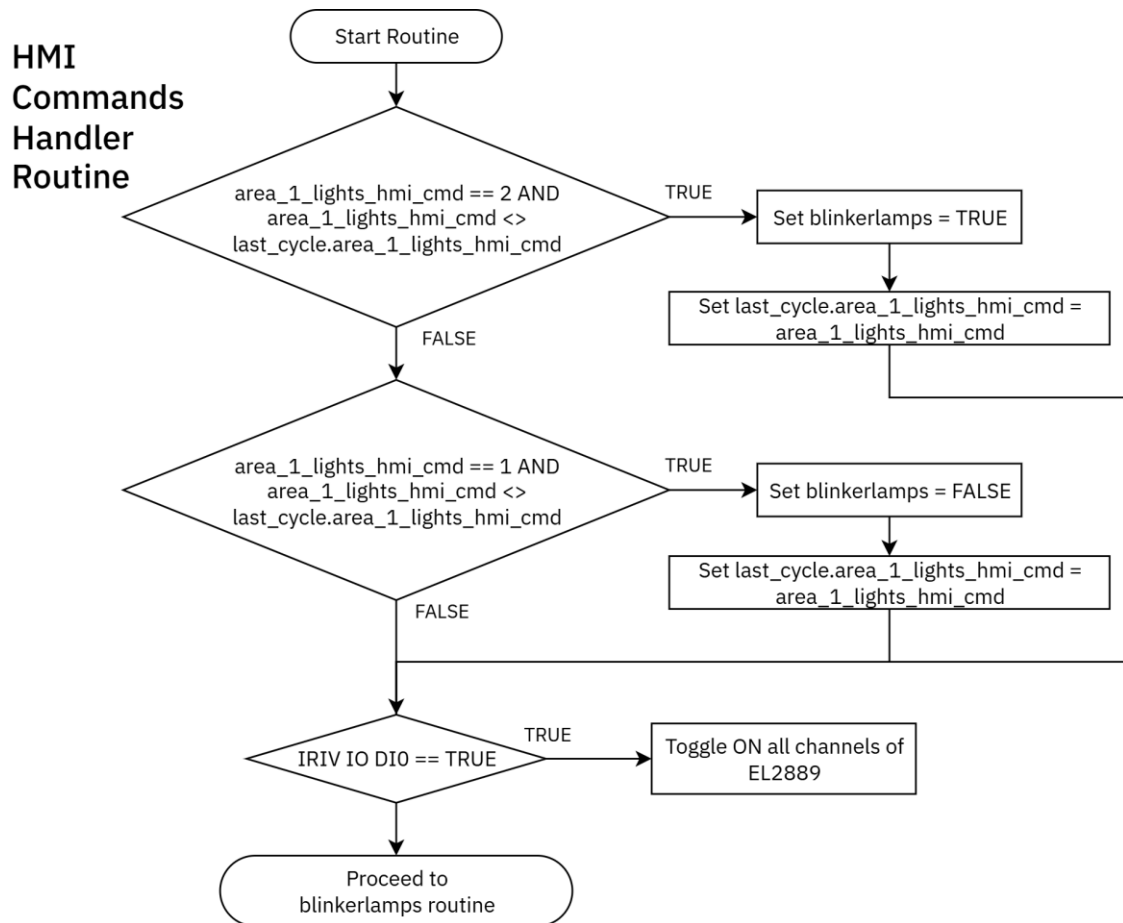


Figure 4.2.5(d)-2: HMI Commands Handler routine in PLC program.

The HMI may also control the same outputs controlled by the PTM200, however, priority is given to the PTM200 as its input may override that of the HMI.

blinkerlamps routine

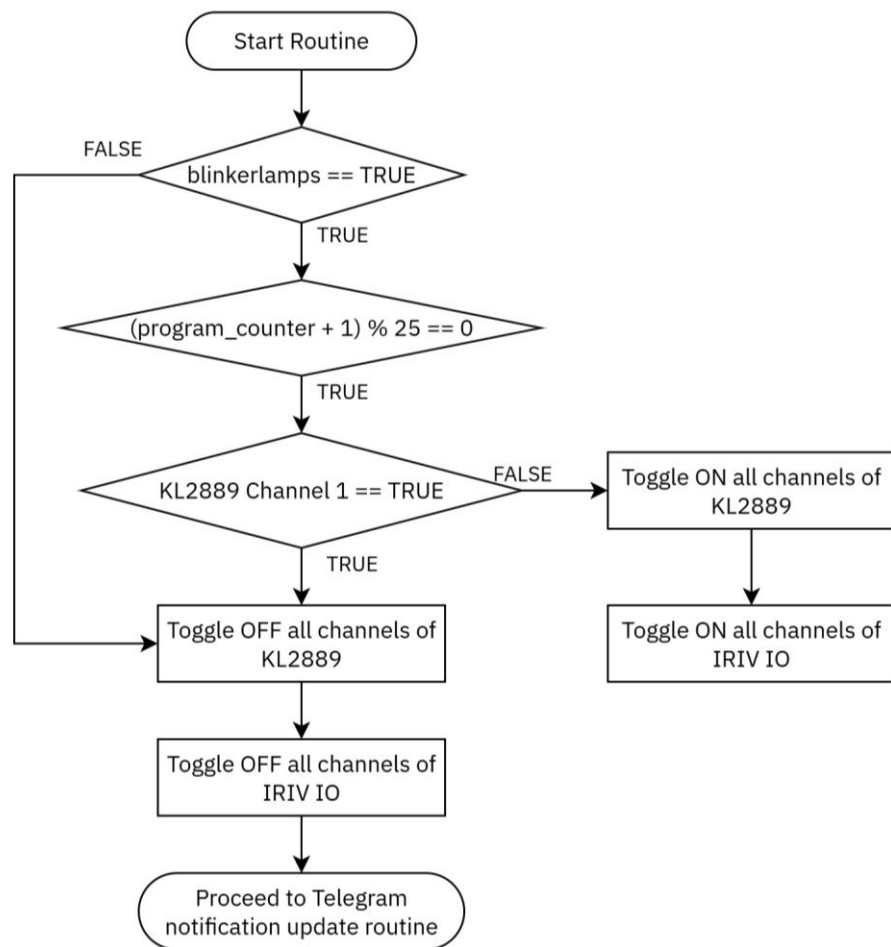


Figure 4.2.5(d)-3: Blinkerlamps routine in PLC program.

This routine only executes if the `blinkerlamps` variable is set, which occurs in the EnOcean routine and HMI Commands Handler routine as described in Figure 4.2.5(d)-1 and Figure 4.2.5(d)-2 respectively.

Telegram notification update routine

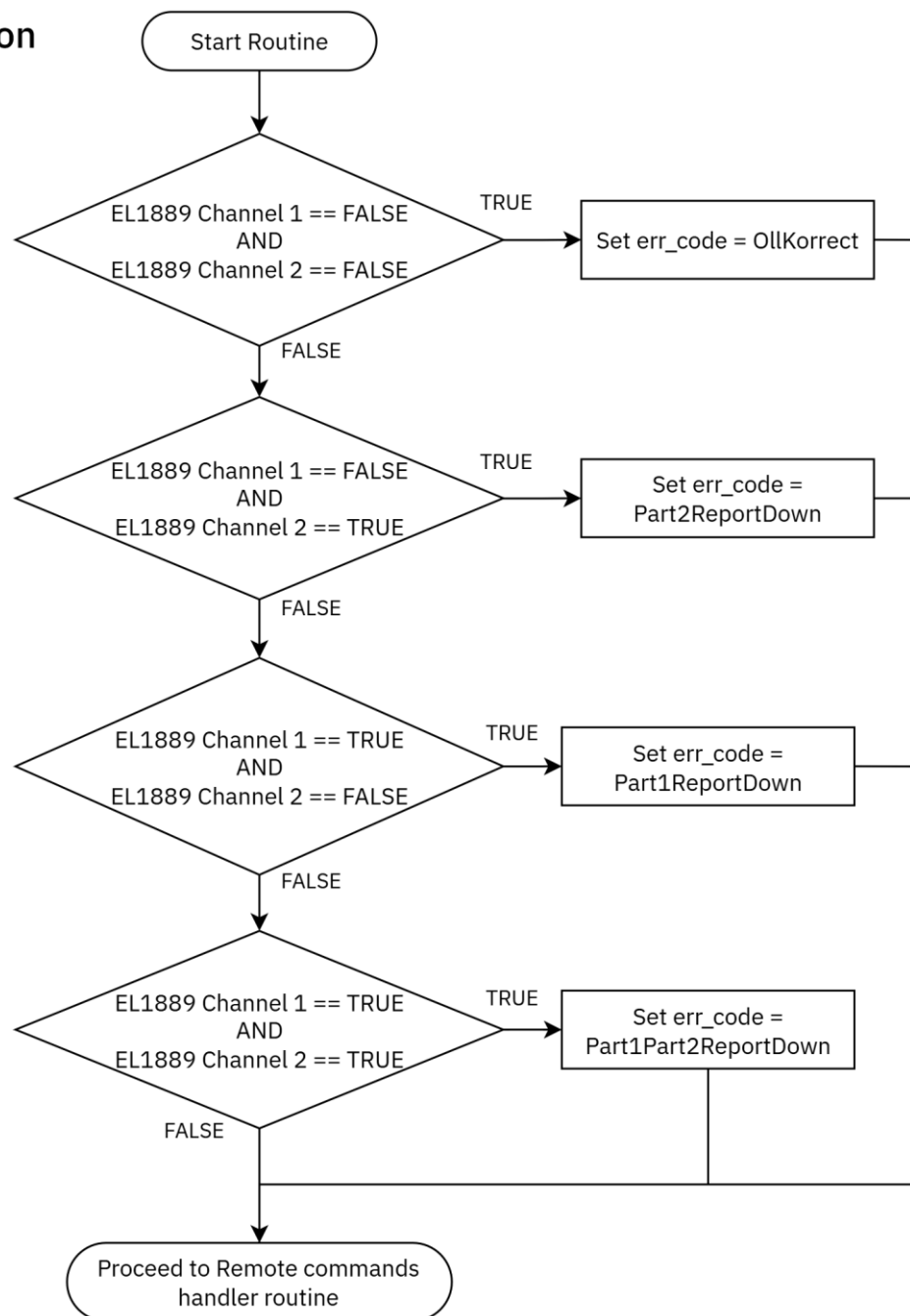


Figure 4.2.5(d)-4: Telegram Notification Update routine in PLC program.

The purpose of this routine is to demonstrate the Telegram bot. A direct message link to an appointed person in charge may be used to notify emergencies where urgent intervention is necessary.

Remote commands handler routine

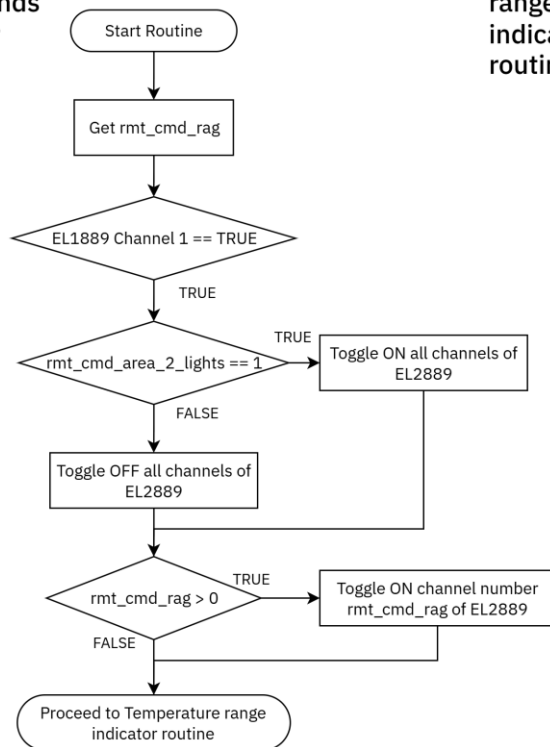


Figure 4.2.5(d)-5: Remote Commands Handler routine in PLC program.

Temperature range indicator routine

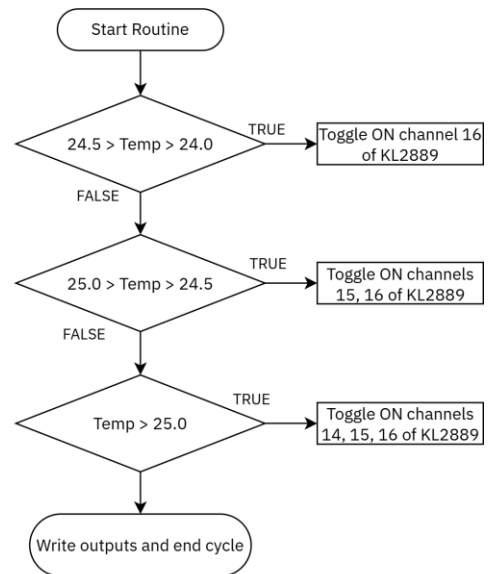


Figure 4.2.5(d)-6: Temperature Range Indicator routine in PLC program.

The Remote Commands Handler routine will only execute if Channel 1 of the EL1889 is toggled ON, functioning as an interlock. However in this case, since there is only a single action step conditioned on the ‘interlock’, it is effectively a simple permissive.

4.3 Gantt Chart and Milestones

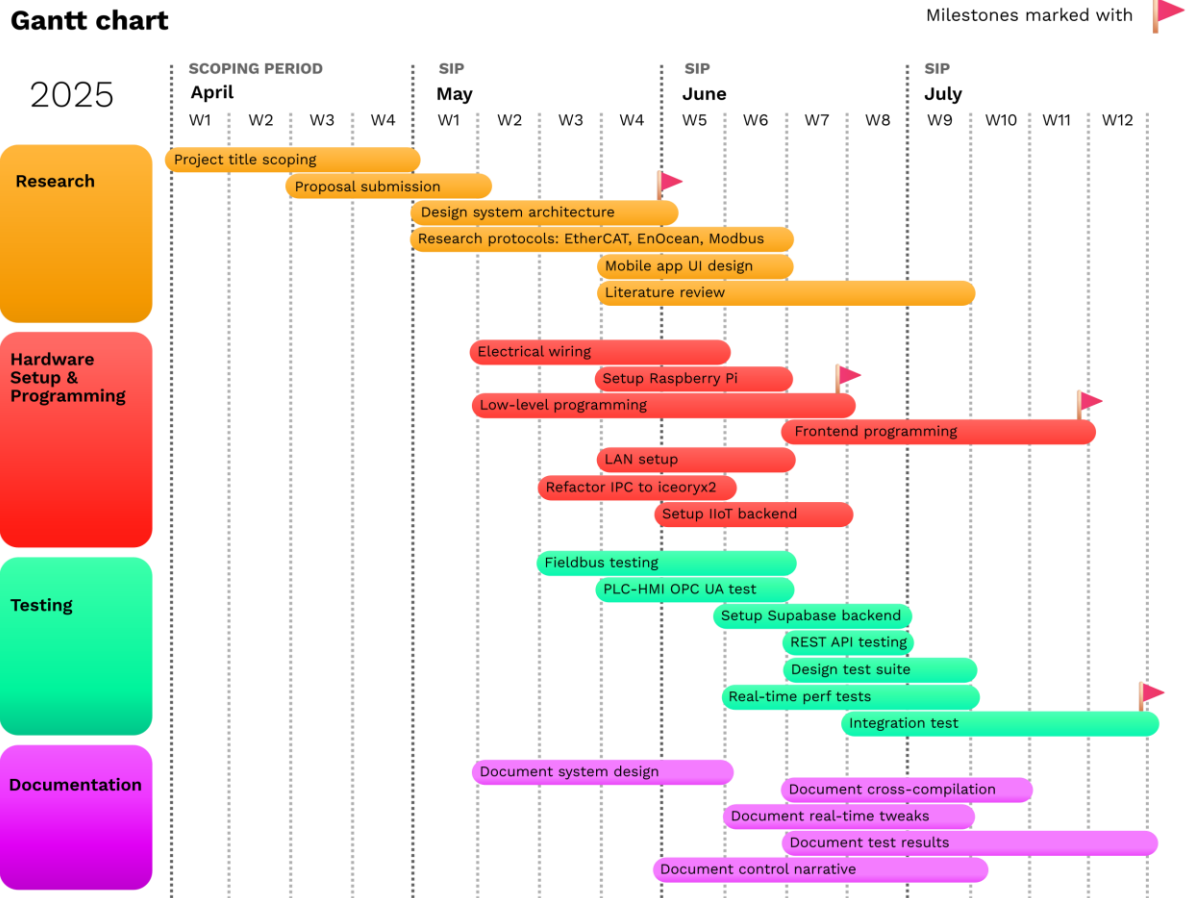


Figure 4.3-1: Project Gantt Chart and Milestones.

Milestones

Milestone	Completed in SIP Week
Design system architecture	4
Low-level programming	7
Frontend programming	11
Integration test	12

Table 4.3-1: Project Milestones and Week of Completion.

5.0 Results and Discussion

Testing of the system focused predominantly on real-time performance, being that the core functionality implemented is an EtherCAT-based control loop. Despite targeting a building automation system as a mock application example, at the very minimum, the system should have reasonable soft real-time capability, backed up by empirical testing.

Integration testing for IIoT functionality was conducted without expectation of real-time behavior of the IIoT components, since there are no critical deadlines and arbitrary retries are allowed (a natural consequence of the TCP/IP stack). Much of the IIoT infrastructure was outsourced to cloud backend providers. Thus, testing involved simple verification of CRUD operations via the Supabase API. First using a REST testing tool to directly test the PostgREST API, as well as end user testing of the mobile app client.

5.1 Results

The following tests were conducted without a GUI running, swap disabled, and wireless interfaces (Bluetooth and WiFi) soft blocked with *rftkill*. The Raspberry Pi 5 used was the basic B variant with 2GB of RAM. The Pi was not connected to the internet during real-time-performance-related testing. Also note that several non-real-time but critical processes are still run; namely, the OPC UA server and the Modbus/TCP polling process. The `tx-usecs` and `rx-usecs` parameters for the EtherCAT NIC were both set to 0 using *ethtool* to disable packet coalescing. In this analysis, ‘jitter’ refers to cycle activation jitter of the PLC, which is defined as the following:

$$5.1(1) \quad \text{Jitter} = t_{\text{sch},i-1} - t_{\text{ins},i}$$

where i is the current cycle, $t_{\text{sch},i-1}$ is the scheduled next cycle start time that was calculated in the immediate previous cycle, and $t_{\text{ins},i}$ is the time instance calculated in the current cycle. The control loop thread is instructed to sleep until wakeup for the next cycle. Hence in using the definition above, jitter is always non-negative.

5.1.1 *cyclicttest* Latency Results

To establish baseline latency of the Pi 5, *cyclicttest* was used, with additional stress loads run concurrently with *stress-ng*. This is conducted to survey the performance of the host hardware itself, generic with respect to the actual real-time application to be run.

Run	Maximum per-thread latency (μ s)	Maximum per-thread latency average across all 8 threads (μ s)	Average of average per-thread latency (μ s)
1	148	101.75	15.75
2	119	104.00	11.88
3	128	107.88	12.38

Table 5.1.1-1: Baseline Pi 5 latency results with stress load run concurrently.

5.1.2 Worst-Case 48-Hour Soak Test

Test results	Release (1ms cycle)
Maximum jitter (μ s)	210
Maximum sleep interval (μ s)	121

Table 5.1.2-1: Maximum jitter and maximum sleep interval in μ s over 48 hours.

A torture soak test was conducted over a span of 48 hours nonstop, with constant RAM utilization between 50-60%, and non-isolated CPU cores (CPU0-1) constantly utilized at 100%. The CPU temperature was verified to be below 60°C, far below the 85°C throttling threshold. *stress-ng* was used for the simulated CPU and RAM load, and was tweaked so that RAM utilization does not contend excessively and that the control loop remains operable.

5.1.3 12-Hour Soak Test

Maximum jitter (μ s)	206
Maximum sleep interval (μ s)	220

Table 5.1.3-1: Maximum jitter and maximum sleep interval in μ s over 12 hours.

(**Release** mode, 1ms cycle)

The 12-hour soak test recorded a similarly intolerable maximum jitter of 206 μ s. While this is intolerable for <10ms cycles (jitter must be <200 μ s), it is sufficient for slower cycles (>20ms) and is helpful in delineating where the real-time capability of the Pi 5 starts to fail. Similarly, the RAM utilization remained below the upper bound of 80%, staying between 50-60%.

5.1.4 Proxy Soak Test

Short-term tests (10-20 minutes) recorded maximum jitters between 194-206 μ s. To simulate the 48-hour soak test maximum jitter, an additional 30% scaling was applied to the short-term test maximum jitter. This 30% rule-of-thumb scaling was derived from the ratio between the rounded highest and lowest recorded maximum jitters.

Avg. RAM usage (%)	37	27	25	22	21
Test results					
Scaled maximum jitter (μ s)	188.5	160.0	160.0	100.1	154.7
Maximum sleep interval (μ s)	7374	7558	7718	8241	7663

Table 5.1.4-1: Proxy soak tests with *stress-ng* RAM and CPU stress test running
(**Debug** build mode, 10ms scan cycle).

Avg. RAM usage (%)	34	27	24	20	20
Test results					
Scaled maximum jitter (μ s)	83.2	74.1	74.1	85.8	81.9
Maximum sleep interval (μ s)	133	147	147	239	126

Table 5.1.4-2: Proxy soak tests with *stress-ng* RAM and CPU stress test running
(**Release** build mode, 500 μ s scan cycle).

The jitter upper limit is determined by the selected cycle period. The jitter must not exceed 2% of the cycle period. For a 10ms cycle, the maximum jitter must therefore be lower than 200 μ s, and for a 5ms cycle, the maximum jitter must be lower than 100 μ s.

5.1.5 Light Load Tests

Lighter load tests were conducted to see how light the system resource utilization has to be for the Pi to achieve better maximum jitter to support quicker scan cycles. This involved significantly reducing the amount of memory allocated to *stress-ng* memory stressor workers.

Avg. RAM usage (%)	21
Scaled maximum jitter (μ s)	27.3
Maximum sleep interval (μ s)	8537

Table 5.1.5-1: Light load test results (**Debug** build mode, 10ms scan cycle).

Avg. RAM usage (%)	18
Scaled maximum jitter (μ s)	16.9
Maximum sleep interval (μ s)	393

Table 5.1.5-2: Light load test results (**Release** build mode, 500 μ s scan cycle).

stress-ng was used to stress the memory using 64 concurrent workers, with 1MiB allocated to each. From this result, the achieved maximum jitter stayed bounded below 20-30 μ s. This is sufficient to support a 1ms scan cycle at the quickest (for a tolerance of <2% of cycle time maximum jitter). Depending on the exact application, the Pi is capable of fast motion control and multitasking, but is bottlenecked by the memory bus bandwidth.

From the results, the real-time performance of the Pi is not necessarily affected by the total percentage of RAM used, but rather the bandwidth utilization. The Pi is still capable of supporting scan cycles <10ms despite a large number of concurrent processes accessing RAM, but only such that each process does so in small enough chunks at one time.

5.1.6 Floating above Idle Test

This test is run without *stress-ng* or any other stressors in the background, other than the components needed for operation. Real-time performance should always be tested with additional stress conditions. However, the results of this test show the asymptote of real-time performance as the system load tends towards idle (best performance under perfectly ideal low system load).

Avg. RAM usage (%)	17
Scaled maximum jitter (μ s)	19.5
Maximum sleep interval (μ s)	8868

Table 5.1.6-1: Floating above idle test results
(20-minute run, **Debug** build mode, 10ms scan cycle).

Avg. RAM usage (%)	14
Scaled maximum jitter (μ s)	14.3
Maximum sleep interval (μ s)	405

Table 5.1.6-2: Floating above idle test results
(20-minute run, **Release** build mode, 500 μ s scan cycle).

5.1.7 12-Hour Test: Floating above Idle

Avg. RAM usage (%)	15
Maximum jitter (μ s)	22
Maximum sleep interval (μ s)	289

Table 5.1.7-1: Floating above idle test results
(12-hour run, **Release** build mode, 500 μ s scan cycle).

Without any stress load, the maximum jitter recorded over a 12-hour test run is 22 μ s. Comparing the 12-hour and the 20-minute run for the Release build, the 30% scaled maximum jitter is still significantly lower than the actual maximum jitter recorded. Hence, the 30% rule of thumb clearly does not work in this condition. Whilst the recorded maximum jitter is still lower than the 25 μ s upper bound (to satisfy <5% jitter for a 500 μ s cycle), it is not significantly lower with just a 3 μ s difference.

5.1.8 Testing of Heuristic: 12-Hour Light Soak Test

The additional 30% scaling heuristic was tested by conducting a 12-hour light soak test, using the same lighter stress load (upper bound of 25% RAM utilization).

Avg. RAM usage (%)	22
Actual maximum jitter (μ s)	78
Maximum sleep interval (μ s)	78

Table 5.1.8-1: 12-hour light soak test results (**Release** build mode, 500 μ s scan cycle).

Given that in Table 5.1.4-2, the scaled maximum jitter was 74.1 μ s for the 24-27% RAM usage regime, it is not far off from the actual 78 μ s maximum jitter measured from the 12-hour run.

5.1.9 Jitter Distribution

The following plots were obtained from short 20-minute test runs. Although real-time characteristics are quantified in terms of maximum jitter and latency recorded during worst-case scenarios, it is still useful to study the distribution of cycle jitter albeit from short test runs. In the following figures, the dashed orange lines indicate the boundary of $\pm 1\sigma$ (one standard deviation) away from the average.

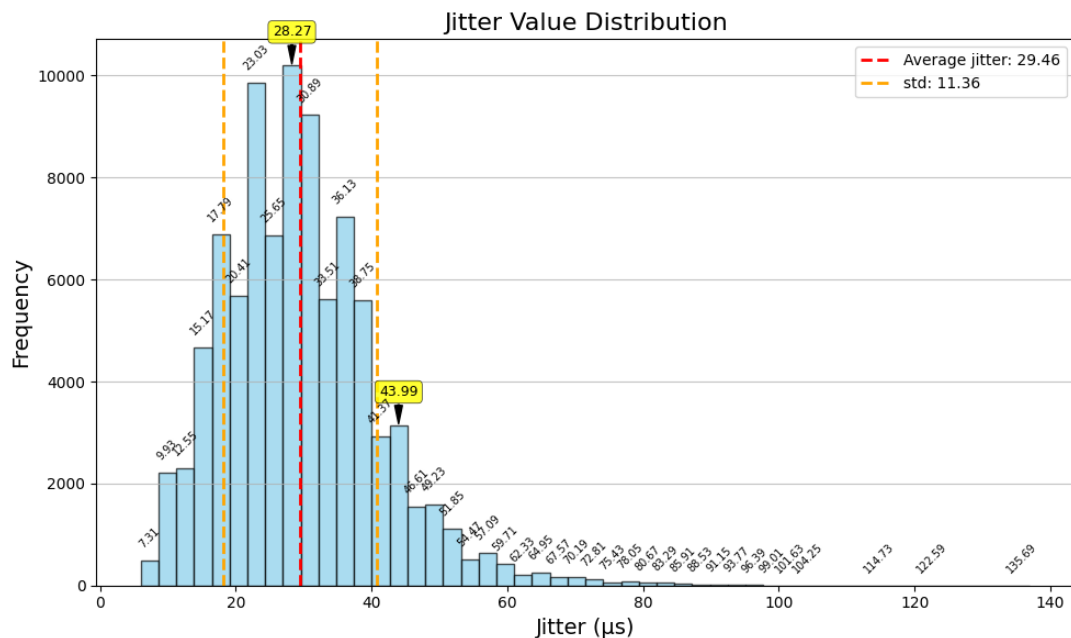


Figure 5.1.9-1: Jitter Distribution under Stress Load.

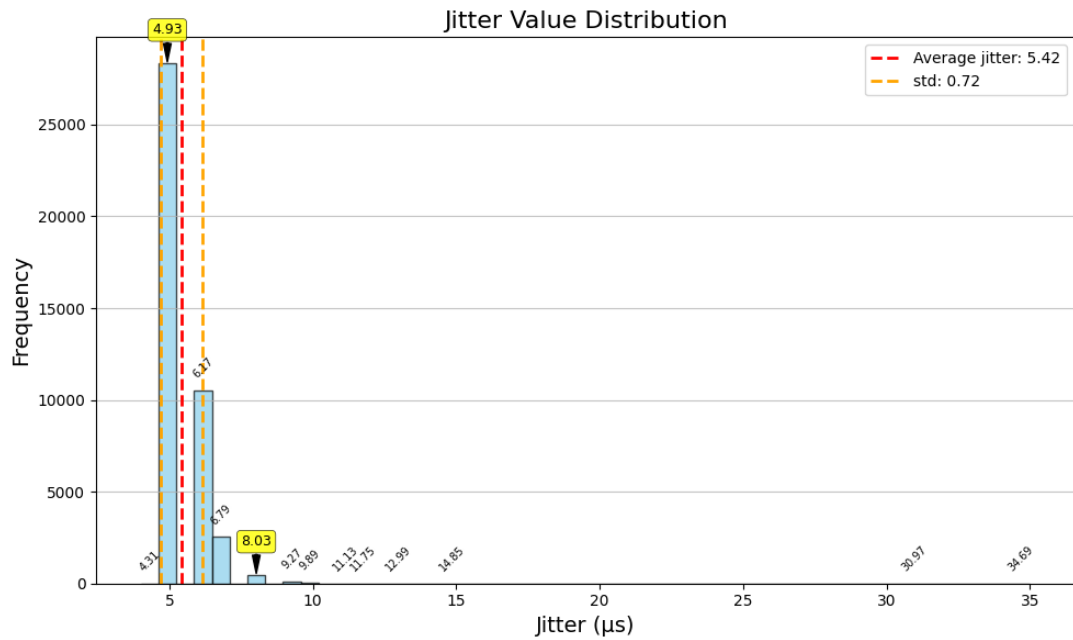


Figure 5.1.9-2: Jitter Distribution when Floating above Idle.

In both Figure 5.1.9-1 and Figure 5.1.9-2, there is a long tail to the right that drops sharply from the average. In Figure 5.1.9-1, the distribution follows a more Bell-curve like shape, in contrast with Figure 5.1.9-2 that has more of an exponentially decreasing curve. Although the long right tail in Figure 5.1.9-1 also appears like an exponentially decreasing curve, more intermediate points exist between the average and the maximum values, whereas in Figure 5.1.9-2, the frequency drop is much more abrupt.

5.1.10 Real-Time Performance Test Caveats

There are some notable caveats with regards to the real-time performance of the Pi. Primarily, real-time performance may vary widely with respect to the software architecture. In this project, only one EtherCAT MainDevice is tested, which runs in user mode. Better real-time performance may be attainable using a kernel-mode EtherCAT MainDevice, such as the IgH Etherlab Master developed and maintained by Florian Pose (Ingenieurgemeinschaft, 2025). Despite these caveats, the test results indicate that the Pi is able to guarantee cycle times of at least 5ms with less than 2% maximum jitter for the majority of applications with sufficiently light system load.

5.1.11 IIoT Integration Test

Tested Aspect	Qualitative Result
Supabase API Gateway Codes	All returned status code 200 (Success)
PLC Real-Time Performance Impact	None. Cycle time fully respected and jitter stays under 2% cycle time threshold (10ms cycle). Maximum jitter recorded was 35 μ s.
Mobile Client	Occasional UI rendering issues; Assets sometimes load slowly and animations get skipped.
HMI	UI elements respond slowly, but are sent to the PLC relatively quickly.

Table 5.1.11-2: Qualitative Integration Test Results.

Most issues uncovered during IIoT integration testing are frontend-related. Often having to do with the UI, this is not critical as the transmission of data on the backend of each node remains to be adequately reliable. One caveat of the integration test is that none of the nodes are subjected to soak tests, each node was subjected to normal loads, hence are not floating above idle, either.

Integration test durations are also short, as the internet connection is provided by the SM-A528B/DS hotspot. Using the building network to connect to the internet is a security risk, hence the duration of an integration test run never exceeded the duration of a work day.

5.2 Discussion

If the Pi is used for deployment, it should be run in console mode only to reduce the overhead introduced by a graphical desktop environment. VNC, RDP, or any other remote GUI interfacing should be avoided, and disabled when the PLC is running its control loop(s). Any serious deployment of the Pi must also exclusively use a dedicated SSD for maximum reliability. A microSD card is simply not a reliable storage medium, especially for host hardware running real-time control software. Internet connection should best be limited to a dedicated non-real-time Ethernet NIC.

From experimentation, without the use of memory lock syscalls, GUI applications almost always cause some missed cycles. This might be due to disk I/O DMA, that cannot be preempted even by real-time tasks on `SCHED_FIFO`. Calling `mlockall()` at the start of the real-time program seem to have made the real-time tasks resilient, even while the memory and disk I/O are stressed. As described in Madden (2019), the usage of memory locking in real-time contexts such as this is typical. Nevertheless, it is best for any graphical HMIs to be hosted on a separate device. The Pi is best suited for process data acquisition, light processing, and retransmission; to be done without the overhead of a graphical desktop environment.

With the knowledge that web browsers such as Chromium and Firefox require large memory and disk bandwidths, concurrent memory and disk stress tests were conducted using *stress-ng*, which reproduced the missed cycle deadlines and EtherCAT timeouts. Since this ultimately depends on hardware implementation of memory and disk I/O management of the host hardware, a different architecture (different SoC or a typical x86 motherboard configuration) may offer better real-time task resilience. However, `mlockall()` does provide a safeguard against page faults by guaranteeing that mapped pages of the calling process will stay in physical memory and are not evicted until `munlockall()` is called or the process terminates.

It is however, important to note that exact control operations of hardware resources cannot be preempted by user-space processes, despite being scheduled using `SCHED_FIFO` at high priorities. Unbounded latencies may still occur due to hardware controller operations, which are outside the control of even the kernel.

Other possibly confounding factors namely inter-process communications were eliminated, both theoretically and experimentally. *iceoryx2* is a lock-free wrapper of shared memory for inter-process communications. As such, it could not have blocked on any RwLocks or Mutexes (as none exist in the API internal data path) and would have simply returned errors if samples are not successfully sent or received. This was also tested experimentally; with or without inter-process communication operations, the presence of missed deadlines only respond to significant concurrent memory and disk loads, which can be triggered by normal web browsing and *stress-ng*.

5.2.1 Sub-Millisecond Real-Time Performance

At least in Release build mode, the Pi is able to support sub-millisecond scan cycles, however, from testing, it is unable to support maximum cycle jitter of <3% of the cycle time for a 500 μ s cycle, if the specific PLC runtime architecture in this report is used without additional load (floating above idle). With a sufficiently light system load, it can achieve a maximum jitter of <5% of a 500 μ s cycle, amounting to a bound on the maximum jitter of 25 μ s. Since this level of performance is only observed with light stress tests, the ability to support sub-millisecond cycles on the Pi is only a limited guarantee and heavily depends on the exact application.

5.2.2 Comparison with Equivalent Benchmarks

Qiu, Varis, and McArthur (2024) tested a CODESYS system deployed on several Texas Instruments ARM SoCs. One SoC tested was the AM62x, a quad-core SoC with 512KB of shared L2 cache clocked at 1.4GHz maximum. The AM62x is the most equivalent SoC to the quad-core BCM2712 of the Raspberry Pi 5, though the BCM2712 does have a higher clock frequency at 2.4GHz.

The rest of the SoCs tested by Qiu et al. had specifications that were too disparate from the BCM2712 that would make it an unfair comparison either way. Namely, the AM69 is an octa-core SoC clocked at 2GHz with 1MB of shared L2 cache, while the AM64x is a dual-core SoC clocked at 800-1000MHz, with 256KB of shared L2 cache. It is important to note that the results from Qiu et al. did not involve a stress load running concurrently. Hence, an ideally light load.

SoC	Maximum jitter recorded (μ s)	Maximum cycle time (μ s)	Minimum time spent not working (μ s)
AM62x	116	700	300
AM69	53	384	616
TDA4VM	65	371	629
AM64x	973	1906	0

Table 5.2.2-1: Adapted results from Qiu et al. (2024). The column “Minimum time spent not working” is derived by subtracting the 1ms cycle time used by Qiu et al. from the maximum recorded cycle time.

For the particularly underpowered AM64x, it failed to respect the 1ms cycle at least once. The Raspberry Pi 5B recorded a maximum jitter of 210 μ s in the worst-case stress test, and 22 μ s when floating above idle. The AM62x recorded 116 μ s without a concurrent stress load.

5.2.3 Functional Safety

Soft PLCs are incredibly powerful and flexible, but being used for real-time controls, there are a lot more critical factors that need to be taken into consideration to ensure reliable and safe operation. Being soft real-time systems, (non-safety) PLCs cannot be solely relied upon for safety. Although PREEMPT_RT promises bounded latencies, it does not provide any theoretical and formally quantifiable bounds. It is practically impossible to statically analyze a GPOS such as in the case of a Linux PREEMPT_RT-enabled distro the same way more typical RTOSs (or bare-metal/OS-less control subsystems) are analyzed (Reghenzani, Massari, & Fornaciari, 2019).

Ultimately, hard real-time systems (e.g. aerospace, automobile passive restraints, nuclear fission reactors, etc.) are application-specific, and require bespoke-architected solutions. When used as part of a larger application system (e.g. assembly line, continuous/batch chemical processes, etc.), soft real-time subsystems by themselves are not sufficient. The vast majority of non-safety PLCs on the market fall within the category of soft real-time, as even ‘hard’ microcontroller-based non-safety PLCs are designed to operate in general situations, and not with any specific production floor layouts and contextual factors in mind.

However, the degree to which an entire application system is functionally safe cannot be quantified by the real-time firmness of the PLC alone. In practice, functional safety must be realized with hard real-time safety subsystems (e.g. e-stops, light curtains, safety relays, FSoE, PROFISAFE, etc.). Any application system cannot solely rely on any single specific subsystem to implement functional safety, at least up to the required SIL in accordance with relevant standards.

5.2.4 The Pi vs. Other Host Hardware

Compared to less-capable though cheaper SBCs, the quad-core ARM SoC of the Pi can be used for redundant processing or independent concurrent control of multiple processes, with lower power consumption and better thermals. The Pi is more capable of running the plethora of computer vision libraries and APIs, and is more capable of edge inference of vision or machine learning models. This makes it a compelling alternative against vendor-locked peripherals and software stack (an example being vision solutions from KEYENCE).

However, the ability to run such additional software is not unique to the Pi, as they can be deployed on any host hardware running a GPOS. When contrasted with more expensive industrial PC offerings (such as ones from ASRock, Advantech, etc.), in the vast majority of situations, the latter is orders of magnitude more reliable; being purpose-built for use in industrial environments. The Pi has scant advantages compared to IPCs, notwithstanding its lower upfront cost.

5.2.5 Improving Operational Security

IR 4.0 principles inherently require a network of control devices. Such network of control devices serve as a wide attack surface (Honeywell International, 2021). This is unavoidable, but it does not mean that it cannot be mitigated with best operational security practices. Currently, by virtue of being an experimental system, the PLC runtime makes calls to the remote database via HTTPS with encrypted payloads. Nonetheless, this is not enough, a VPN tunnel should be used and is in fact already standard practice in the industry. Zero trust should also be assumed; processes should be executed with the least amount of privilege possible for operation.

The subsystem binaries are also run as root. This can be avoided by setting the minimum required capabilities for non-root user(s) to run the binaries. Supervisory tools such as Monit can be used to periodically check for suspicious processes and system resource utilization. In particular, Monit can be used to send alerts and be set up to judiciously execute specific actions, such as sending SIGTERM/SIGKILL to misbehaving non-critical processes.

5.2.6 Extending Support for IEC 61131-3

IEC 61131-3 describes the industry standard for programming languages used to program PLCs, among which, Structured Text and Ladder Diagram being the most popular. Unfortunately, feature-complete open source standalone compilers for IEC 61131-3 are few and far between. This is in stark contrast with general purpose programming languages such as C, which has numerous standards-compliant compilers. Part of this discrepancy is due to the tight coupling between the compiler and the runtime in typical proprietary IEC 61131-3 implementations, and the relative simplicity of C itself.

Despite this, compilers such as MATIEC and RuSTy exist, though the former is not active, while the latter is still under heavy development. Both of these compilers are also vastly different in implementation. MATIEC is written in C++ and works by converting IEC 61131-3 source code into ANSI C, whereas RuSTy is written in Rust and utilizes LLVM as the backend, hence generating LLVM IR from Structured Text source code. It is important to note that these are standalone compilers, and a dedicated runtime must be built upon the API provided by these compilers, should any future work be done to support IEC 61131-3. Basic features such as online changes or hot reloading/swapping are then up to the runtime implementation. This will involve additional subsystems such as a debugger for online variable read/writes.

5.2.7 Lock-Free Design

The control logic should ideally not hold locks to the data structure also accessed by the EtherCAT MainDevice. Currently, this is not much of an issue, as the lock accesses are carefully laid out and control logic evaluation is synchronous, hence blocking. However, this is not a resilient or expandable design, and is prone to deadlocking. Additionally, it prevents separation of cyclic tasks. A single EtherCAT MainDevice cyclic task should be able to handle process data for multiple (a)cyclic tasks that may run on different cycle times. This is the implementation used by CODESYS and some of its derivatives; for example, PLCNext (CODESYS Group, n.d.-a; WAGO GmbH, 2022). Holding locks to shared data structures makes this common setup impossible to implement without additional overhead for complex task scheduling. Despite that, if more than one task has write access, a Mutex is the simplest solution, as implemented by TwinCAT 3 (Beckhoff Automation, n.d.-c).

5.2.8 Relevance to Open Process Automation (OPA)

A software-defined approach for PLCs is a critical part of implementing an open and interoperable system. An open-source soft PLC runtime and development environment builds upon the already-pervasive CODESYS platform used by different vendors. Despite that, CODESYS remains to be a closed-source, proprietary platform. Its closed-source nature remains to be an obstacle against completely liberating industrial automation from vendor locking. For example, Beckhoff had announced TwinCAT PLC++, a complete rewrite of their CODESYS-based soft PLC runtime and development environment (Beckhoff Automation, n.d.-d).

While it appears to be a promisingly great product, Beckhoff has no plans to make it completely vendor-agnostic, and such exclusivism is inherently against the openness ethos. It is then imperative that an open-source, community-led soft PLC runtime and development environment is worked on, and this particular project discussed in this report is a glimpse of such vision.

A relevant industrial standard to gauge openness and interoperability is the O-PAS standard, published and maintained by The Open Group, a coalition of over 100 industry members (The Open Group, n.d.). Such open-source, Linux-based PLC platform would satisfy Application Layer L as defined in Part 1 of the O-PAS Standard (The Open Group, 2023). Any sort of vendor-locking will tie the entire layer to specific hardware, or at least limit interoperability at Layers L and above. An open and modular software architecture akin to the one implemented in this project also readily lends itself to operating as a DCN, as is already being used at ExxonMobil's Resin Finishing Plant in Baton Rouge, Louisiana (Kasprzak, 2025).

5.3 Sustainability

The project finds relevance in four of the 17 United Nations Sustainable Development Goals (UNSDG), spanning across three pillars of sustainability, namely; Environmental, Social, and Economic.

5.3.1 Environmental

Target		Project relevance
7.b	Expand infrastructure and upgrade technology for supplying modern and sustainable energy services	Modular and flexible software-defined architecture adapts to existing instrumentation, facilitating infrastructure control system expansion and upgrade

Table 5.3.1-1: Project Environmental UNSDG (Goal 7).

Adapted from United Nations (2015).

Target		Project relevance
11.2	Provide access to safe, affordable, accessible, and sustainable transport systems by expanding public transport	Project is sufficiently robust to support non-critical control systems for public transport infrastructure such as access control, ticketing, station platform HMIs, surveillance and assistance request systems

Table 5.3.1-2: Project Environmental UNSDG (Goal 11).

Adapted from United Nations (2015).

The project supports Target 7.b in expanding infrastructure and upgrading technology in the service of sustainable energy services. The modular and flexible software-defined architecture seamlessly integrates with existing instrumentation, directly facilitating the expansion and upgrade of the control systems that power infrastructure.

In addition, the project supports Target 11.2 in a similar manner. Public transportation infrastructure also requires non-critical control systems that can be controlled with sufficient robustness using the system implemented in this project. Non-critical control systems do not require airtight functional safety guarantees. The system implemented may be applied in HMIs that simply visualize and show time of

arrival, station information, etc., whose downtime will not result in catastrophic failure. Hence, such applications do not require functional safety and can benefit from the open architecture of the system.

5.3.2 Economic

Target		Project relevance
9.3	Increase access of small-scale industrial and other enterprises to integration into value chains and markets	Offers lower cost alternative for automating facilities and machinery, without sacrificing state of the art IR 4.0 capabilities
9.4	Upgrade infrastructure and retrofit industries to make them sustainable	Hardware-agnostic, open source architecture maximize compatibility with wide variety of already-existing equipment
9.b	Support domestic technology development, research and innovation in developing countries	This project lays the groundwork for a Malaysian product to make a break in the automation industry

Table 5.3.2-1: Project Economic UNSDG. Adapted from United Nations (2015).

With regards to the Economic pillar of Sustainability, the project aids in advancing inclusive industrialization and innovation, as per UNSDG 9. More specifically, per Target 9.3, the project enables small-scale enterprises to embrace and adopt automation and not be left behind in IR 4.0. The hardware-agnostic and open-source architecture maximizes compatibility with existing equipment, thus helping to achieve Target 9.4 which aims to upgrade infrastructure and retrofit industries to make them sustainable.

The project comprehensively leverages cost-effective, software-defined solutions that can be supported even on commodity hardware, thus lowering the barrier to entry and leveling up the playing field for small- and medium-scale enterprises. Furthermore, the solution that the project offers can extend the operational lifespan of legacy systems that are still capable of operation. This ultimately increases resource efficiency and reduces waste.

5.3.3 Social

Target		Project relevance
17.6	Enhance cooperation on and access to science, technology, and innovation and enhance knowledge sharing	Project abides by GNU General Public License; Learnings from the project have been shared publicly to the global open source community
17.16	Enhance the Global Partnership for Sustainable Development; Mobilize and share knowledge, expertise, technology	Project relied on FOSS dependencies; Contributors to open-source dependencies are from all around the globe

Table 5.3.3-1: Project Social UNSDG. Adapted from United Nations (2015).

The project also finds significant relevance in the Social pillar of Sustainability. In particular, the project fosters scientific and technological cooperation and knowledge sharing, thus helping to reach Target 17.6. The dependencies used in the implementation of the project are all open-source, whose contributors coming from various backgrounds and nationalities. The project aids to enhance the global partnership for sustainable development by mobilizing and sharing knowledge, expertise, and technology. The project was not developed in a vacuum, and numerous knowledge exchange within the open source community was involved in its development.

This project is deeply rooted in the FOSS ecosystem, and does not live in a silo. Rather, its source code, documentation, and implementation insights are openly accessible thus lending itself to community collaboration. The indirect effect of being an open source project is the low direct cost of auditing, as the source is publicly accessible and contributors have a vested interest in the quality of dependencies in the FOSS ecosystem. The Social aspect is especially significant due to the self-reinforcing cycle of quality and accountability, as the development does not occur behind opaque walls.

6.0 Conclusion

In conclusion, the project has successfully met the objectives set out. The feasibility of a fully open-source based commercial-grade automation system is possible within the realm of building automation. A functional prototype of a BAS was created using a fully FOSS stack, hence demonstrating the feasibility of applying open source tools. HMI/SCADA functionality and IIoT integration was demonstrated across multiple platforms including desktop and mobile. Deployment reliability was quantified by measuring jitter, latency, uptime, system resource utilization, and CPU temperature range. The extent of possible future expansion into more critical applications was also evaluated, taking into consideration possible expansion into process automation and involvement of motion controls and functional safety.

References

- Baeldung. (2023, May 5). *Guide to the “Cpu-Bound” and “I/O Bound” Terms*. Retrieved July 7, 2025, from <https://www.baeldung.com/cs/cpu-io-bound>
- Ballo, T., Ballo, M., & James, A. (2022). *High Assurance Rust: Developing Secure and Robust Software* [Online]. <https://highassurance.rs>
- Beckhoff Automation. (2021, November 15). “*We are ensuring a competitive advantage.*” Retrieved June 10, 2025, from <https://www.beckhoff.com/en-en/company/news/we-are-ensuring-a-competitive-advantage.html>
- Beckhoff Automation. (2023). *Documentation | EN KL6581 and KL6583 EnOcean Master Terminal and Receiver*. Retrieved July 17, 2025, from https://www.beckhoff.com/ms-my/support/download-finder/search-result/?download_group=37014605&download_item=356189743
- Beckhoff Automation. (n.d.-a). *TwinCAT 2 / System Concept*. Beckhoff Information System - English. Retrieved June 12, 2025, from <https://infosys.beckhoff.com/english.php?content=../content/1033/tcsystemover/12695813131.html&id=>
- Beckhoff Automation. (n.d.-b). *TwinCAT 3 / Basics — Real-Time*. Beckhoff Information System - English. Retrieved June 12, 2025, from <https://infosys.beckhoff.com/english.php?content=../content/1033/tcsystemover/12695813131.html&id=>
- Beckhoff Automation. (n.d.-c). *Multi-task data access synchronization in the PLC*. Beckhoff Information System. Retrieved July 3, 2025, from https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/45844579955484184843.html&id=
- Beckhoff Automation. (n.d.-d). *TwinCAT PLC++: Next generation PLC technology*. Retrieved July 3, 2025, from <https://www.beckhoff.com/en-en/company/news/twincat-plc-next-generation-plc-technology.html>
- Beckhoff Automation. (n.d.-e). *TwinCAT / Automation software*. Retrieved July 8, 2025, from <https://www.beckhoff.com/en-en/products/automation/twincat/>
- Beckhoff Automation. (n.d.-f). *EtherCAT / System Description*. Beckhoff Information System. Retrieved July 23, 2025, from <https://infosys.beckhoff.com/english.php?content=../content/1033/ethercatsystem/1036980875.html&id=>

- Broling, T. (2007, September 5). *File:EthercatOperatingPrinciple.webm* - Wikimedia Commons. Retrieved July 8, 2025, from <https://commons.wikimedia.org/wiki/File:EthercatOperatingPrinciple.webm>
- Brown, J. H., & Martin, B. (2010). *How fast is fast enough? Choosing between Xenomai and Linux for real-time applications*. Invariant Systems Inc., Cambridge, MA. Tech. Rep. [Online]. Retrieved from <https://web.archive.org/web/20151004142300/https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf>
- CODESYS Group. (2024). *Definitions of Jitter and Latency*. Retrieved July 7, 2025, from https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_task_configuration_jitter_definitions.html
- CODESYS Group. (n.d.-a). *Bus Cycle Task – EtherCAT*. Retrieved July 3, 2025, from https://content.helpme-codesys.com/en/CODESYS%20EtherCAT/_ecat_buscycle_task.html
- CODESYS Group. (n.d.-b). *Your device with CODESYS*. Retrieved July 7, 2025, from <https://www.codesys.com/device-manufacturers/codesys-for-you/your-device-with-codesys/>
- Dunn, A. (2009, June 12). *The Father of Invention: Dick Morley looks back on the 40th anniversary of the PLC*. Manufacturing AUTOMATION. <https://www.automationmag.com/855-the-father-of-invention-dick-morley-looks-back-on-the-40th-anniversary-of-the-plc/>
- Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., & Halderman, J. A. (2014, November). The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (pp. 475-488). <https://dl.acm.org/doi/pdf/10.1145/2663716.2663755>
- EnOcean Alliance. (2017). *EnOcean Equipment Profiles (EEP)*. EnOcean Alliance - Technical Task Group Interoperability. Retrieved July 17, 2025, from https://www.enocean-alliance.org/wp-content/uploads/2017/05/EnOcean_Equipment_Profiles_EEP_v2.6.7_public.pdf
- EtherCAT Technology Group FAQ*. (2025). EtherCAT Technology Group. <https://www.ethercat.org/en/faq.html#779>
- Feo-Arenis, S., Westphal, B., Dietsch, D., Muñoz, M., Andisha, S., & Podelski, A. (2016). Ready for testing: ensuring conformance to industrial standards

- through formal verification. *Formal Aspects of Computing*, 28(3), 499–527.
<https://doi.org/10.1007/s00165-016-0365-3>
- Heller, M. (2023, February 3). *What is garbage collection? Automated memory management for your programs*. InfoWorld. Retrieved July 22, 2025, from
<https://www.infoworld.com/article/2337816/what-is-garbage-collection-automated-memory-management-for-your-programs.html>
- Henriksson, R. (1998). *Scheduling Garbage Collection in Embedded Systems* [PhD dissertation, Lund University].
<https://portal.research.lu.se/files/5860617/630830.pdf>
- Honeywell International. (2021). *Application Whitelisting for Better Industrial Control System Defense - Service Note (SV-21-04-ENG)*. Retrieved June 30, 2025, from <https://process.honeywell.com/content/dam/process/en/documents/gated/Honeywell-AWL-Service-Note.pdf>
- Huang, J., & Yang, C.F. (2017). *Effectively Measure and Reduce Kernel Latencies for Real-Time Constraints*. Embedded Linux Conference North America, Portland, Oregon, United States. <http://events17.linuxfoundation.org/sites/events/files/slides/ELC2017-%20Effectively%20Measure%20and%20Reduce%20Kernel%20Latencies%20for%20Real-time%20Constraints%20%281%29.pdf>
- Ingenieurgesellschaft. (2025). *IgH EtherCAT Master 1.6.6 Documentation (1.6.6-2-gb82a6673)*. Retrieved July 7, 2025, from https://docs.etherlab.org/ethercat/1.6/pdf/ethercat_doc.pdf
- Intel Corporation. (2022). *Improving Real-Time Performance of CODESYS Control Applications with Intel's Real-Time Technologies (757527-1.0)*. Retrieved July 1, 2025, from <https://builders.intel.com/solutionslibrary/improving-real-time-performance-of-codesys-control-applications-with-intel-s-real-time-technologies>
- International Electrotechnical Commission. (2023). *Industrial Communication Networks - Fieldbus Specifications - Part 1: Overview and Guidance for the IEC 61158 and IEC 61784 Series (IEC 61158-1:2023)*. Retrieved June 10, 2025, from <https://webstore.iec.ch/en/publication/66931>
- International Electrotechnical Commission. (2024). *Programming Languages — C (ISO/IEC Standard No. 9899:2024)*. <https://www.iso.org/standard/82075.html>

- Kasprzak, S. (2025, March 3). New insights: 100-controller ExxonMobil Open Process Automation. *Control Engineering*. Retrieved July 17, 2025, from <https://www.controleng.com/new-insights-100-controller-exxonmobil-open-process-automation/>
- Madden, M. M. (2019). *Challenges Using Linux as a Real-Time Operating System*. NASA Langley Research Center. Retrieved July 7, 2025, from https://www.researchgate.net/publication/330199464_Challenges_Using_Linux_as_a_Real-Time_Operating_System
- McKenney, P. (2005, August 10). *A Realtime Preemption Overview*. LWN.net. Retrieved June 12, 2025, from <https://lwn.net/Articles/146861/>
- Microsoft. (2006, June 29). *Benchmarking Real-Time Determinism in Microsoft Windows CE*. Microsoft Learn. Retrieved June 12, 2025, from [https://learn.microsoft.com/en-us/previous-versions/windows/embedded/ms836535\(v=msdn.10\)?redirectedfrom=MSDN](https://learn.microsoft.com/en-us/previous-versions/windows/embedded/ms836535(v=msdn.10)?redirectedfrom=MSDN)
- Microsoft. (2019, July 18). *We need a safer systems programming language*. Microsoft Security Response Center. Retrieved July 23, 2025, from <https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/>
- Milewski, B. (2015, January 13). *Simple algebraic data types*. Bartosz Milewski's Programming Cafe. Retrieved July 23, 2025, from <https://bartoszmilewski.com/2015/01/13/simple-algebraic-data-types/>
- Perez, D. J., Walzl, J., Prenzel, L., & Steinhorst, S. (2022, September). *How Real (Time) are Virtual PLCs?* In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)* (pp. 1-8). IEEE.
- Qiu, D., Varis, P., & McArthur, J. (2024). *Performance Metrics of TI Embedded Processors as CODESYS EtherCAT Controller* (No. SPRADH0). Texas Instruments. Retrieved July 1, 2025, from <https://www.ti.com/lit/an/spradh0/spradh0.pdf#:~:text=The%20primary%20performance%20metric%20is%20the%20shortest%20achievable,both%20within%20the%20controller%20and%20within%20the%20network.>
- Reghenzani, F., Massari, G., & Fornaciari, W. (2019). The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM Computing Surveys (CSUR)*, 52(1), 1-36.
- Rostedt, S. (2016). *Understanding a Real-Time System*. Kernel Recipes, Paris, France. <https://elinux.org/images/f/f1/Rostedt.pdf>

- RT-mutex subsystem with PI support* — *The Linux Kernel Documentation*. (n.d.). The Linux Kernel Documentation. Retrieved July 7, 2025, from <https://docs.kernel.org/locking/rt-mutex.html>
- Siemens. (n.d.). The Virtual PLC is Revolutionizing Production at Audi. Retrieved June 10, 2025, from <https://www.siemens.com/global/en/company/stories/industry/factory-automation/virtual-plc-audi.html>
- The Chromium Projects. (n.d.). *Memory safety*. Retrieved July 23, 2025, from <https://www.chromium.org/Home/chromium-security/memory-safety/>
- The Open Group. (2023). *O-PASTM Standard, Version 2.1: Part 1 – Technical Architecture Overview (Informative)* (No. C230-1).
- The Open Group. (n.d.). *Open Group OPA Forum Membership Report*. Retrieved July 3, 2025, from https://reports.opengroup.org/opa_forum.shtml
- Understanding Linux Real-Time with PREEMPT_RT Training*. (2025). Bootlin. <https://bootlin.com/doc/training/preempt-rt/>
- United Nations. (2015). Transforming Our World: The 2030 Agenda for Sustainable Development. In *United Nations Digital Library* (A/RES/70/1). Retrieved July 8, 2025, from <https://digitallibrary.un.org/record/3923923?v=pdf>
- VanderLeest, S. (2022, November 16). *Linux in Aerospace: A Personal Journey*. Linux.com. Retrieved June 30, 2025, from <https://www.linux.com/news/linux-in-aerospace-a-personal-journey/>
- WAGO GmbH. (2022). *Relationship between Bus Cycle and Task*. Retrieved July 3, 2025, from <https://techdocs.wago.com/Software/EtherCAT/en-US/1443591051.html>
- Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., & Kaashoek, M. F. (2012, July). Undefined Behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems* (pp. 1-7).
- Wayand, B. (2020, March 20). *What is a PLC (Programmable Logic Controller)?* MRO Electric Blog. <https://www.mroelectric.com/blog/what-is-a-plc/>
- Wu, X., & Xie, L. (2019). Performance Evaluation of Industrial Ethernet Protocols for Networked Control Application. *Control Engineering Practice*, 84, 208-217.

Appendix A

Links to Project GitHub Repositories

The following are repositories hosted on GitHub of the software components created to implement the project.

Main soft PLC implementation:	www.github.com/andergisomon/Gipop/
IIoT Supabase-OPC UA Gateway:	www.github.com/andergisomon/sunsuyon
Mobile app client:	www.github.com/andergisomon/mantadsodu

Appendix B

Screenshots of Parts of the System

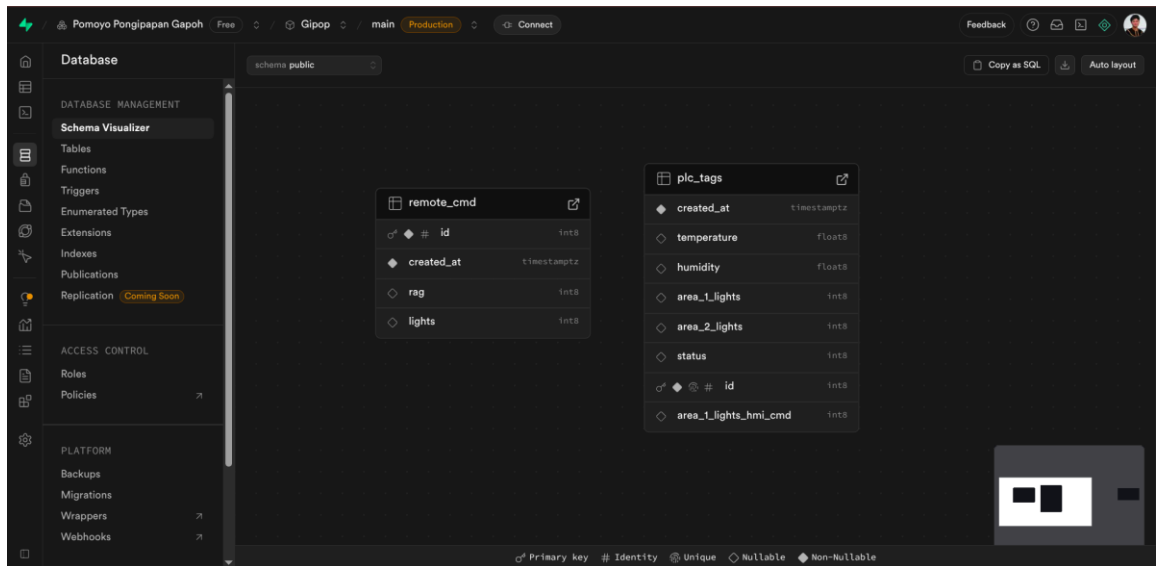


Figure B1: Supabase Console.

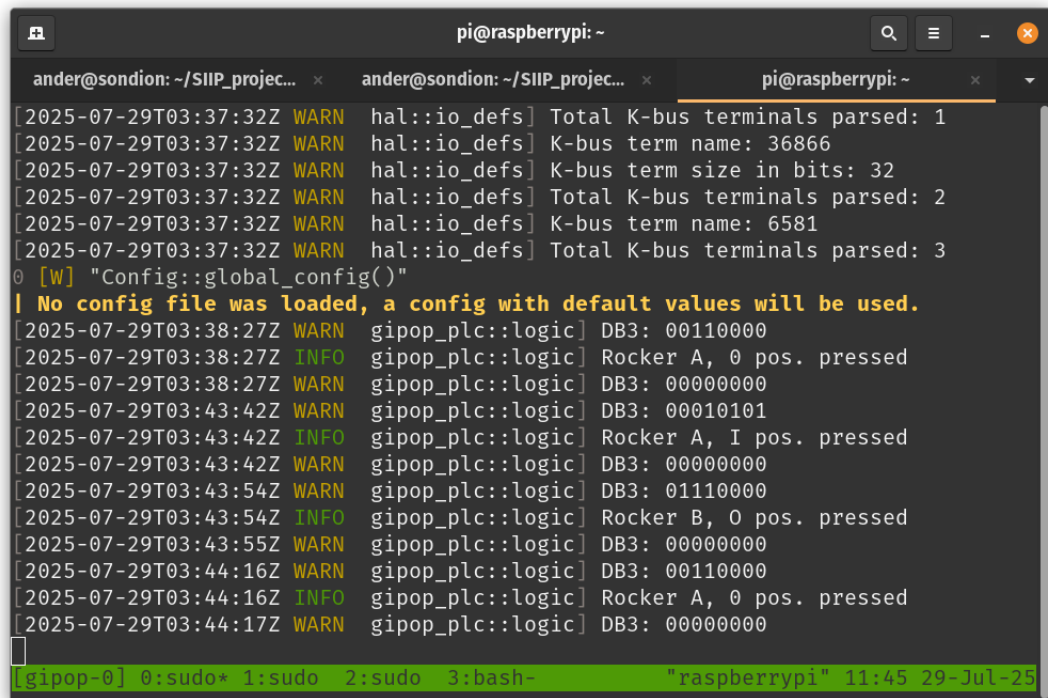


Figure B2: PLC EtherCAT and control loop process.

```
pi@raspberrypi: ~
ander@sondion: ~/SIIP_projec... x ander@sondion: ~/SIIP_projec... x pi@raspberrypi: ~
WARNING:supabase:Fetching remote_cmd table
WARNING:supabase:Fetched 0
WARNING:supabase:Fetched 0
WARNING:asyncua:Sending rmt_cmd_rag_node 0
WARNING:asyncua:Sending rmt_cmd_area_2_lights_node 0
WARNING:supabase:Fetching remote_cmd table
WARNING:supabase:Fetched 0
WARNING:supabase:Fetched 0
WARNING:asyncua:Sending rmt_cmd_rag_node 0
WARNING:asyncua:Sending rmt_cmd_area_2_lights_node 0
WARNING:supabase:Fetching remote_cmd table
WARNING:supabase:Fetched 0
WARNING:supabase:Fetched 0
WARNING:asyncua:Sending rmt_cmd_rag_node 0
WARNING:asyncua:Sending rmt_cmd_area_2_lights_node 0
WARNING:supabase:Fetching remote_cmd table
WARNING:supabase:Fetched 0
WARNING:supabase:Fetched 0
WARNING:asyncua:Sending rmt_cmd_rag_node 0
WARNING:asyncua:Sending rmt_cmd_area_2_lights_node 0
WARNING:supabase:Fetching remote_cmd table
WARNING:supabase:Fetched 0
WARNING:supabase:Fetched 0
WARNING:asyncua:Sending rmt_cmd_rag_node 0
WARNING:asyncua:Sending rmt_cmd_area_2_lights_node 0
[gipop-0] 0:sudo 1:sudo 2:sudo- 3:./bin/python* "raspberrypi" 11:46 29-Jul-25
```

Figure B3: Remote command Supabase-OPC UA gateway process.

```
pi@raspberrypi: ~
ander@sondion: ~/SIIP_projec... x ander@sondion: ~/SIIP_projec... x pi@raspberrypi: ~
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] AI0: 4.589, DI0: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
[2025-07-29T03:38:06Z INFO modbus] [Via iceoryx2] Modbus D00: 0
^C
pi@raspberrypi:~/Gipop/modbus $ sudo RUST_LOG=warn ./modbus
[W] "Config::global_config()"
| No config file was loaded, a config with default values will be used.
[gipop-0] 0:sudo 1:sudo- 2:sudo* 3:bash "raspberrypi" 11:45 29-Jul-25
```

Figure B4: Modbus/TCP polling process.

```
pi@raspberrypi: ~  
ander@sondion: ~/SIIP_projec... x ander@sondion: ~/SIIP_projec... x pi@raspberrypi: ~  
pi@raspberrypi:~ $ cd Gipop/sunsuyon/toburus  
pi@raspberrypi:~/Gipop/sunsuyon/toburus $ ls  
example_c_publisher example_c_subscriber gipop_tgbot  
pi@raspberrypi:~/Gipop/sunsuyon/toburus $ sudo ./gipop_tgbot  
0 [W] "Config::global_config()"  
| No config file was loaded, a config with default values will be used.  
🤖 Bot @gipop_tgbot is online!  
info: Client identified successfully.  
info: received: 0  
info: received: 0  
info: received: 0  
info: received: 0  
info: received: 0  
info: received: 0  
info: received: 0  
info: received: 0  
info: received: 0  
info: received: 0  
info: received: 0  
info: received: 0  
[gipop-0] < 1:sudo- 2:sudo 3:./bin/python 4:sudo*"raspberrypi" 11:46 29-Jul-25
```

Figure B5: Telegram bot process.

```
pi@raspberrypi: ~  
ander@sondion: ~/SIIP_projec... x ander@sondion: ~/SIIP_projec... x pi@raspberrypi: ~  
[2025-07-29T03:45:31Z INFO opcua] [Modbus] AN0: 4.649, DI0: 0  
[2025-07-29T03:45:31Z INFO opcua] [Modbus] AN0: 4.649, DI0: 0  
[2025-07-29T03:45:31Z INFO opcua] [Via iceoryx2] temp: 23.245, humd: 36.302315,  
stat: 0, ar1:0, ar2:0  
[2025-07-29T03:45:31Z INFO opcua] [Via iceoryx2] temp: 23.245, humd: 36.291977,  
stat: 0, ar1:0, ar2:0  
[2025-07-29T03:45:31Z INFO opcua] [Modbus] AN0: 4.649, DI0: 0  
[2025-07-29T03:45:31Z INFO opcua] [Modbus] AN0: 4.649, DI0: 0  
[2025-07-29T03:45:31Z INFO opcua] [Via iceoryx2] temp: 23.245, humd: 36.286804,  
stat: 0, ar1:0, ar2:0  
[2025-07-29T03:45:31Z INFO opcua] [Via iceoryx2] temp: 23.245, humd: 36.286804,  
stat: 0, ar1:0, ar2:0  
[2025-07-29T03:45:31Z INFO opcua] [Modbus] AN0: 4.649, DI0: 0  
[2025-07-29T03:45:31Z INFO opcua] [Modbus] AN0: 4.649, DI0: 0  
[2025-07-29T03:45:32Z INFO opcua] [Via iceoryx2] temp: 23.245, humd: 36.284218,  
stat: 0, ar1:0, ar2:0  
[2025-07-29T03:45:32Z INFO opcua] [Via iceoryx2] temp: 23.245, humd: 36.276466,  
stat: 0, ar1:0, ar2:0  
[2025-07-29T03:45:32Z INFO opcua] [Modbus] AN0: 4.649, DI0: 0  
[2025-07-29T03:45:32Z INFO opcua] [Modbus] AN0: 4.649, DI0: 0  
[gipop-0] 0:sudo- 1:sudo* 2:sudo 3:bash "raspberrypi" 11:45 29-Jul-25
```

Figure B6: OPC UA server process.

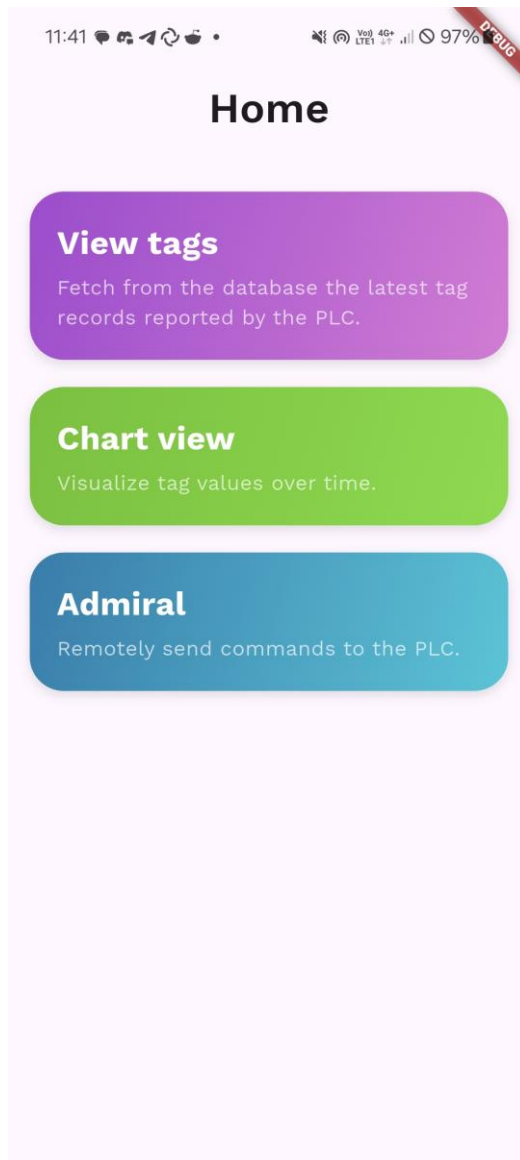


Figure B7: Mobile client app
Home screen.

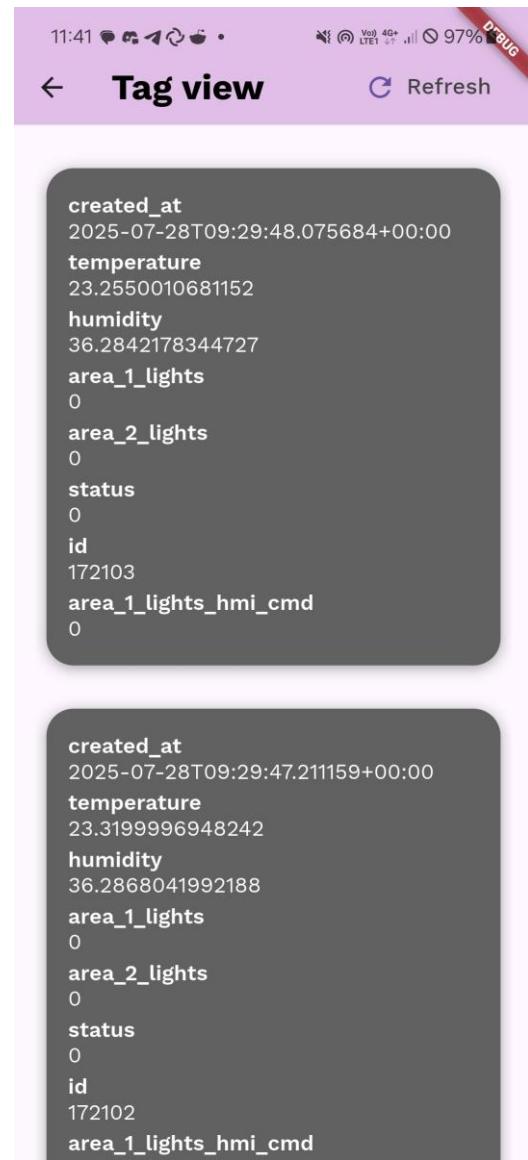


Figure B8: Mobile client app
Tag View screen.

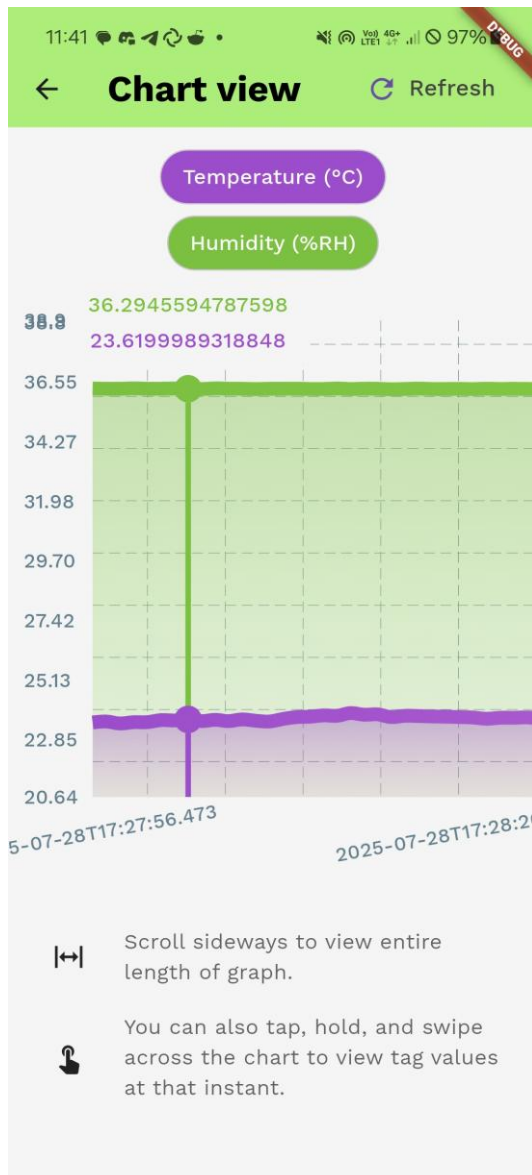


Figure B9: Mobile client app
Chart View screen.

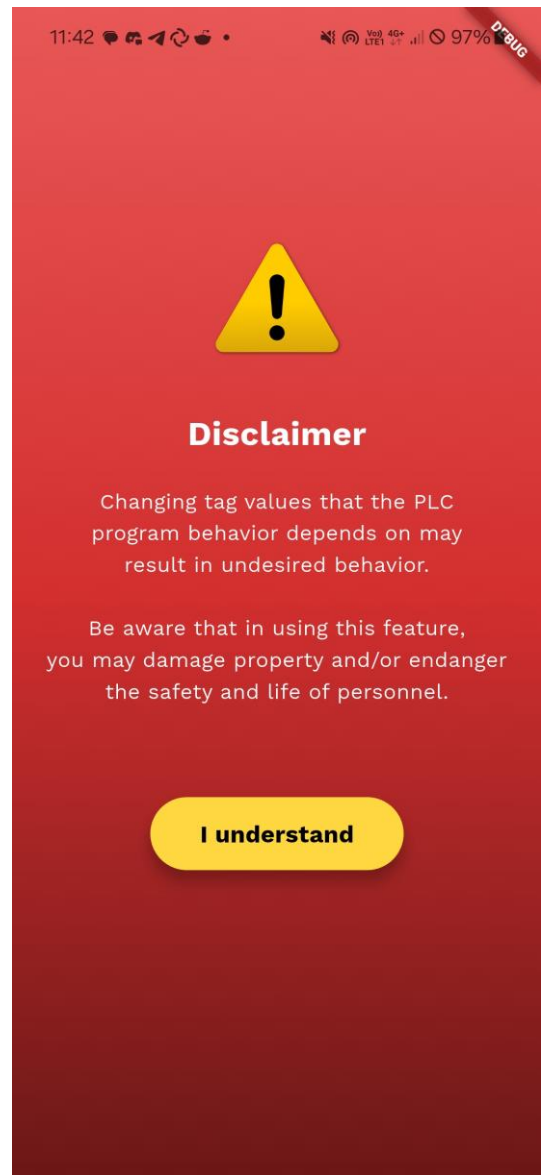


Figure B10: Mobile client app
Admiral disclaimer screen.

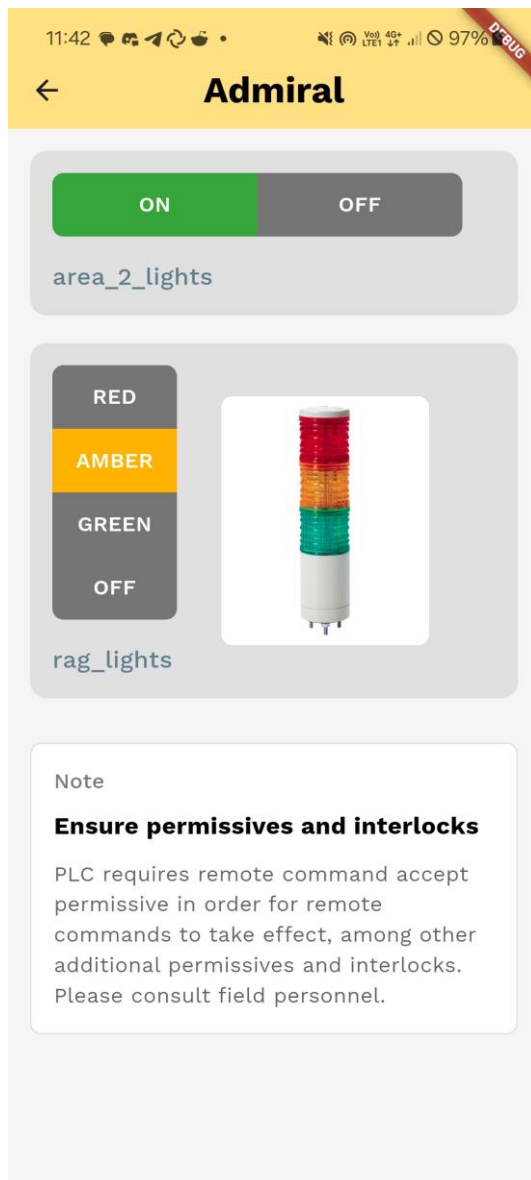


Figure B11: Mobile client app Admiral screen.

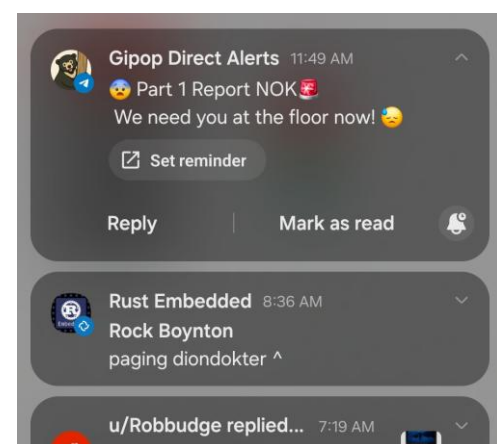
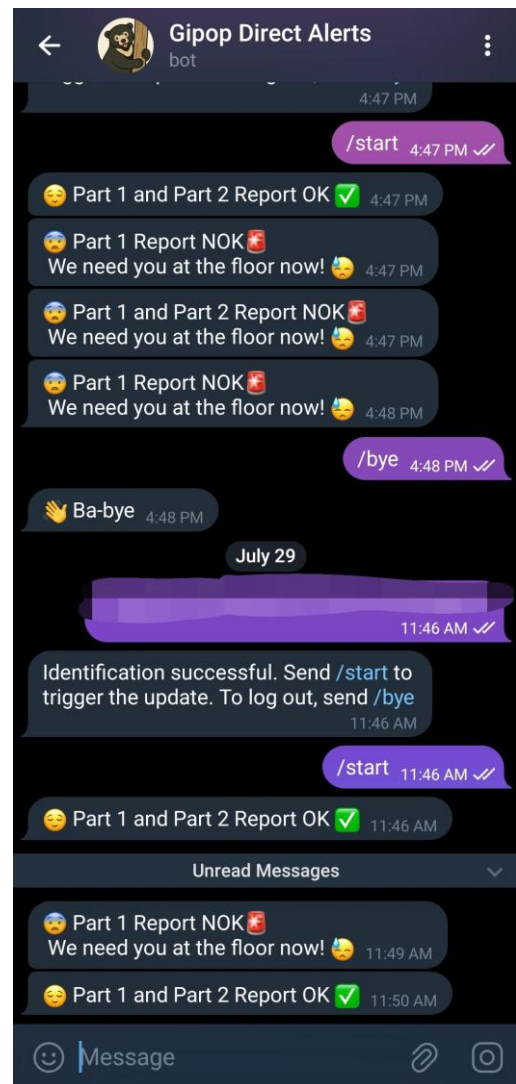


Figure B12: Telegram bot notification and chat screen on mobile.

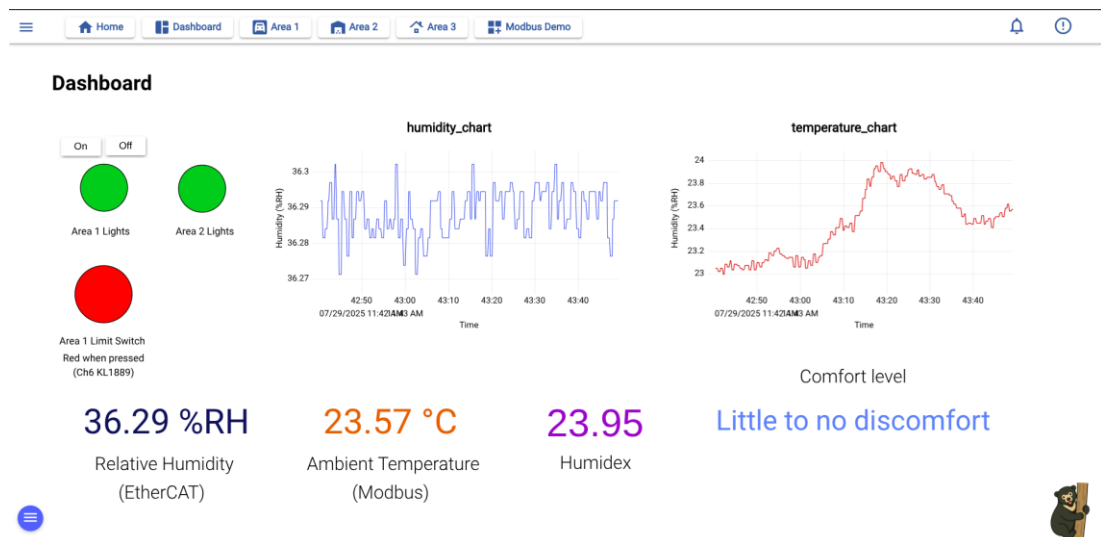


Figure B13: FUXA HMI Dashboard.

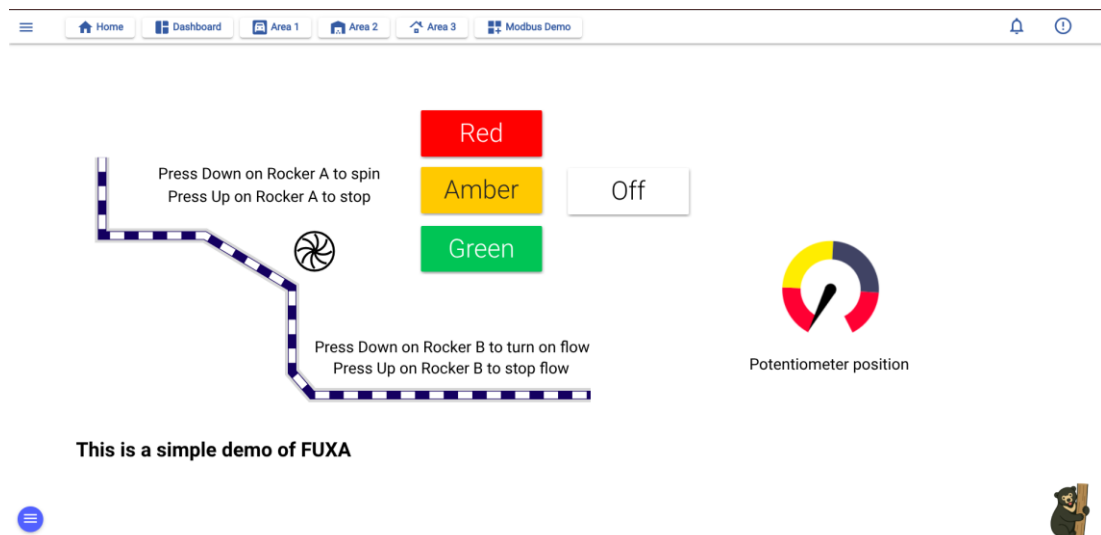


Figure B14: FUXA UI element demo.

Appendix C

Pictures of the Physical Test Bench

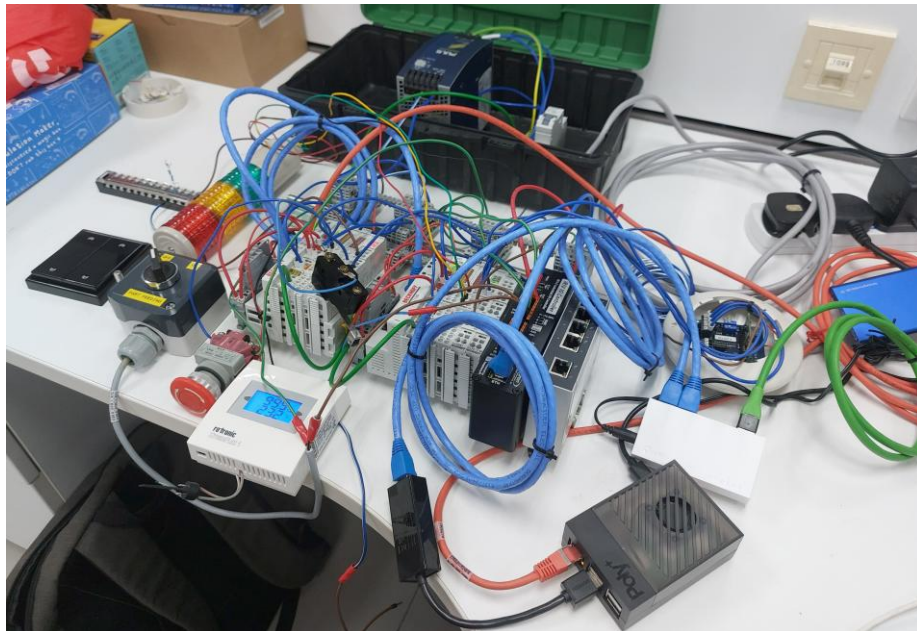


Figure C1: Test bench.

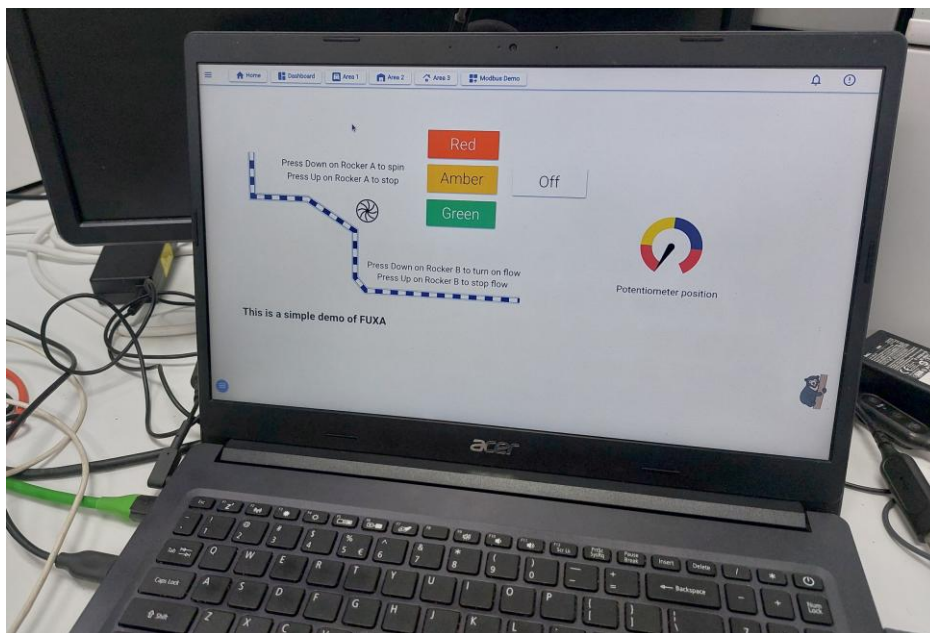


Figure C2: Development laptop acting as HMI, hosting FUXA, showing 'Home' screen.



Figure C3: Development laptop acting as HMI, hosting FUXA, showing 'Dashboard' screen.

Appendix D

Linux Kernel Build and Boot Configuration

Link to kernel .config: <https://gist.github.com/andergisomon/f3cd74ec9829d45029abc3a906f1b6d8#file-config>

Link to kernel boot parameters: <https://gist.github.com/andergisomon/f3cd74ec9829d45029abc3a906f1b6d8#file-kernel-boot-arguments>

Appendix E

Equipment Bill of Materials

The following BOM lists the equipment used for the functional demonstration of the integrated system. Consumables such as ferrules, fuses, resistors, and stranded copper wires are not included.

Item	Description	Manufacturer	Part Number
Network switch	5-port 10/100Mbps Desktop Switch	TP-Link	LS1005
Non-RT NIC	USB to Gigabit Ethernet Adapter	UGREEN	CR111
Power supply	DIN rail power supplies for 1-phase system 24 V, 10 A	PULS	QS10.241
EK1100	EtherCAT Coupler	Beckhoff	EK1100
BK1120	EtherCAT Bus Coupler for standard Bus Terminals	Beckhoff	BK1120
IRIV IO	Modbus/TCP Remote I/O Extender	Cytron	IRIV-IOC
EL1889	EtherCAT Terminal, 16-channel digital input, 24 V DC, 3 ms, ground switching	Beckhoff	EL1889
EL2889	EtherCAT Terminal, 16-channel digital output, 24 V DC, 0.5 A, ground switching	Beckhoff	EL2889
EL3024	EtherCAT Terminal, 4-channel analog input, current, 4...20 mA, 12 bit, differential	Beckhoff	EL3024
KL1889	Bus Terminal, 16-channel digital input, 24 V DC, 3 ms, ground switching	Beckhoff	KL1889
KL2889	Bus Terminal, 16-channel digital output, 24 V DC, 0.5 A, ground switching	Beckhoff	KL2889
KL6581	Bus Terminal, 1-channel communication interface, EnOcean®, master	Beckhoff	KL6581
KL6583	EnOcean®, radio transceiver, for KL6581	Beckhoff	KL6583
PTM200	Push button Transmitter Device	EnOcean	PTM200
PLC	Raspberry Pi 5 2 GB BCM2712 2.4GHz Single Board Computer	Raspberry Pi	Pi 5 2GB
Local HMI	Development Laptop	Acer	Aspire A315-57G
Remote HMI	Smartphone	Samsung	SM-A528B/DS
Humidity Temperature Sensor	Digital transmitter for humidity & temperature: Space version	Rotronic	HF135-SB1XDXXX
Tower Light	AC/DC 24V, Ø45mm Pole Mount Type LED Steady Tower. Red, Amber, Green	Qlight	ST45B-3-24-RAG
Circuit breaker	Miniature circuit breaker 230/400 V 6kA, 1-pole, C, 6 A, D=70 mm	Siemens	5SY6106-7