FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Design and Experimental Validation of a Specification-Oriented IO-Link Device Stack

Manuel Lima Brites Kankura Salazar



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Paulo José Lopes Machado Portugal

Feb 28, 2026

© Manuel Lima Brites Kankura Salazar, 2026>

Design and Experimental Validation of a Specification-Oriented IO-Link Device Stack

Manuel Lima Brites Kankura Salazar

Mestrado em Engenharia Eletrotécnica e de Computadores

Approved in oral examination by the committee:

President: Prof. Hélio Mendes de Sousa Mendonça
Examiner: Prof. Mário Jorge De Andrade Ferreira Alves
Member: Prof. Paulo José Lopes Machado Portugal

Feb 28, 2026

Resumo

Esta dissertação apresenta o desenvolvimento e a validação experimental de uma *stack* IO-Link orientada à especificação (versão 1.1.4), implementada em `C++` para o microcontrolador RP2040. O trabalho enquadra-se no paradigma do *retrofit* industrial, no qual sensores legados — analógicos ou digitais — carecem de integração em sistemas de automação modernos através de comunicação digital normalizada, evitando assim a substituição integral do equipamento existente. A solução proposta visa disponibilizar uma *stack* de código aberto (*open-source*), estruturalmente transparente e experimentalmente verificável. Para o efeito, adotou-se como ponto de partida a implementação *open-source lwIOLink*, cuja base apresentava uma cobertura funcional limitada e ausência de rastreabilidade explícita face à norma. A partir desta fundação, o *software* foi sistematicamente analisado, corrigido, expandido e reestruturado, resultando num subconjunto de comportamentos protocológicos rigorosamente delimitado, observável e tecnicamente rastreável.

A implementação opera em modo de transmissão COM2 e foi validada num ambiente experimental controlado, recorrendo a uma ligação UART de 3.3 V em modo *half-duplex*, configurada para 38.4 kbaud com codificação 8E1. Esta abordagem arquitetual permitiu isolar o comportamento lógico da camada de ligação de dados (*data link layer*) face aos constrangimentos elétricos inerentes à camada física, suprimindo o impacto de variáveis como transitórios de tensão, ruído eletromagnético e latências de comutação típicas dos *transceivers* industriais de 24 V. A arquitetura modela explicitamente as máquinas de estados operacionais do IO-Link (*Startup*, *Preoperate* e *Operate*), assegurando que o provimento de serviços está condicionado ao estado de execução do dispositivo. O processamento da comunicação é assegurado através do *parsing* de *M-sequences*, com validação de integridade baseada na soma de verificação (*checksum*) CK6. O sistema suporta transações ISDU do tipo *on-request* (Type 0), incluindo o acesso indexado e o respetivo reencaminhamento lógico através de *callbacks* aplicacionais, impondo uma limitação estrita à dimensão do *payload* e garantindo um comportamento determinístico perante pedidos de dimensão anómala. Adicionalmente, implementou-se um mecanismo de persistência de parâmetros com verificação de integridade em memória não-volátil baseada em CRC32, bem como a sinalização de eventos de diagnóstico através do canal de *Diagnosis*, acomodando os eventos associados ao fluxo de *Data Storage*. Todas as estruturas de dados e *buffers* são alocadas estaticamente, garantindo a contenção da pegada de memória e impedindo a ocorrência de alocação dinâmica em tempo de execução.

A validação experimental foi conduzida com recurso a um *Master* IO-Link dedicado, desenvolvido especificamente em Python para o efeito, o qual se revelou capaz de executar o processo de *startup handshake* no âmbito definido, realizar trocas de dados cíclicas e acíclicas, injetar anomalias (*fault injection*) — tais como CK6 inválido, truncação de tramas e pedidos fora de limites — e capturar os respetivos traços de execução para posterior correlação. Os ensaios documentam o estabelecimento de uma comunicação COM2 estável, transições de modo operacionais coerentes, o fornecimento apropriado de serviços e uma capacidade de recuperação previsível e determinística perante condições de erro e ciclos de reinicialização elétrica.

As fronteiras e limitações do protótipo desenvolvido encontram-se inequivocamente demarcadas. Sublinha-se que a validação empírica desconsiderou a adoção de uma camada física industrial a 24 V, não tendo sido implementados os mecanismos associados a transferências ISDU segmentadas (Type 1 e Type 2). O trabalho de desenvolvimento não foi sujeito a qualquer processo formal de certificação IEC 61131-9 ou a testes de conformidade validados pelo *IO-Link Consortium*; por conseguinte, as afirmações de conformidade circunscrevem-se ao comportamento protocológico comprovado experimentalmente através dos referidos traços de execução, dentro do estrito subconjunto abordado. Em suma, esta dissertação estabelece um alicerce de *software* robusto, transparente e experimentalmente escrutinável, constituindo uma plataforma viável para investigação, pedagogia e futuras iterações que ambicionem uma integração industrial plena.

Palavras-chave: IO-Link • RP2040 • sistemas embebidos • comunicação industrial • retrofit • device stack • validação

Abstract

This thesis presents the development and experimental validation of a specification-oriented IO-Link V1.1.4 device stack implemented in C++ for the RP2040 microcontroller. The work addresses industrial retrofit scenarios in which conventional sensors — including analog devices and non-IO-Link digital sensors — require standardized digital communication without full replacement of the existing hardware. The proposed solution delivers a transparent and experimentally verifiable open-source stack, built upon a pre-existing open-source implementation that exhibited limited functional coverage and lacked explicit traceability to the normative specification. Starting from this baseline, the stack was systematically analysed, corrected, extended, and restructured, resulting in a clearly delimited and technically traceable implementation.

The implemented stack operates in COM2 mode and was validated using a controlled experimental setup based on a UART connection configured for 38.4 kbaud with 8E1 framing in half-duplex operation. This was intentionally adopted to isolate data link layer behavior from electrical-layer variability associated with the 24 V physical interface. The IO-Link operational modes (Startup, Preoperate, and Operate) are explicitly modeled, and service availability is gated according to the active mode. Communication is processed through M-sequence parsing with CK6 checksum validation. The stack supports bounded Type-0 (on-request) ISDU transactions, including indexed read and write access via application callbacks, and enforces explicit payload limits to preserve deterministic memory behavior. A subset of identification parameters is persisted in non-volatile memory with CRC32-based integrity verification. Diagnostic events are generated and delivered through the Diagnosis channel, including events associated with the Data Storage workflow exercised during validation. All protocol buffers and data structures are statically allocated, avoiding dynamic memory allocation at runtime.

Validation was performed using a custom Python-based IO-Link master capable of executing the startup handshake within the defined scope, performing cyclic and acyclic exchanges, injecting malformed or oversized requests, and capturing protocol traces for deterministic analysis. A mock UART backend was additionally implemented to enable repeatable, hardware-independent end-to-end testing. Within the evaluated environment, experimental results demonstrate stable COM2 communication, correct state-dependent service gating, deterministic handling of malformed frames, bounded ISDU behavior, and reliable parameter restoration across power cycles.

The scope and limitations of the prototype are explicitly defined. The final validation setup does not include a 24 V industrial physical layer, segmented ISDU transfers (Type-1/Type-2) are not implemented, and no formal IEC 61131-9 certification or IO-Link Consortium conformance testing was performed. Compliance claims are therefore limited to trace-validated behavioral alignment within the explicitly implemented data link layer subset.

The presented work provides an open, transparent, and experimentally verifiable foundation for IO-Link device research and education, and establishes a structured basis for future extension toward full industrial reintegration.

Keywords: IO-Link • RP2040 • embedded systems • industrial communication • retrofit • device stack • validation

The United Nations Sustainable Development Goals

The United Nations **SDG!**s (**SDG!**s) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals (https://sdgs.un.org/goals) to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This dissertation aligns with these global efforts by proposing digital retrofitting strategies for industrial automation. By integrating conventional analog and digital sensors into modern Industry 4.0 architectures via a specification-oriented IO-Link stack, this work promotes sustainable industrial upgrading and reduces the electronic waste associated with the premature disposal of functional equipment.

The specific Sustainable Development Goals directly supported by this research are:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.

SDG 12 Ensure sustainable consumption and production patterns.

SDG	Target	Contribution	Performance Indicators and Metrics
9	9.4	The developed IO-Link device stack facilitates the digital retrofitting of legacy industrial infrastructure, enabling modern digital capabilities without requiring complete hardware replacement.	Availability of accessible, open-source validation tools for industrial communication upgrades.
12	12.5	By providing a standardized digital interface for conventional sensors, this work promotes the reuse of existing functional equipment, directly mitigating industrial electronic waste (e-waste).	Extension of the operational lifecycle for conventional sensors in modernized automation deployments.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Paulo José Lopes Machado Portugal, for the guidance, feedback, and support throughout this work. I also thank the Faculty of Engineering of the University of Porto for providing the necessary resources and technical environment.

I am grateful to my colleagues and friends for their helpful discussions and encouragement. Finally, I would like to thank my family for their continued support and motivation during the completion of this thesis.

A special thank you goes out to my dearest Sarah, whose words of inspiration and unyielding belief in my abilities continuously pushed me to elevate my work beyond my own expectations.

Manuel K. Salazar

Declaração de Honra

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.

I also declare that I have used Artificial Intelligence tools to complete part(s) of this dissertation, and that this use and its results are duly noted and have been carefully checked for errors, inaccuracies, unfounded attributions or other forms of bias in content and arguments, as well as determining authorship.

Manuel Lima Brites Kankura Salazar

Manuel K. Salazar

"In theory, there is no difference between theory and practice.
In practice, there is."

Yogi Berra

Contents

Resumo i

Abstract iii

Acknowledgements vi

1 Introduction 1
- 1.1 Context and Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Objectives . . . 3
- 1.4 Research Methodology and Approach . . . 4
- 1.5 Main Contributions . . . 4
- 1.6 Work Plan and Timeline . . . 5
- 1.7 Document Structure . . . 6

2 Background 7
- 2.1 Evolution of Industrial Sensor Communication . . . 7
- 2.2 Overview of the IO-Link Protocol . . . 8
- 2.3 Physical Layer and Signaling . . . 9
  - 2.3.1 Asynchronous Serial Transmission and Framing . . . 10
  - 2.3.2 Wake-up Signaling . . . 11
- 2.4 Data Link Layer Fundamentals: M-sequences, Timing, and CK6 . . . 11
  - 2.4.1 M-sequences and Frame Structure . . . 11
  - 2.4.2 Timing Constraints . . . 12
  - 2.4.3 CK6 Checksum Validation . . . 12
- 2.5 Operational Model and Startup Procedure . . . 12
  - 2.5.1 Communication Modes: COM1, COM2, and COM3 . . . 12
  - 2.5.2 Startup Sequence and COM Negotiation . . . 13
- 2.6 Data Types, Services, and State-Dependent Availability . . . 14
  - 2.6.1 Operational States and Service Availability . . . 15
  - 2.6.2 Cyclic data exchange: Process Data and On-request Data . . . 16
  - 2.6.3 Representative frame structures and message examples . . . 17
  - 2.6.4 Acyclic services: Parameter access (ISDU) and diagnostic events . . . 19
  - 2.6.5 Diagnosis and event signaling . . . 20
- 2.7 IO Device Description (IODD) . . . 21
  - 2.7.1 Role of the IODD . . . 21
  - 2.7.2 IODD Generation and Practical Considerations . . . 22

3 State of the Art **23**
3.1 Introduction . . . 23
3.2 Sensor Retrofit in Industrial Automation . . . 23
3.3 Digital Communication at the Sensor Level . . . 24
3.4 Commercial IO-Link Retrofit Solutions . . . 25
3.5 Academic and Open Research on IO-Link . . . 25
3.5.1 IO-Link Master Implementations and Bridging . . . 25
3.5.2 System Integration and Industry 4.0 . . . 26
3.5.3 IO-Link Wireless and Sensor Extensions . . . 26
3.5.4 Summary of Identified Gaps . . . 26
3.6 Open IO-Link Device Stack Implementations . . . 27
3.7 Summary of Identified Gaps and Positioning . . . 27

4 Architecture of the SpecIOLink-Device Stack **30**
4.1 Introduction . . . 30
4.1.1 Normative Context . . . 30
4.1.2 Baseline and Architectural Contribution . . . 31
4.2 Architectural Scope and Boundaries . . . 31
4.3 Layered Architecture . . . 32
4.3.1 Transport Interface Layer . . . 33
4.3.2 Protocol Core Layer . . . 33
4.3.3 Application Interface Layer . . . 33
4.3.4 Diagnostic Event Subsystem . . . 33
4.3.5 Persistence Subsystem . . . 34
4.4 Device Core Structural Model . . . 34
4.4.1 Core Architectural Components . . . 34
4.4.2 Static Allocation and Determinism . . . 36
4.5 State Modelling Architecture . . . 36
4.5.1 Runtime Control State . . . 37
4.5.2 Operational Mode State . . . 38
4.5.3 Architectural Separation Rationale . . . 38
4.5.4 Deferred Mode Transition Mechanism . . . 38
4.6 Message Processing Architecture . . . 38
4.6.1 Frame Acquisition . . . 39
4.6.2 Structural Validation . . . 39
4.6.3 Service Dispatch . . . 40
4.6.4 Response Construction . . . 40
4.7 ISDU Architectural Model . . . 40
4.7.1 Architectural Scope of ISDU Support . . . 40
4.7.2 Bounded Payload Strategy . . . 41
4.7.3 Callback-Based Parameter Access . . . 41
4.7.4 Rationale for Excluding Segmentation . . . 41
4.8 Event and Persistence Architecture . . . 41
4.8.1 Diagnostic Event Architecture . . . 42
4.8.2 Persistence Architecture . . . 43
4.9 Architectural Constraints and Declared Limitations . . . 45
4.9.1 Excluded Features . . . 45
4.9.2 COMLOST and Wake-Up Handling . . . 45
4.9.3 Physical Layer Abstraction . . . 45

4.9.4 Compliance Statement 46
4.10 Summary 46

5 Implementation and Validation **47**
5.1 Introduction 47
5.1.1 Validation Methodology and Hardware Evolution 47
5.2 lwIOLink Stack Implementation 48
5.2.1 Core Library Structure 48
5.2.2 Operational Mode Handling and Frame Validation 49
5.2.3 ISDU Handling and Transport Modes 50
5.2.4 Persistent Parameter Storage 50
5.2.5 Event Generation and Diagnosis Handling 50
5.2.6 Safety Mechanisms and ISDU Payload Bounding 52
5.3 Validation Setup and Methodology 53
5.3.1 Custom Python-Based IO-Link Master 53
5.3.2 Test Mechanisms and Instrumentation 53
5.3.3 Test Methodology 54
5.3.4 Normative Validation Logic 54
5.4 Validation Results 56
5.4.1 Test 1: Basic Communication (Startup & Cyclic) 56
5.4.2 Test 2: Data Persistence (Write Tag → Reset → Read) 58
5.4.3 Test 2: Persistence and *Data Storage* Verification 58
5.4.4 Test 3: Factory Reset (System Command 0x82) 62
5.4.5 Test 4: End-to-End Subset Validation (Persistence + Error Handling) . . . 64
5.4.6 Test 5: Advanced Features (Page 2, CRC32, Truncation) 66
5.4.7 Test 6: Compliant Event & *Data Storage* Flow (Diag + DS) 68
5.4.8 Test 6: Fault Injection and Event Handling 68
5.4.9 Test 6: ISDU Payload Bounding Behavior 70
5.5 Discussion 70
5.6 Summary 71

6 Conclusions and Future Work **73**
6.1 Conclusions 73
6.2 Limitations 74
6.3 Future Work 75

References **76**

List of Figures

2.1 Typical IO-Link system configuration. The IO-Link Master manages point-to-point communication with multiple Devices and provides a gateway to higher-level control systems. 8
2.2 Simplified IO-Link communication stack illustrating the separation between application logic, protocol handling, and physical transmission. 9
2.3 IO-Link three-wire single-drop connection. L+ and L- provide device power, while C/Q operates either as a communication line (COMM) or as a conventional switching signal (SIO). The Master controls the link and communication is half-duplex. 10
2.4 Conceptual asynchronous serial frame structure used by IO-Link signaling (start bit, data bits, parity, stop bit). Exact framing parameters depend on the negotiated COM mode [4], [5]. 10
2.5 Wake-up and startup timing. Numerical constraints for wake-up duration, idle time, and response windows are defined by the IO-Link specification [4], [5]. . . 11
2.6 IO-Link M-sequence structure: a Master request followed by a Device response. Payload interpretation depends on operational mode and requested service; frames include protocol-level integrity information (CK6) and sequence control fields as defined by the specification [4], [5], [6]. 11
2.7 Conceptual Master-driven exchange: each request is followed by a Device response within a defined response window t_{RESP}. The specification defines the timing bounds required for deterministic operation [4], [5]. 12
2.8 Overview of a Master-driven IO-Link startup sequence. Wake-up may precede capability discovery and COM selection; operational state transitions enable cyclic and acyclic services as specified by IO-Link [4], [5]. 14
2.9 Overview of IO-Link data types and services: cyclic process data exchange, an optional on-request data (OD) slot within the M-sequence, acyclic ISDU access, and diagnostic event signaling between Master and Device. 15
2.10 Simplified IO-Link operational state progression (Startup, Preoperate, Operate) and typical service availability per state. Exact transitions and command sequencing are defined by the specification. 15
2.11 Conceptual relationship between IO-Link operational states and service availability. ✓ indicates that a service category is available/allowed in the corresponding state, whereas ✗ indicates that it is not available/not allowed. Exact service constraints are defined by the specification [4], [5]. 16
2.12 Conceptual cyclic exchange showing Process Data and the optional On-request Data (OD) slot within a single M-sequence. The OD field enables parameter or diagnostic access without leaving cyclic operation [4]. 17

2.13 Observed high-level structure of representative Master→Device and Device→Master frames in the validation setup. The figure emphasizes the grouping of fields (MC, integrity/sequence, OD, PD, status) as used in the experimental logs. Exact normative definitions and constraints are given in [4], [5]. 18
2.14 Conceptual workflow for Type-0 ISDU transactions and diagnostic event retrieval. ISDU requests (Index/Subindex) are explicitly issued by the Master and completed by a single response. Diagnostic events follow the same Master-driven paradigm: the Device indicates pending diagnostic information within cyclic exchange, and the Master retrieves event details using defined diagnostic services. Normative field encodings and timing constraints are defined in [4], [5]. 20

4.1 Layered architecture of SpecIOLink-Device (conceptual view). The stack is partitioned into a transport interface (byte-level half-duplex communication), a protocol core implementing IO-Link Data Link semantics within the declared subset, an application interface exposing controlled integration hooks, and a persistence/diagnostic subsystem. This separation supports traceability and bounded embedded execution while keeping hardware binding concerns outside the protocol semantics [4], [5]. 32
4.2 UML structural class diagram of the SpecIOLink-Device core. The `Device` class centralizes protocol buffers and state machines and coordinates transport interaction, ISDU handling, diagnostic events, and persistence. Associations indicate ownership (composition) and usage relationships. Visibility markers follow UML conventions (+ public, private). 35
4.3 Dual-state modelling (runtime control view). Execution phases are clearly separated with bounded buffering and validation prior to semantic dispatch. Invalid or incomplete frames are dropped without semantic side-effects. 37
4.4 Architectural message processing pipeline (conceptual view). Incoming bytes are accumulated into a bounded buffer, the expected length is inferred from the Master Command field, and structural validation (including CK6) is performed before semantic dispatch. Only validated messages reach service handling and response construction, supporting deterministic behavior under malformed and truncated communication [4], [5]. 39
4.5 Bounded diagnostic event queue architecture (conceptual view). Events are stored in a fixed-capacity structure with severity-based prioritization and a defined overflow policy (reject-on-full). Event delivery is master-driven: pending diagnostics are indicated within cyclic exchange, and detailed event data is retrieved through diagnostic services, after which transmitted entries are removed from the queue [4], [5]. 42
4.6 Persistence and integrity verification architecture (conceptual view). Selected configuration parameters are stored as a fixed-layout persistent block together with a CRC32 checksum. During initialization, the block is loaded and verified; on mismatch, deterministic default values are restored. Persistence operations are architecturally isolated from the message processing pipeline to prevent unintended coupling with runtime control and operational mode logic. 44

5.1 ISDU service flow in the validation environment (IO-Link V1.1.4). The master issues a Type 0 on-request transaction; the device validates and decodes it, dispatches parameter semantics via a callback, and returns a success or error response. 51

5.2 Priority-based event handling mechanism implemented in the lwIOLink stack. Diagnostic events are queued and transmitted in order of severity, ensuring that critical conditions are reported first. 52
5.3 Validation harness overview illustrating deterministic frame generation, trace capture, and behavioral comparison. In this context, the "comparison" component denotes a rule-based expected-versus-observed trace checker (test oracle in the software-testing sense), implemented as part of the Python master's validation logic rather than an external certified conformance tool. 56
5.4 Test 1 (Reference): Startup handshake, mode transitions (Startup → Preoperate → Operate), and stable cyclic exchange under COM2 configuration. 57
5.5 Test 2 (Phase B): Type-0 ISDU write to Index 0x0018 (Application Tag) embedded in cyclic OD, followed by immediate readback from RAM. 59
5.6 Test 2 (Phase C): System reset (System Command 0x80) and reconnection sequence. The Startup handshake is treated as reference (see Fig. 5.4). 60
5.7 Test 2 (Phase D): Post-reset ISDU read of Index 0x0018 confirming non-volatile persistence of the Application Tag. 62
5.8 Test 3 (Phase A): Factory reset via System Command 0x82. 63
5.9 Test 3 (Phase B): Reconnection attempts during Startup after factory reset. 63
5.10 Test 3 (Phase C): Verification of default Application Tag after factory reset. . . . 64
5.11 Test 4 (Phase B): Error handling validation via ISDU read of an unsupported index (e.g., 0x5000) and verification of the returned error indication within the implemented subset. Persistence cycle is treated as a reference (see Test 2, Section 5.4.2). 65
5.12 Test 5 (Phase A): Page 2 identification reads (addresses 0x10–0x1F). 66
5.13 Test 5 (Phase B): CRC32 verification (Index 0x001A) before and after a persisted parameter update. 67
5.14 Test 5 (Phase C): ISDU truncation behavior for Vendor Name (Index 0x0010) under fixed OD size constraint. 68
5.15 Event generation and *Data Storage* trigger validation flow (Test 6). 69

List of Tables

1.1 Project Work Plan and Timeline 5

3.1 Structured comparison between commercial IO-Link retrofit converters, open-source device stacks, and the specification-oriented approach adopted in this thesis. 28

5.1 Mapping between implemented mechanisms and validated IO-Link V1.1.4 behavioral requirements (data link layer subset). 55
5.2 Trace-based validation of ISDU write, reset cycle, and non-volatile persistence (Test 2). 61
5.3 Trace-based validation of diagnostic event handling and *Data Storage* trigger (Test 6). 70

List of Acronyms

EEPROM Electrically Erasable Programmable Read-Only Memory

OPC UA Open Platform Communications Unified Architecture

RP2040 Raspberry Pi Pico microcontroller (RP2040)

UART Universal Asynchronous Receiver/Transmitter

SDCI Single-Drop Digital Communication Interface

COM3 IO-Link Communication Mode 3 (230.4 kbaud)

COM2 IO-Link Communication Mode 2 (38.4 kbaud)

COM1 IO-Link Communication Mode 1 (4.8 kbaud)

IEC International Electrotechnical Commission

IEC61131-9 IEC 61131-9 (IO-Link standard)

API Application Programming Interface

CRC32 32-bit Cyclic Redundancy Check

PDOut Process Data Out (Master→Device)

PDIn Process Data In (Device→Master)

PLC Programmable Logic Controller

EMI Electromagnetic Interference

XML eXtensible Markup Language

ISDU Indexed Service Data Unit

CK6 6-bit Checksum (IO-Link)

CRC Cyclic Redundancy Check

IODD IO Device Description

SIO Standard Input/Output

ADI Analog Devices, Inc.

MCU Microcontroller Unit

RAM Random Access Memory

NVM Non-Volatile Memory

DLL Data Link Layer

PHY Physical Layer

OD On-request Data

MC Master Command

PD Process Data

DS Data Storage

TX Transmit

RX Receive

Chapter 1

Introduction

1.1 Context and Motivation

The digital transformation of industrial production systems, commonly designated as Industry 4.0, promotes the integration of communication, computation, and data analytics across all layers of automation architectures [1]. A fundamental requirement of this paradigm is the availability of structured, traceable, and interoperable data originating from sensors and actuators deployed at the factory floor level.

Despite this evolution, a large portion of currently installed industrial sensors do not natively implement standardized digital communication interfaces such as IO-Link. Many devices provide only conventional electrical outputs, including analog signals (e.g., 0 V to 10 V, 4 mA to 20 mA) [1], [2] or discrete digital switching outputs (e.g., 24 V PNP/NPN signals).

While these interfaces are robust and widely adopted in industrial environments [3], they are limited to transmitting process values or basic switching states. They do not provide standardized mechanisms for structured parameterization, device identification, diagnostic event reporting, or lifecycle data exchange as defined in IO-Link [4], [5].

Consequently, the challenge addressed in this work is not the conversion of analog signals per se, but the structured integration of conventionally interfaced sensors — analog or digital — into IO-Link-based communication architectures through an embedded device-side protocol stack.

Replacing existing sensing hardware with fully digital, IO-Link-enabled smart devices is often economically unjustifiable in brownfield industrial environments. Such environments are typically characterized by long asset lifecycles, high downtime costs, certification constraints, and limited documentation of legacy systems [4], [5]. Consequently, retrofit strategies that preserve existing sensing elements while introducing standardized digital communication capabilities have become increasingly relevant.

IO-Link, standardized as IEC 61131-9 [8] and maintained by the IO-Link Consortium [6], offers a point-to-point, master-driven communication interface specifically designed for sensor- and actuator-level integration. It provides deterministic cyclic and acyclic services, structured parameter access via ISDU mechanisms, and standardized diagnostic event signaling while maintaining

comparatively low implementation complexity. These properties position IO-Link as a suitable candidate for embedded retrofit experimentation and device-level digital augmentation.

In this work, IO-Link is approached from a protocol-engineering and validation perspective rather than from an industrial certification standpoint. The focus lies on the transparent, open-source implementation and experimental validation of an IO-Link V1.1.4 device stack, with explicit delimitation of the implemented data link layer scope and traceable alignment to the normative specification [4], [5].

1.2 Problem Statement

Although commercially certified IO-Link devices and retrofit converters are widely available [9], their firmware architectures are typically proprietary and closed. As a result:

- internal operational state transitions are not externally observable;
- traceability between specification clauses and implementation logic is not publicly documented;
- systematic protocol-level fault injection is generally not possible.

Conversely, open-source IO-Link device implementations are available, such as the lightweight lwIOLink project[1], which provides a minimal device-side implementation intended primarily for experimentation and prototyping.

While such projects offer valuable starting points, they often provide limited coverage of the IO-Link specification, lack explicit documentation of state-machine behavior, and do not clearly delimit the implemented subset of normative functionality. Common limitations include simplified startup handling, incomplete ISDU (Indexed Service Data Unit) support for structured acyclic parameter access [6], absence of explicit bounded memory guarantees, and limited documentation of failure modes.

This creates a structural gap between industrially mature but opaque implementations and academically accessible but often simplified prototypes. There is limited availability of openly inspectable IO-Link device stacks that simultaneously provide:

- explicit operational state modeling (Startup, Preoperate, Operate);
- deterministic and statically bounded memory behavior;
- structured ISDU handling with well-defined scope boundaries;
- documented alignment with IO-Link V1.1.4 data link layer requirements;
- controlled and reproducible validation workflows.

[1] https://github.com/unref-ptr/lwIOLink

Addressing this gap requires not only extending existing lightweight implementations but also clearly delimiting implementation scope and experimentally validating behavioral alignment with the normative specification.

1.3 Objectives

The primary objective of this dissertation is the extension, restructuring, and experimental validation of a specification-oriented IO-Link V1.1.4 device stack with a clearly defined data link layer scope.

The implementation targets low-cost embedded platforms and retrofit-oriented experimentation rather than full industrial certification or complete ecosystem integration.

More specifically, the objectives are:

- To systematically analyze, restructure, and extend the lwIOLink open-source device stack, transforming it from a lightweight prototyping implementation into a specification-oriented and experimentally validated IO-Link device stack;
- To implement an open-source IO-Link V1.1.4 data link layer subset with explicitly defined resource limits and clearly documented behavioral scope, including:
  - COM2 communication mode (38.4 kbaud);
  - M-sequence parsing with CK6 checksum validation;
  - state-dependent service gating;
  - Type-0 (on-request) ISDU transactions with bounded payload size;
  - diagnostic event signaling with priority handling;
  - CRC32-protected non-volatile parameter persistence;
- To explicitly delimit the implemented subset of IO-Link V1.1.4 functionality, documenting excluded mechanisms such as segmented ISDU transfers (Type-1/Type-2) and COM3 operation;
- To design and implement a dedicated validation environment based on a custom Python IO-Link Master enabling deterministic startup execution, trace capture, malformed frame injection, oversized payload testing, and diagnostic workflow verification;
- To experimentally validate behavioral alignment with IO-Link V1.1.4 data link layer requirements within the explicitly defined implementation scope.

A central methodological contribution of this work is the staged validation of protocol logic using a transport-level abstraction. By temporarily replacing the industrial 24 V physical layer with a controlled 3.3 V UART pivot operating under COM2 framing parameters (8E1, half-duplex), data link behavior can be evaluated independently of transceiver-specific electrical effects. This

approach enables deterministic observation of protocol execution and systematic debugging prior to any future industrial reintegration.

1.4 Research Methodology and Approach

The development and validation of the stack followed an iterative, specification-driven methodology. The project began with the adoption of the open-source *lwIOLink* implementation as an architectural baseline. This baseline was systematically analyzed against the IEC 61131-9 specification to identify structural gaps, resulting in a targeted rewrite in C++ to introduce strict operational state modeling, bounded payload enforcement, and protocol-compliant cyclic/acyclic routines.

A critical phase of the methodology involved resolving hardware integration challenges. Initial validation attempts utilizing industrial 24 V physical transceivers revealed non-deterministic timing behaviors—specifically concerning Transmit Enable (TxEN) directional control and analog signal propagation—that obscured the software-level protocol observation.

To ensure academic rigor and isolate the Data Link Layer logic from physical layer disturbances, a deliberate methodological pivot was executed. The industrial transceivers were bypassed in favor of a direct 3.3 V UART link operating at COM2 framing parameters (8E1, half-duplex). Concurrently, a custom Python-based IO-Link Master was developed to run on a Raspberry Pi 4[2]. This setup allowed for absolute determinism in M-sequence generation, systematic fault injection, and trace-aligned validation of the device stack's behavior without interference from electrical-layer variables. The implementation then proceeded through an iterative cycle: standard analysis, C++ implementation, custom Python test generation, validation, and regression testing, culminating in the final robust stack.

1.5 Main Contributions

This dissertation contributes to the field of industrial embedded systems by bridging the gap between closed commercial retrofits and incomplete open-source prototypes. The main contributions are:

- **A Specification-Oriented IO-Link Device Stack:** An open-source, bounded, and heavily documented C++ device stack optimized for the RP2040 microcontroller. It features transparent operational state transitions, Type-0 ISDU mechanisms with explicit memory limits, and a CRC32-backed parameter persistence layer. The complete source code is publicly available in the project repository`https://github.com/Gitkankura/SpecIOLink-Device.git`.

[2]During the development of both the custom Python IO-Link Master and the C++ device stack, Large Language Models (LLMs) were utilized as coding assistants to expedite structural framing, syntax generation, and routine data parsing. All AI-generated logic was systematically reviewed, modified, and validated by the author, who assumes full responsibility for the final functional architecture and its alignment with the normative specification.

- **A Deterministic Python IO-Link Master Testing Suite:** A customized master implementation capable of executing exact startup handshakes, cyclic exchanges, and intentional fault injections (e.g., CK6 checksum errors, oversized payloads) to evaluate device robustness.
- **A Traceable Validation Framework:** A documented methodology for isolating and testing industrial Data Link Layer semantics independently of transceiver hardware constraints, providing a reproducible framework for academic research and education in industrial communications.

1.6 Work Plan and Timeline

The execution of this work spanned approximately five months, from October 2025 to late February 2026. The project was structured into successive phases encompassing literature review, hardware preparation, baseline analysis, iterative development, and final documentation.

The specific timeline of activities is summarized in Table 1.1. The shift to the Python-emulated Master occurred dynamically during the initial implementation phase to address unforeseen hardware non-determinism, ensuring the project remained on track for rigorous software validation.

Table 1.1: Project Work Plan and Timeline

Timeframe	Tasks & Milestones
Oct 2025 (Weeks 1–2)	**Research & State of the Art:** Familiarization with the IEC 61131-9 standard, IO-Link specifications, and review of existing retrofit solutions.
Oct 2025 (Weeks 3–4)	**Hardware Procurement:** Selection and ordering of the RP2040 microcontroller, transceivers, and necessary electronic components.
Nov 2025 (Weeks 1–2)	**Baseline Analysis:** In-depth review of the *lwIOLink* open-source stack to identify architectural gaps regarding standard compliance.
Nov 2025 (Week 3)	**Equipment Setup:** Hardware assembly, datasheet familiarization, and initial environmental configuration.
Nov – Dec 2025 (4 Weeks)	**Initial Testing & Methodological Pivot:** Implementation of Wake-up, Handshake, and Startup routines. Identification of transceiver-induced non-determinism, leading to the strategic decision to emulate the Master in Python via 3.3 V UART.
Dec 2025 – Feb 5, 2026	**Iterative Development & Validation:** Cyclical process of analyzing specific capabilities (e.g., ISDU, events), implementing in C++, writing Python Master test cases, and confirming robust operation.
Feb 6 – Feb 28, 2026	**Refinement & Documentation:** Final synthesis of experimental data, formatting traces, writing the manuscript, and final delivery of the dissertation.

1.7 Document Structure

Chapter 2 presents the technical background of IO-Link, including physical signaling concepts, M-sequence structure, timing constraints, CK6 checksum validation, operational state modeling, ISDU mechanisms, diagnostic workflows, and the role of the IODD.

Chapter 3 analyzes the state of the art in industrial retrofit solutions and IO-Link device stack implementations, identifying structural and methodological gaps that motivate the present work.

Chapter 4 describes the architecture of the implemented IO-Link device stack, detailing its layered firmware organization, runtime control flow, ISDU dispatch mechanism, event handling structure, and parameter persistence subsystem.

Chapter 5 presents the implementation and experimental validation methodology, including the staged hardware evolution, the direct UART validation pivot, normative behavioral mapping, trace-based verification, and controlled fault injection.

Finally, Chapter 6 summarizes the contributions, establishes traceability between requirements, design decisions, and validation evidence, discusses scope limitations, and outlines directions for future extension and industrial reintegration.

Chapter 2

Background

2.1 Evolution of Industrial Sensor Communication

Historically, industrial automation systems have relied on standardized electrical signal interfaces for the transmission of sensor measurements and status information. These interfaces include both analog signal ranges such as 0 V to 10 V and 4 mA to 20 mA, and discrete digital switching outputs commonly used at 24 V (e.g., PNP/NPN) for binary state reporting. Analog ranges became widely adopted due to their simplicity, robustness, and tolerance to electrical noise, while current-loop transmission has been extensively used in harsh industrial environments because of its resilience to line resistance and electromagnetic interference [1], [2], [3].

It is important to distinguish between the nature of a sensor and the physical representation used to transmit its output signal. Both analog and digital sensors may rely on analog electrical interfaces for signal transmission, including voltage- or current-based signaling. While such interfaces remain effective for conveying measured quantities, they inherently provide limited information content beyond the primary process value.

As industrial processes increased in complexity and scale, the need for additional information— such as device identification, configuration parameters, calibration data, and diagnostics— became increasingly evident. This limitation motivated the gradual adoption of digital communication technologies within industrial environments, enabling richer data exchange and standardized access to device-level information.

Early digital industrial communication solutions were largely based on low-level serial protocols such as Modbus (operating over RS-485) and HART, as well as more complex multi-drop fieldbus systems like PROFIBUS and CAN/CANopen. While these legacy systems enabled multi-device communication and more expressive data models [3], they often introduced significant complexity in terms of hardware requirements, configuration effort, and cost. As a result, they proved unsuitable for simple sensors or retrofit scenarios involving existing installations. The emergence of the Industry 4.0 paradigm, with its emphasis on flexibility, scalability, and data availability, further highlighted the need for lightweight and cost-effective digital communication

interfaces at the sensor level [7]. This context provided the foundation for the adoption of simpler, sensor-oriented communication protocols such as IO-Link.

2.2 Overview of the IO-Link Protocol

IO-Link emerged in the mid-2000s as a manufacturer-driven initiative aimed at providing a standardized, low-complexity digital interface for sensors and actuators. The technology was subsequently standardized as IEC 61131-9 under the designation Single-Drop Digital Communication Interface (SDCI) [4], [5]. IO-Link is defined as a point-to-point industrial communication protocol enabling bidirectional digital communication between sensors or actuators (referred to as *Devices*) and an IO-Link *Master*, while preserving the simplicity of traditional sensor wiring.

The IO-Link Master acts as the central communication controller within the system. It is responsible for initiating all communication cycles, managing device startup and operational mode transitions, providing power to the connected devices, and acting as a gateway between IO-Link Devices and higher-level automation systems such as PLCs or industrial networks. From the perspective of the Device, the Master defines the timing, structure, and direction of all data exchange.

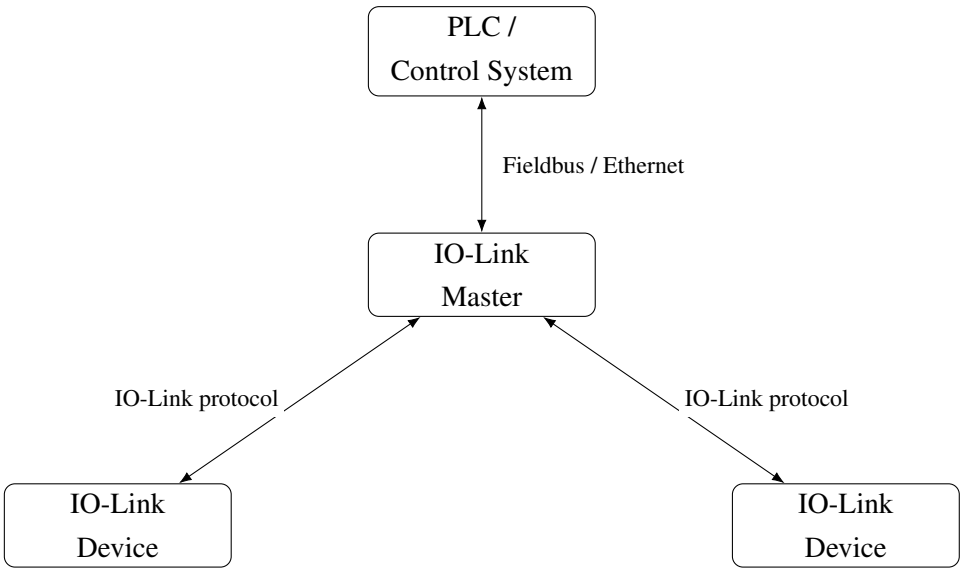


Figure 2.1: Typical IO-Link system configuration. The IO-Link Master manages point-to-point communication with multiple Devices and provides a gateway to higher-level control systems.

Unlike conventional fieldbus systems, IO-Link operates over a standard three-wire unshielded cable and follows a strict master–device communication model, in which each device is connected to the Master via a dedicated point-to-point link. This topology eliminates bus arbitration and enables deterministic communication behaviour, while significantly reducing cabling and integration complexity. This reduction in complexity is achieved not by minimizing the number of wires per device, but by allowing the reuse of standard, unshielded three-wire cables already prevalent in legacy analog and discrete sensor installations. This eliminates the need for expensive,

specialized shielded cabling and complex termination resistors typically required by multi-drop fieldbuses. These characteristics facilitate IO-Link Device implementations on low-cost microcontrollers while maintaining predictable timing properties and low system overhead, provided that the protocol timing and framing requirements are respected [5], [6].

The protocol supports both cyclic and acyclic data exchange. Cyclic communication is used for the continuous transmission of process data during normal operation, whereas acyclic communication enables access to parameters, identification data, diagnostics, and events. Through this combination of cyclic and acyclic services, IO-Link extends conventional sensors with standardized digital communication capabilities and facilitates their integration into higher-level automation and information systems [4], [6].

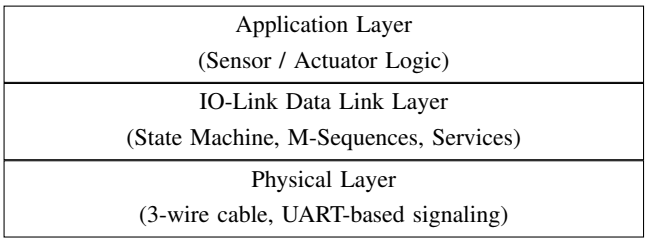


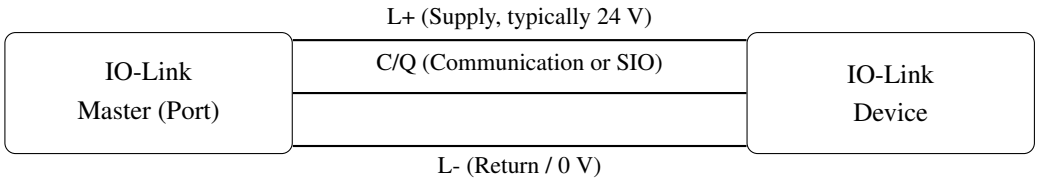
Figure 2.2: Simplified IO-Link communication stack illustrating the separation between application logic, protocol handling, and physical transmission.

2.3 Physical Layer and Signaling

IO-Link communication is implemented over a three-wire connection (L+, L-, and C/Q), where L+ and L- provide the device supply voltage and reference (typically 24 V DC supply and return), and the C/Q line is used as the bidirectional single-drop communication channel. The link is half-duplex: at any instant, only one side actively drives the C/Q line while the other side receives. The Master remains the communication controller and initiates all exchanges, thereby defining when the Device is allowed to respond [4], [5], [6].

In addition to carrying serial communication frames, the C/Q line may also operate in Standard Input/Output (SIO) mode. In SIO mode, the line is treated as a conventional binary switching signal rather than a digital communication channel [4], [5]. This mode provides crucial backward compatibility: an IO-Link device can function as a standard 24 V discrete sensor (e.g., PNP/NPN) when connected to a legacy digital input module that does not support IO-Link. The device remains in SIO mode until the IO-Link Master explicitly initiates digital communication via the wake-up procedure.

Communication model:
Master-driven link: the Master initiates each transaction and the Device responds.
Half-duplex on C/Q: only one side actively drives the line at any time.



Operating modes on C/Q:
COMM mode: asynchronous serial frames (UART-like framing).
SIO mode: conventional binary switching signal.

Figure 2.3: IO-Link three-wire single-drop connection. L+ and L- provide device power, while C/Q operates either as a communication line (COMM) or as a conventional switching signal (SIO). The Master controls the link and communication is half-duplex.

Figure 2.3 illustrates the three-wire wiring model and the dual role of the C/Q line in COMM and SIO modes.

2.3.1 Asynchronous Serial Transmission and Framing

At the signaling level, IO-Link uses asynchronous serial transmission with fixed framing and parity. This choice enables implementations on low-cost microcontrollers and allows the data link layer to be realized on top of conventional UART peripherals, provided that timing and framing constraints are respected [4], [5]. Communication speeds are grouped into COM1, COM2, and COM3, which correspond to distinct baud rates negotiated during startup based on Master and Device capabilities [4], [5].

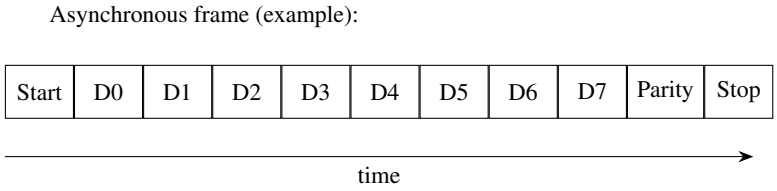


Figure 2.4: Conceptual asynchronous serial frame structure used by IO-Link signaling (start bit, data bits, parity, stop bit). Exact framing parameters depend on the negotiated COM mode [4], [5].

2.3.2 Wake-up Signaling

Before normal communication begins, the IO-Link Master may issue a wake-up signal on the C/Q line to transition a Device from an inactive state (or SIO mode) into communication-ready operation. At the physical layer, this wake-up signal is generated by forcing the C/Q line to the opposite of its current logic potential—either pulling a 24 V signal down to 0 V, or forcing a 0 V signal up to 24 V—for a strictly defined duration (typically around 80 µs). This electrical pulse precedes the subsequent startup sequence managed by the data link layer [4], [5], [6].

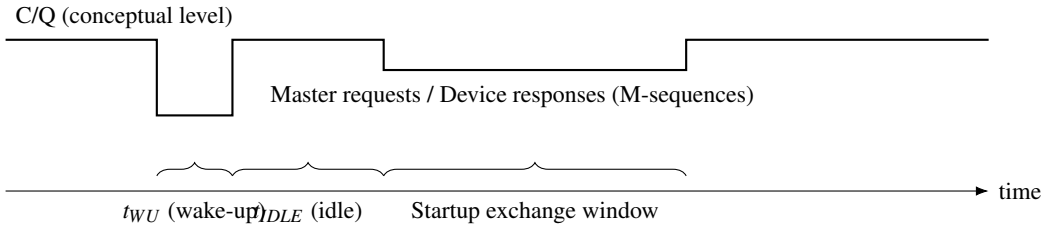


Figure 2.5: Wake-up and startup timing. Numerical constraints for wake-up duration, idle time, and response windows are defined by the IO-Link specification [4], [5].

2.4 Data Link Layer Fundamentals: M-sequences, Timing, and CK6

Above the physical layer, the IO-Link data link layer defines how information is exchanged in structured frames. Communication is organized into Master-initiated transactions where each Master message is followed by a corresponding Device response. These paired exchanges are referred to as M-sequences and form the basic unit of cyclic and acyclic communication services [4], [5].

2.4.1 M-sequences and Frame Structure

A M-sequence consists of a request sent by the Master and a response returned by the Device within a specified response window. Depending on the active operational state and the requested service, the payload transported in the request/response may represent cyclic process data, on-request data, diagnostic/event access, or ISDU-related content [4], [6].

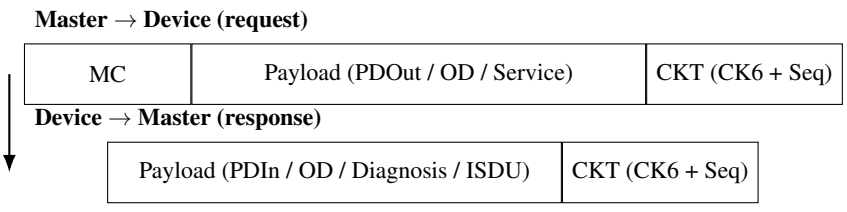


Figure 2.6: IO-Link M-sequence structure: a Master request followed by a Device response. Payload interpretation depends on operational mode and requested service; frames include protocol-level integrity information (CK6) and sequence control fields as defined by the specification [4], [5], [6].

2.4.2 Timing Constraints

IO-Link defines explicit timing relationships between Master requests and Device responses to ensure deterministic behavior. These include minimum idle times between frames, response windows, and cycle timing constraints for cyclic exchange. Parameters (e.g., t_{IDLE}, t_{RESP}, t_{CYCLE}), with numerical values and limits defined by the normative specification [4], [5].

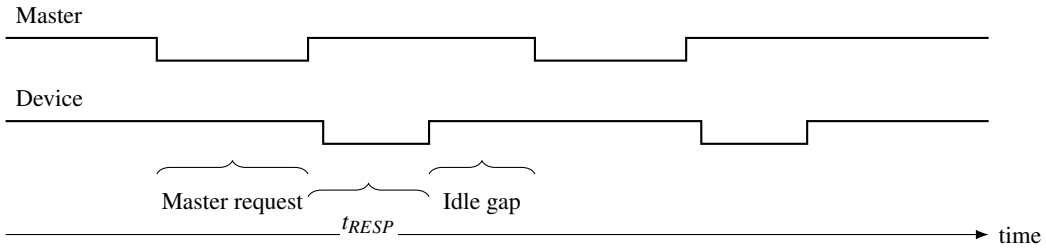


Figure 2.7: Conceptual Master-driven exchange: each request is followed by a Device response within a defined response window t_{RESP}. The specification defines the timing bounds required for deterministic operation [4], [5].

2.4.3 CK6 Checksum Validation

To protect communication integrity, IO-Link frames include a checksum mechanism (CK6) that enables detection of corrupted data at the protocol level. Checksum validation is performed at the receiver side and is used to accept or discard received frames. Robust implementations must treat checksum failures as invalid frames and avoid unintended state transitions triggered by corrupted traffic [4], [5].

2.5 Operational Model and Startup Procedure

Building on the framing and timing rules defined by the data link layer, IO-Link specifies an operational model that constrains which services may be accessed at each stage of the connection. This model is realized by a state machine comprising the *Startup, Preoperate,* and *Operate* states and is essential for predictable Master–Device interaction and interoperability in the IO-Link ecosystem as defined by the specification [4], [5].

2.5.1 Communication Modes: COM1, COM2, and COM3

IO-Link defines three standardized communication speeds, referred to as COM1, COM2, and COM3, corresponding to 4.8 kbaud, 38.4 kbaud, and 230.4 kbaud, respectively [4], [5]. Beyond throughput, these modes impose different timing constraints and implementation requirements, affecting achievable cycle time, interrupt latency tolerance, and buffer sizing.

COM1 represents the lowest transmission speed and provides the highest tolerance to timing uncertainty and signal degradation. It is suitable for simple devices and electrically demanding environments but results in longer communication cycles.

COM2 offers a balanced compromise between timing performance and implementation complexity. It enables faster cyclic exchange while maintaining manageable interrupt and buffer requirements for low-cost microcontrollers. For many embedded device implementations, including the prototype developed in this thesis, COM2 provides sufficient bandwidth for process data and bounded acyclic services without imposing excessive real-time constraints.

COM3 provides the highest throughput and shortest achievable cycle times. However, it requires stricter timing control, tighter interrupt latency bounds, and careful buffer management. Implementations targeting COM3 must ensure that the Device can respond within the specified response window while maintaining deterministic behavior under worst-case conditions.

The choice of COM mode therefore directly impacts buffer sizing, interrupt handling strategy, and overall system determinism. From a design perspective, communication speed is not merely a throughput parameter but a constraint that influences architectural decisions at the firmware level [4], [6].

2.5.2 Startup Sequence and COM Negotiation

IO-Link communication is established through a Master-driven startup procedure that transitions the Device from an initial inactive or unconfigured condition into an operational state. During startup, the Master determines the Device capabilities, selects a compatible communication mode (COM1/COM2/COM3), and configures the link for subsequent cyclic and acyclic services. This procedure is normative and must be handled deterministically by the Device to ensure interoperability with standard-compliant Masters [4], [5].

At a conceptual level, the startup process can be understood as a sequence of Master requests followed by Device responses (M-sequences), where specific Master Commands are used to query or configure communication parameters and to trigger transitions between operational states. The COM mode ultimately used for communication is selected based on the intersection of Master and Device capabilities and is applied consistently for subsequent exchanges [4], [5].

Figure 2.8 summarizes the main phases of a typical startup sequence. The exact command ordering and timing constraints are defined by the specification and depend on the Device capabilities and the operational context.

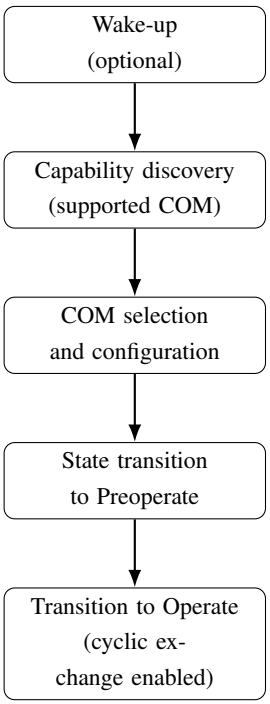


Figure 2.8: Overview of a Master-driven IO-Link startup sequence. Wake-up may precede capability discovery and COM selection; operational state transitions enable cyclic and acyclic services as specified by IO-Link [4], [5].

2.6 Data Types, Services, and State-Dependent Availability

IO-Link data exchange is structured into cyclic and acyclic services governed by the data link layer. The protocol defines the structure and timing of exchanged frames, as well as the set of services that are available depending on the operational mode (Startup, Preoperate, and Operate) [4], [5]. This section provides an overview of the core data types and services that an IO-Link Device is expected to support.

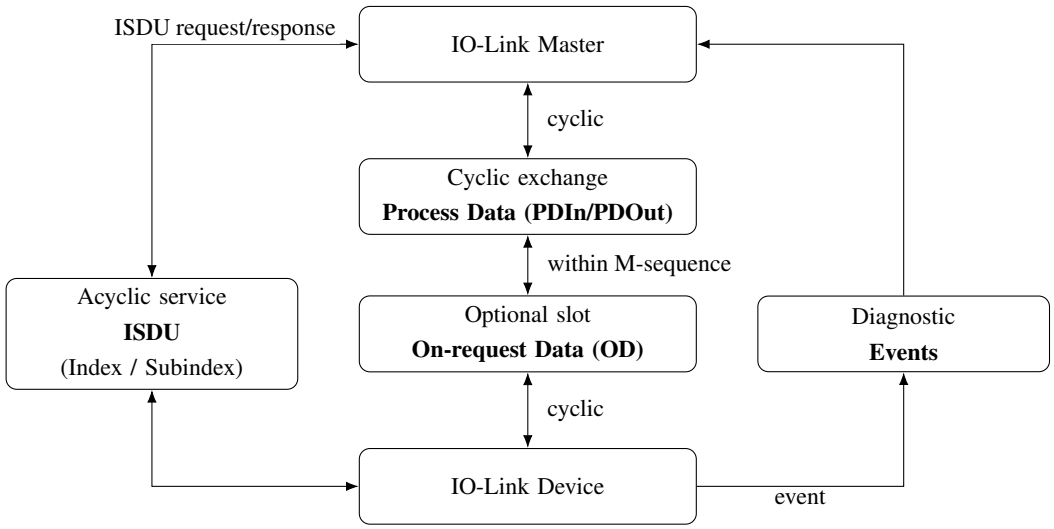


Figure 2.9: Overview of IO-Link data types and services: cyclic process data exchange, an optional on-request data (OD) slot within the M-sequence, acyclic ISDU access, and diagnostic event signaling between Master and Device.

2.6.1 Operational States and Service Availability

IO-Link defines an operational state model that constrains which services may be accessed and which data exchange patterns are valid at each phase of the connection. This state model is realized within the data link layer and is essential for predictable Master–Device interaction and interoperability [4], [5].

In *Startup*, the Master establishes communication parameters and triggers mode transitions. In *Preoperate*, the Device is communication-ready and supports acyclic services such as parameter access and diagnostics, while cyclic process data exchange is not yet enabled. In *Operate*, cyclic process data exchange is active, and acyclic services may still be accessed either through explicit acyclic exchanges or embedded mechanisms such as the on-request data field, depending on the service [4], [6].

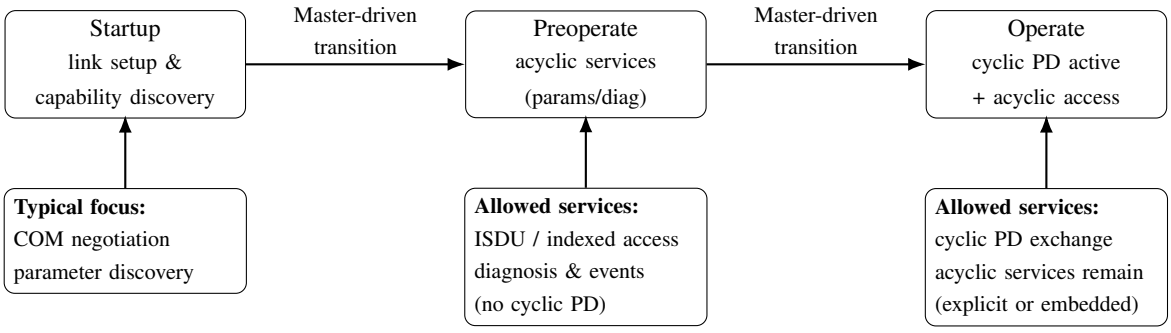


Figure 2.10: Simplified IO-Link operational state progression (Startup, Preoperate, Operate) and typical service availability per state. Exact transitions and command sequencing are defined by the specification.

Figure 2.11 summarizes the relationship between operational states and the main categories of services.

Service category	Startup	Preoperate	Operate
Startup control / negotiation	✓	✗	✗
Cyclic Process Data (PDIn/PDOut)	✗	✗	✓
On-request Data (OD)	✗	✓	✓
ISDU parameter access (acyclic)	✗	✓	✓
Diagnostics / Events	✗	✓	✓

Figure 2.11: Conceptual relationship between IO-Link operational states and service availability. ✓ indicates that a service category is available/allowed in the corresponding state, whereas ✗ indicates that it is not available/not allowed. Exact service constraints are defined by the specification [4], [5].

2.6.2 Cyclic data exchange: Process Data and On-request Data

During normal operation, the Master polls the Device in a deterministic cycle. The primary payload exchanged in this cycle is *Process Data* (PD), which conveys the sensor/actuator process value(s). Process Data is device-specific: its size and semantics are defined by the device design and described through the corresponding device description mechanisms (e.g., IODD in industrial contexts). In IO-Link terminology, Process Data is directional:

- **PDOut** (Master → Device): cyclic output data, typically used to command an actuator or to provide setpoints;
- **PDIn** (Device → Master): cyclic input data, typically the measured process value or device status information.

In addition to PD, cyclic frames may contain an *On-request Data* (OD) field. OD provides a mechanism to transport acyclic services alongside cyclic exchange, allowing the Master to request parameter or diagnostic information without leaving the cyclic communication context. In practice, OD acts as a bounded container whose interpretation depends on the requested service. In Operate mode, this enables a unified view of normal cyclic operation while still supporting configuration and diagnostic access [4].

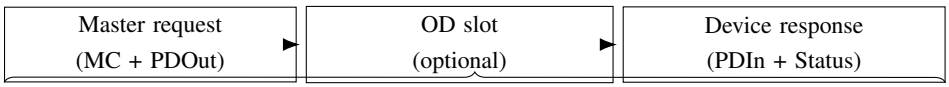


Figure 2.12: Conceptual cyclic exchange showing Process Data and the optional On-request Data (OD) slot within a single M-sequence. The OD field enables parameter or diagnostic access without leaving cyclic operation [4].

2.6.3 Representative frame structures and message examples

A frequent source of ambiguity when describing communication protocols is the absence of concrete message examples. For this reason, this section complements the conceptual description of IO-Link services with representative frame structures and selected real exchanges captured during validation using the simulated Master and the RP2040-based Device.

The logs were captured with the device operating in COM2 (38.4 kbaud) using UART framing 8E1. The examples below do not enumerate every possible message, but instead illustrate the general frame structure and a small set of cases that are particularly relevant for understanding the startup handshake, cyclic exchange, Type-0 ISDU transfers embedded in OD, and diagnostic/event retrieval.

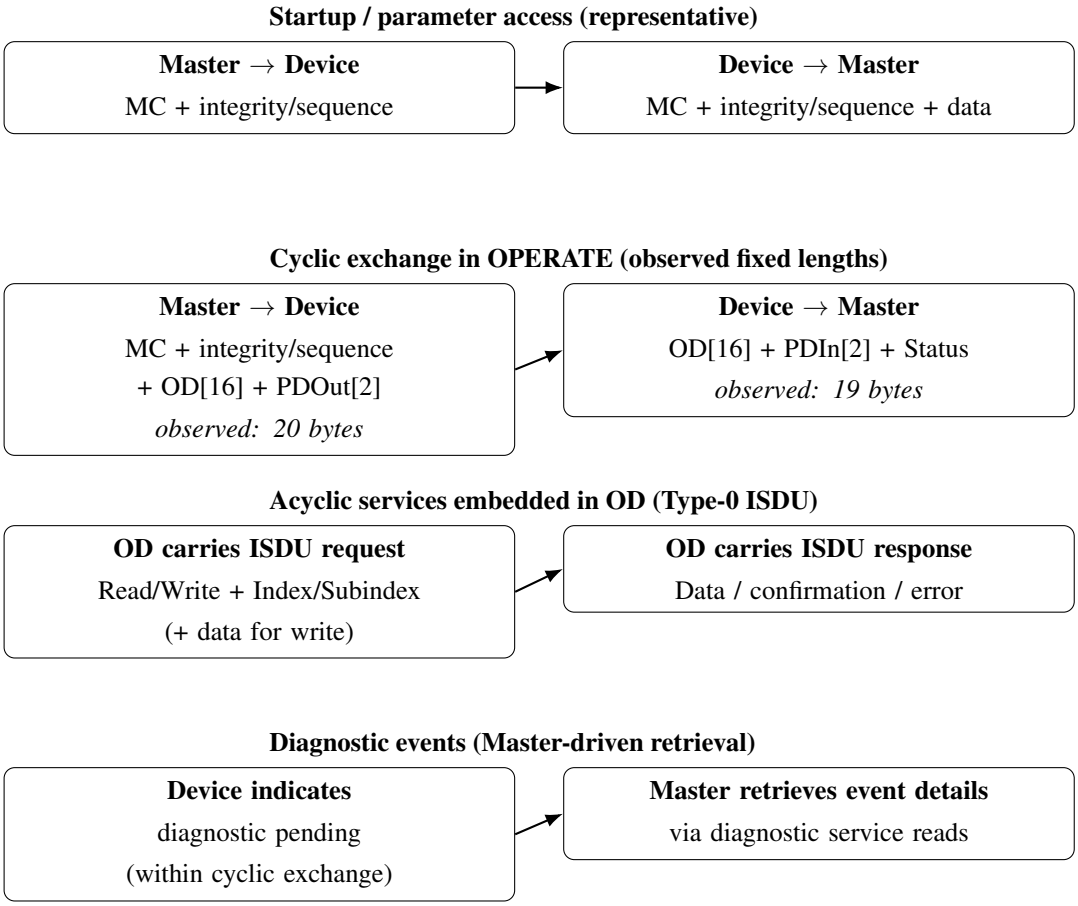


Figure 2.13: Observed high-level structure of representative Master→Device and Device→Master frames in the validation setup. The figure emphasizes the grouping of fields (MC, integrity/sequence, OD, PD, status) as used in the experimental logs. Exact normative definitions and constraints are given in [4], [5].

2.6.3.1 Startup parameter access (examples)

During STARTUP, the Master executes a deterministic parameter discovery sequence. Each parameter is retrieved using the same request/response structure:

- The Master sends an explicit **read request** identified by a Master Command (MC).
- The Device replies with the corresponding response frame, returning the requested data.

Example 1 — Read `MinCycleTime` (captured logs)

```
Master -> Device:  A2 01
Device -> Master:  A2 21 59
```

Interpretation.

- MC = 0xA2 identifies a STARTUP read of the `MinCycleTime` object.
- The returned data byte is 0x59.

Example 2 — Read M-sequence capability (captured logs)

```
Master -> Device:  A3 53
Device -> Master:  A3 4B 3F
```

Interpretation.

- MC = 0xA3 identifies a STARTUP read of the capability object.
- The returned data byte is 0x3F.

These examples highlight that STARTUP discovery is structurally uniform: the Master queries mandatory identification/capability objects and the Device returns the corresponding values in bounded responses.

2.6.4 Acyclic services: Parameter access (ISDU) and diagnostic events

While Process Data supports continuous cyclic exchange of primary process values, IO-Link additionally defines acyclic mechanisms for structured parameter access and diagnostic communication.

Indexed Service Data Unit (ISDU) The Indexed Service Data Unit (ISDU) provides indexed read and write access to device parameters using an **Index/Subindex addressing model**. This mechanism allows the Device to expose a structured parameter dictionary, enabling remote configuration, identification retrieval, and access to diagnostic objects independently of the cyclic process data format [4], [5].

An ISDU transaction follows a Master-driven request/response model:

- The Master issues a Read or Write service specifying an Index and optional Subindex.
- For Write services, a payload is included.
- The Device returns the requested data (Read), a confirmation (Write), or a standardized error response.

Type-0 (on-request) ISDU IO-Link defines multiple ISDU transport mechanisms. In minimal device implementations, **Type-0 (on-request) ISDU** transfers are typically sufficient. A Type-0 transaction consists of a single request and a single response embedded within the Master-driven communication cycle.

Because the payload is bounded and no segmentation or multi-frame reassembly is required, Type-0 ISDU is particularly suitable for embedded devices with fixed memory resources and deterministic timing constraints [4], [5].

More advanced implementations may support segmented ISDU transfers for larger parameter objects. In practice, implementations usually enforce explicit payload limits to ensure robustness against malformed or oversized requests.

Conceptual frame structure Conceptually, a Type-0 ISDU request (Master → Device) includes:

- a service identifier (Read or Write),
- Index and Subindex fields,
- an optional payload (Write),
- and integrity/sequence information defined at the data link layer.

The corresponding response (Device → Master) includes:

- a status field (OK or Error),
- returned data (Read) or confirmation (Write),
- and integrity/sequence information.

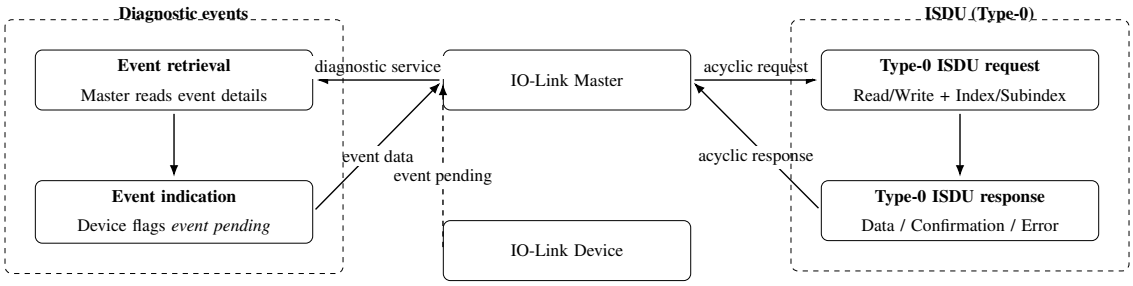


Figure 2.14: Conceptual workflow for Type-0 ISDU transactions and diagnostic event retrieval. ISDU requests (Index/Subindex) are explicitly issued by the Master and completed by a single response. Diagnostic events follow the same Master-driven paradigm: the Device indicates pending diagnostic information within cyclic exchange, and the Master retrieves event details using defined diagnostic services. Normative field encodings and timing constraints are defined in [4], [5].

Diagnostic events Diagnostic events follow the same strictly Master-driven model. Devices do not transmit unsolicited messages arbitrarily. Instead, diagnostic information is indicated within the cyclic exchange, and the Master retrieves event details through explicit diagnostic reads.

Events may represent internal faults, parameter changes, limit violations, or application-defined conditions and are classified according to severity levels defined by the specification. This approach preserves deterministic communication while enabling timely reporting of warnings and errors relevant to operation and maintenance [4], [5].

2.6.5 Diagnosis and event signaling

Beyond parameter access, IO-Link defines standardized diagnostic mechanisms that allow a Device to report abnormal conditions and relevant state changes to the Master. The protocol includes

an event signaling workflow in which events are generated by the Device and then reported to the Master using diagnostic services.

Events are classified by severity (e.g., notifications, warnings, and errors) and can represent conditions such as parameter changes, internal faults, or application-defined diagnostic states. Rather than being transmitted spontaneously at arbitrary times, event information is integrated into the Master-driven communication model: the Device signals that diagnostic information is pending, and the Master retrieves the corresponding event data through the defined diagnostic channel. This approach preserves deterministic communication while still enabling timely reporting of relevant conditions [4], [6].

Together, cyclic process data exchange, indexed parameter services (ISDU), and diagnostic event signaling constitute the core set of IO-Link device-facing services. Their correct availability and behavior are constrained by the operational mode and are fundamental to interoperability with standard-compliant IO-Link Masters.

2.7 IO Device Description (IODD)

In addition to protocol-level communication, IO-Link relies on a standardized device description mechanism known as the IO Device Description (IODD). The IODD is an XML-based file that formally describes the capabilities, parameters, process data structure, diagnostic information, and identification details of a specific IO-Link Device [4], [6].

2.7.1 Role of the IODD

The IODD serves as the semantic bridge between the Device and engineering or configuration tools. While the IO-Link protocol defines how data is transported, the IODD defines what that data represents. It specifies, among other elements:

- Device identification information (Vendor ID, Device ID);
- Supported communication capabilities;
- Process Data structure and bit-level meaning;
- Indexed parameters and their data types;
- Diagnostic events and their semantic interpretation.

When an IO-Link Master or engineering tool connects to a Device, the IODD enables the system to interpret raw protocol data in a structured and user-friendly manner. Without an IODD, a Device may still communicate at the protocol level, but higher-level tools cannot automatically interpret parameter meaning, value ranges, or diagnostic codes.

2.7.2 IODD Generation and Practical Considerations

Generating a valid IODD file for a new IO-Link Device is a non-trivial process. The file must conform strictly to the schema defined by the IO-Link Consortium and pass validation checks to ensure interoperability with compliant engineering tools. In industrial practice, IODD generation is often performed using specialized authoring tools, some of which are commercial and require licensed access.

For newly developed devices, professional IODD generation and certification workflows may involve paid services or dedicated software environments provided by authorized vendors. These tools assist in ensuring schema compliance, correct data typing, multilingual support, and alignment with device identification requirements defined by the IO-Link ecosystem.

Within the scope of this thesis, the focus is placed on protocol-level implementation and validation rather than on full industrial deployment or certification. Consequently, while the role of the IODD is acknowledged as essential for industrial integration, formal IODD generation and certification were considered beyond the validated operational scope of the prototype. The developed stack, however, is architecturally compatible with the future generation of a compliant IODD should industrial deployment be pursued.

The separation between protocol transport (defined by the IO-Link data link layer) and semantic device description (defined by the IODD) constitutes a fundamental architectural principle of IO-Link. This separation allows protocol implementations to remain lightweight at the device side while delegating semantic interpretation and visualization to higher-level tools.

Chapter 3

State of the Art

3.1 Introduction

Building upon the technical foundations introduced in Chapter 2, this chapter shifts from conceptual protocol description to implementation-facing realities. Rather than reintroducing IO-Link fundamentals, the focus is on how sensor-level digital communication and retrofit principles materialize in (i) commercial retrofit products, (ii) academic frameworks, and (iii) publicly available device-side stacks.

The objective is twofold:

- Position IO-Link within practical retrofit strategies for brownfield environments;
- Identify gaps in openly documented, implementation-focused IO-Link *device-side* stacks that motivate the work presented in this thesis.

Particular emphasis is placed on transparency, specification traceability, and experimental validation capability, as these dimensions determine whether a stack can be inspected, stress-tested, and argued academically.

3.2 Sensor Retrofit in Industrial Automation

Building upon the background introduced in Chapter 2, this section summarizes the IO-Link characteristics most relevant to retrofit positioning. The retrofitting of existing industrial equipment has become increasingly relevant in both academic research and industrial practice, particularly in the context of Industry 4.0. Multiple studies report that a significant share of industrial assets in operation predates widespread digital connectivity, yet remains mechanically and functionally adequate, making full replacement economically unattractive [8], [9].

Brownfield environments are commonly characterized by long equipment lifecycles, high downtime costs, certification constraints, and limited documentation of legacy systems [8]. Within this context, digital retrofitting aims to extend digital visibility and connectivity while minimizing operational disruption and capital expenditure [9].

Common retrofit approaches include [8]:

- **Signal conversion modules**, which digitize analog outputs (e.g., 0 V to 10 V, 4 mA to 20 mA) and expose them through digital interfaces;
- **Gateway-based solutions**, aggregating multiple sensors into a centralized node communicating with higher-level systems;
- **Embedded retrofit adapters**, attached directly to individual sensors and providing standardized digital communication at the device level.

Embedded retrofit adapters preserve decentralization and enable per-device identification, diagnostics, and parameterization, which are aligned with Industry 4.0 design goals [8]. However, they require a sensor-level communication technology that balances simplicity, determinism, bounded resource usage, and interoperability.

3.3 Digital Communication at the Sensor Level

Traditional industrial fieldbus technologies such as PROFIBUS and CANopen support multi-device communication and rich data models [3]. In practice, adopting such technologies at the sensor level can imply additional hardware requirements (e.g., dedicated transceivers/controllers) and increased integration effort. Moreover, depending on the system scale and operational constraints, commissioning can involve non-trivial configuration and addressing tasks, particularly when retrofitting legacy installations.

For simple sensors and cost-sensitive retrofit scenarios, this effort may be disproportionate. Implementing a full fieldbus stack on a resource-constrained microcontroller increases development and verification burden and may exceed the memory and timing budget of low-cost platforms.

At the sensor level, communication technologies must reconcile electrical simplicity with increasing demands for structured data exchange and device-level semantics. For sensor-level retrofit, three properties are particularly relevant:

- **Deterministic master-driven communication**, to simplify device behavior and testing;
- **Bounded resource requirements**, to ensure implementability under tight memory constraints;
- **Feasibility on low-cost microcontrollers**, enabling scalable retrofit adapters.

IO-Link was designed for point-to-point, master-driven sensor/actuator connectivity and is therefore frequently positioned between purely analog signaling and full-scale multi-drop fieldbuses. From a research and validation perspective, this positioning is relevant because it enables protocol experimentation with comparatively limited implementation complexity, while still providing structured cyclic and acyclic services.

Commercial gateways and signal conditioning modules provide an alternative integration path. However, in many commercial products the detailed protocol logic (e.g., internal state transitions, error handling, and edge-case behavior) is not externally observable beyond vendor-exposed diagnostics and configuration interfaces. This limits independent traceability and systematic protocol-level fault injection, which are relevant requirements for the experimental methodology adopted in this thesis.

3.4 Commercial IO-Link Retrofit Solutions

IO-Link has achieved significant industrial adoption, and multiple vendors provide signal conditioners/converters that interface legacy analog signals to IO-Link process data. Representative examples include vendor families of analog-to-IO-Link converters and signal conditioners [10], [11], [12], [13], [14], [15].

Such products typically offer parameterizable analog I/O channels, configurable scaling and resolution, and standardized IO-Link communication behaviour as part of a complete industrial device. Their availability supports the industrial relevance of IO-Link-based retrofit architectures.

From the standpoint of protocol engineering research, these solutions present practical limitations. Firmware is proprietary, and the device-side protocol implementation is not designed for external instrumentation. Consequently, detailed behaviour—such as precise state machine transitions, checksum handling under malformed traffic, or response generation under boundary conditions— cannot be independently inspected or modified using standard tooling. For these reasons, commercial converters are valuable as reference products for deployment, but are not an appropriate substrate for transparent protocol-level validation.

3.5 Academic and Open Research on IO-Link

Academic work referencing IO-Link frequently targets high-level system integration, architecture comparisons, and broader industrial communication analyses [3], [8], [9]. In contrast, detailed and openly documented *device-side* protocol implementations are less commonly provided as reusable artefacts. To understand this disparity, it is necessary to examine the current focus of IO-Link research, which can be broadly categorized into master implementations, macro-system integration, and wireless extensions.

3.5.1 IO-Link Master Implementations and Bridging

A significant portion of open research focuses on the development and evaluation of IO-Link Master stacks rather than the edge devices. A representative contribution on tooling and reproducibility is the rapid prototyping framework presented by Chavez *et al.* [16], which structures IO-Link development around modular layers (hardware abstraction, protocol stack logic, and application/API). Although focused on master-side prototyping, the work highlights implementation

concerns that also apply to device-side stacks, including deterministic startup handling, traceable state transitions, and systematic validation.

Further research has expanded on master-side capabilities. Recent proof-of-concept implementations have explored the execution of cyclic Process Data and acyclic On-request Data (ISDU) within custom C++ master stacks [17]. Other efforts have demonstrated practical implementations of microcontroller-based IO-Link master bridges utilizing standard UART peripherals for simple data acquisition [18]. From an industrial reliability standpoint, studies have evaluated the application of IEC 61131-9 transmission rates to actively reduce machinery downtime [19], while network-level research has explored integrating IO-Link Masters with Ethernet and SNMP protocols to enable predictive maintenance via web servers [20].

3.5.2 System Integration and Industry 4.0

Beyond protocol handling, academia heavily investigates how IO-Link data is propagated to cloud and edge systems. The coexistence and standardization of Operation Technology (OT) and Information Technology (IT) heavily relies on IO-Link as the foundational standard for actuator and sensor data acquisition [21].

A prominent trend in this domain is the integration of IO-Link with OPC UA (Open Platform Communications Unified Architecture). Industrial feedback from semiconductor manufacturing has highlighted the adoption of standardized OPC UA companion specifications for IO-Link to overcome vertical integration bottlenecks [22]. At the edge level, researchers have developed custom C-based driver frameworks to establish logical connections between Edge computing platforms and IO-Link masters over serial buses, specifically targeting advanced condition monitoring applications [23].

3.5.3 IO-Link Wireless and Sensor Extensions

Recently, the literature has shifted toward extending the physical medium of the protocol. IO-Link Wireless has been extensively analyzed for its potential to enhance factory automation in environments where physical cabling is restrictive [24]. This has led to novel testing methodologies, including the evaluation of over-the-air (OTA) sensitivity testing in reverberation chambers for wireless IO-Link equipment [25]. Furthermore, cutting-edge deployments are currently exploring the fusion of OPC UA with IO-Link Wireless to establish cyber-physical finite element sensor networks [26].

3.5.4 Summary of Identified Gaps

Overall, the literature strongly supports the relevance of IO-Link for retrofit, integration, and Industry 4.0 modernization. However, it also reveals a structural gap: while master bridges, cloud integrations (OPC UA), and wireless extensions are heavily researched, the underlying *device-side* stack is frequently treated as an opaque, commercial "black box". This leaves room for

openly available device-side stacks whose protocol behavior is not only inspectable but explicitly validated under controlled fault injection—which forms the core motivation for this dissertation.

3.6 Open IO-Link Device Stack Implementations

Publicly available open-source IO-Link *device* stacks remain limited in scope and documentation depth. In this context, "device-side stack" refers specifically to the firmware component executing on the IO-Link Device, excluding master implementations and higher-level engineering tools. A representative example used as the starting point for this thesis is the lightweight `lwIOLink` library [27], publicly available as an open repository (GitHub) and targeting microcontroller-based experimentation.

In its baseline form, such lightweight stacks typically prioritize initial bring-up and basic cyclic exchange. As a result, features relevant for specification-oriented validation are frequently absent or only partially structured, including:

- explicit modelling of operational states (Startup, Preoperate, Operate);
- structured ISDU handling with clearly delimited scope;
- diagnostic/event handling aligned with normative semantics;
- non-volatile parameter persistence aligned with Data Storage workflows;
- documented traceability to IEC 61131-9 / IO-Link specification requirements.

From an implementation engineering perspective, the absence of explicit state modelling, bounded buffer management, and documented error semantics limits the ability to assess robustness under malformed traffic, communication disturbances, and edge-case scenarios.

3.7 Summary of Identified Gaps and Positioning

The reviewed landscape reveals a practical gap. Commercial retrofit converters provide industrial maturity but are closed and not designed for transparent protocol inspection. Academic work motivates IO-Link for retrofit and proposes modular development principles, yet often does not provide reusable, device-side protocol artefacts suitable for low-level validation. Open-source stacks improve accessibility, but frequently lack the combination of (i) explicit and verifiable operational state modelling, (ii) bounded-memory guarantees suitable for rigorous embedded reasoning, (iii) explicit alignment to normative requirements within a declared scope, and (iv) controlled and reproducible validation workflows.

Table 3.1 summarizes this comparison and motivates why this thesis focuses on a specification-oriented *subset* with high observability and controlled validation, rather than claiming universal or certified compliance.The objective is therefore not to compete with certified industrial stacks, but

to provide an inspectable and experimentally validated reference implementation within a clearly bounded and explicitly documented scope.

Table 3.1: Structured comparison between commercial IO-Link retrofit converters, open-source device stacks, and the specification-oriented approach adopted in this thesis.

Dimension	**Commercial converters**	**Open-source device stacks**	**This thesis**
Primary objective	Industrial deployment and certified product lifecycle.	Demonstration, prototyping, education.	Specification-oriented subset with explicit validation scope.
Firmware openness	Proprietary; internal implementation not accessible.	Source code available; documentation depth varies.	Open baseline extended; scope and limitations explicitly documented.
Protocol observability	Limited to exposed parameters and vendor tooling.	Logging/trace support varies across projects.	High observability via controllable master and captured protocol traces.
Specification traceability	Compliance evidence may exist, but internal mapping is not public.	Often partial or implicit alignment to specification.	Explicit alignment with IO-Link V1.1.4 within declared implementation scope.
Operational state modelling	Implemented but not externally inspectable.	Frequently simplified or partially implemented.	Explicit modelling of Startup, Preoperate, and Operate states.
Acyclic services (ISDU)	Commonly supported, often including segmentation.	Often limited to simplified Type-0 exchanges.	Type-0 ISDU implemented with bounded payload; segmentation out of scope.
Diagnostics and events	Integrated into vendor ecosystem; internal behaviour opaque.	Frequently simplified or partially structured.	Specification-oriented diagnostic workflow implemented and experimentally validated.
Persistence	Supported; implementation details proprietary.	Often absent or lacks integrity verification.	Selected parameters persisted with CRC-based integrity verification.
Fault injection capability	Not exposed to end users.	Possible in principle but not always systematically structured.	Controlled validation environment enabling malformed and oversized request injection.
Platform flexibility	Vendor-specific hardware platforms.	Often tied to specific MCU/board.	Architecturally reusable; validated on RP2040 with transport abstraction.
Research suitability	Limited (black-box behaviour).	Suitable for experimentation but often lacks systematic validation.	Transparent implementation with controlled and reproducible validation methodology.

In response to this gap, the subsequent chapters present the extension and restructuring of a lightweight open device stack into a specification-oriented implementation on RP2040, together with a validation methodology based on a fully controllable master. The contribution is therefore framed as an *inspectable and experimentally validated subset* aligned with IO-Link V1.1.4, rather than as a claim of certified or universal compliance.

Chapter 4

Architecture of the SpecIOLink-Device Stack

4.1 Introduction

This chapter presents the software architecture of the **SpecIOLink-Device** stack developed in this thesis. The objective of this chapter is strictly architectural: to define the structural organization of the codebase, the responsibility boundaries between modules, the state modelling strategy, and the memory discipline adopted—specifically, the exclusive use of static memory allocation and fixed-capacity buffers—to ensure deterministic embedded execution.

No platform-specific validation details (RP2040 configuration, UART electrical setup, master implementation, or experimental results) are discussed here. Those aspects are addressed exclusively in Chapter 5. This separation ensures that architectural reasoning is not conflated with platform-dependent implementation choices.

4.1.1 Normative Context

IO-Link is a master-driven, point-to-point communication protocol designed for sensor and actuator connectivity over a UART-based physical interface. In its normative definition, IO-Link specifies three standardized communication speeds (COM1, COM2 and COM3), UART framing with 8 data bits and even parity (8E1), structured master-controlled message sequences (M-sequences), distinct operational modes (Startup, Preoperate and Operate), cyclic process data exchange, acyclic indexed parameter access (ISDU), and diagnostic event signalling.

SpecIOLink-Device implements a clearly declared subset of these behaviours. The implementation is specification-oriented, meaning that implemented features follow normative constraints where applicable, but the stack does not claim full or certified compliance with IO-Link V1.1.4. All implemented and excluded features are explicitly documented.

4.1.2 Baseline and Architectural Contribution

As discussed in Chapter 3, the open-source `lwIOLink` library served as an initial experimental baseline. In its baseline form, however, it does not provide a sufficiently explicit architectural separation between (i) runtime control flow and (ii) IO-Link operational semantics. It also lacked a clearly bounded and documented treatment of ISDU handling, deterministic diagnostic event management, and an integrity-protected persistence strategy. Finally, the baseline did not offer the level of architectural traceability required to argue correctness and limitations in an academic setting.

The contribution of this work is the restructuring and extension of that baseline into **SpecIOLink-Device**. The resulting stack is specification-oriented within a declared subset and is architected around explicit state modelling, bounded memory guarantees, deterministic execution, and strict separation of concerns.

In this context, strict separation of concerns denotes the explicit architectural decoupling between three primary domains: (i) hardware-dependent transport primitives (e.g., UART framing and half-duplex direction control), (ii) the protocol core responsible for data link layer semantics, frame validation, and operational state transitions, and (iii) the application layer where sensor-specific process data mapping and ISDU callbacks reside.

By rigidly enforcing these boundaries—specifically ensuring that application-level code cannot directly access protocol buffers or inadvertently trigger operational mode shifts—the architecture safeguards internal state consistency. This structured isolation enables transparent mapping between normative requirements and the concrete code structure without claiming certified or universal compliance.

4.2 Architectural Scope and Boundaries

SpecIOLink-Device is strictly a **device-side** protocol stack. Its architectural scope encompasses Data Link layer behaviour within a declared subset of IO-Link V1.1.4, including cyclic process data exchange (PDIn/PDOut), Type-0 ISDU parameter access, diagnostic event storage and prioritization, and CRC-protected parameter persistence.

The following elements are explicitly outside the architectural scope:

- Master-side protocol implementation;
- Segmented ISDU transfers (Type-1/Type-2);
- Industrial 24 V physical layer integration within the core;
- Certified compliance mechanisms;
- Experimentally validated full COMLOST recovery.

The stack assumes the presence of a byte-oriented half-duplex transport interface. Normative UART framing (8E1) and the selected communication speed (COM1/2/3) are configured at

initialization time via the transport binding and are not dynamically negotiated by the protocol core.

4.3 Layered Architecture

To ensure clarity, maintainability, and controlled reasoning about correctness, SpecIOLink-Device follows a layered architectural model. Each layer has a strictly delimited responsibility and does not directly manipulate the internal state of other layers.

Persistent parameters are stored in non-volatile memory using a CRC32-based integrity verification mechanism. The checksum is computed upon every parameter update and validated during device initialization, ensuring deterministic recovery and protection against corrupted memory states. The checksum is computed upon every parameter update and validated during device initialization, ensuring deterministic recovery and protection against corrupted memory states.

Figure 4.1 will summarize the four conceptual layers: (i) a *transport interface* responsible for byte-oriented half-duplex communication, (ii) a *protocol core* implementing IO-Link Data Link semantics within the declared subset, (iii) an *application interface* exposing controlled integration hooks, and (iv) a *persistence/diagnostic subsystem* handling storage and events.

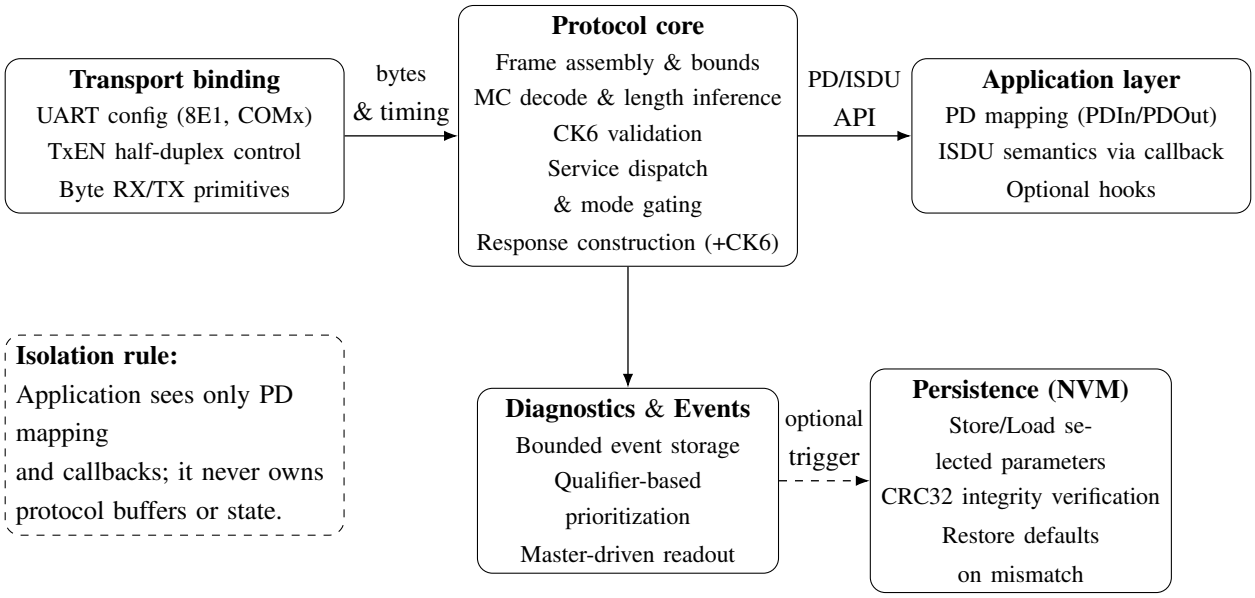


Figure 4.1: Layered architecture of SpecIOLink-Device (conceptual view). The stack is partitioned into a transport interface (byte-level half-duplex communication), a protocol core implementing IO-Link Data Link semantics within the declared subset, an application interface exposing controlled integration hooks, and a persistence/diagnostic subsystem. This separation supports traceability and bounded embedded execution while keeping hardware binding concerns outside the protocol semantics [4], [5].

4.3.1 Transport Interface Layer

This layer abstracts the underlying hardware communication and exposes a deterministic, byte-oriented interface to the protocol core. In SpecIOLink-Device, the transport *binding* configures the UART framing mandated by IO-Link (8E1) and selects the communication speed (COM1/ COM2/ COM3) at initialization time. It also implements half-duplex direction control through transmission-enable signalling (TxEN) and provides basic receive and transmit primitives.

Crucially, this layer contains no IO-Link protocol semantics: it does not interpret frames, enforce operational modes, or dispatch services. Its purpose is to isolate hardware-dependent concerns while providing timing-robust and reproducible communication behaviour to the protocol core.

4.3.2 Protocol Core Layer

The Protocol Core implements IO-Link Data Link behaviour within the declared subset. Architecturally, it is responsible for assembling incoming frames into bounded buffers, validating message structure and checksum (CK6), decoding master commands, and enforcing operational mode constraints. After validation, it dispatches the requested service (process data exchange, ISDU access, or diagnostics) and constructs the corresponding response frame in a deterministic manner.

This layer contains the primary state machines and represents the semantic center of the stack. All protocol-level constraints are enforced here, while transport-specific configuration and application-specific parameter semantics remain external.

4.3.3 Application Interface Layer

The application interface provides controlled integration hooks without exposing internal protocol state. It allows the application to map cyclic process data (PDIn/PDOut) and to implement indexed parameter semantics through the ISDU callback interface. Optional runtime hooks may be provided for application-specific behaviour, but they do not grant direct access to protocol buffers or internal state variables.

This separation ensures that application code cannot unintentionally compromise protocol correctness or state consistency.

4.3.4 Diagnostic Event Subsystem

The diagnostic subsystem manages the generation, prioritization, and queuing of diagnostic events, strictly adhering to the same bounded-memory discipline as the protocol core. Events are stored in statically allocated, fixed-capacity structures governed by a severity-based prioritization strategy. Architecturally, the event management logic is fully isolated from the low-level message parsing pipeline. This separation prevents unintended coupling between diagnostic signaling and the device's runtime control flow, ensuring that a flood of diagnostic events cannot exhaust memory or disrupt the rapid execution path of the communication cycle.

4.3.5 Persistence Subsystem

The persistence subsystem is responsible for the reliable storage and retrieval of critical configuration parameters across power cycles. To ensure operational safety and state consistency, persistent parameters are written directly to Non-Volatile Memory (NVM) and protected by a CRC32-based integrity checksum. During device initialization, this checksum is validated to prevent the loading of corrupted or incomplete memory states. Similar to the diagnostic mechanisms, persistence operations are architecturally decoupled from the message processing pipeline, ensuring that NVM write latencies do not interfere with strict protocol timing or operational mode transitions.

4.4 Device Core Structural Model

The architectural nucleus of SpecIOLink-Device is the `Device` class. This class centralizes ownership of protocol-relevant buffers and coordinates the interaction between the transport interface, the protocol state machines, the ISDU callback interface, the event subsystem, and persistence mechanisms.

Centralizing protocol control within a single core object simplifies reasoning about state transitions, buffer ownership, and deterministic execution flow. Figure 4.2 provides a structural overview of the core classes and interfaces.

4.4.1 Core Architectural Components

The SpecIOLink-Device core is centered around the `Device` class, which coordinates transport interaction, protocol state management, service dispatch, and response construction. The `Device` instance owns all protocol-relevant buffers and internal state machines, ensuring that buffer ownership and state mutation remain centralized and auditable.

Hardware binding parameters are encapsulated in `HWConfig`. This structure defines the communication speed selection (COM1/2/3) and the necessary pins for UART and half-duplex direction control. Protocol semantics are intentionally excluded from `HWConfig` to preserve portability and separation of concerns.

Persistent configuration is stored in `DeviceSettings`, which defines the fields subject to non-volatile storage and includes an integrity checksum. This structure is treated as an atomic persistent unit with fixed layout, enabling deterministic storage size and predictable recovery behaviour.

Acyclic parameter access is decoupled from protocol logic through the `ISDUHandler` callback interface. The protocol core validates ISDU requests, extracts index/subindex fields, enforces payload bounds, and invokes the application callback to resolve parameter semantics, without exposing internal buffers to application code.

Diagnostic events are stored in a fixed-capacity queue owned by the `Device` class. The queue enforces bounded memory usage, applies severity-based prioritization, and rejects insertions once full, thereby preserving deterministic execution.

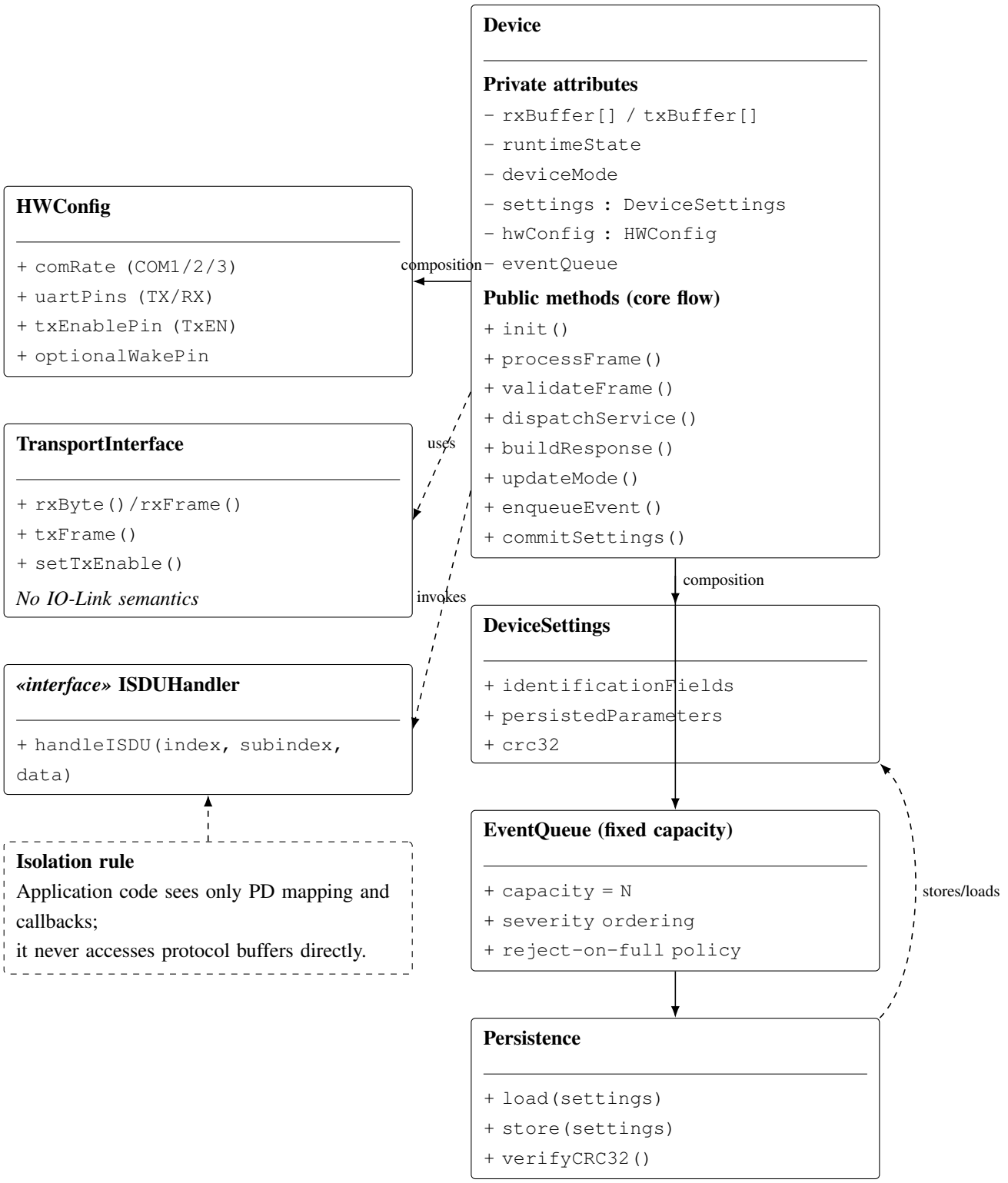


Figure 4.2: UML structural class diagram of the SpecIOLink-Device core. The `Device` class centralizes protocol buffers and state machines and coordinates transport interaction, ISDU handling, diagnostic events, and persistence. Associations indicate ownership (composition) and usage relationships. Visibility markers follow UML conventions (+ public, private).

Configuration and Interface Structures

Hardware binding parameters are encapsulated in `HWConfig`, which defines the selected communication speed (COM1/COM2/COM3) and the pin configuration required for UART and half-duplex direction control. Protocol semantics are intentionally excluded from `HWConfig` to preserve portability and separation of concerns.

Persistent configuration is stored in `DeviceSettings`. This structure defines the fields subject to non-volatile storage and includes an integrity checksum. It is treated as an atomic persistent unit with fixed layout, enabling deterministic storage size and predictable recovery behaviour.

Acyclic parameter access is decoupled from protocol logic through the `ISDUHandler` callback interface. The protocol core validates requests, extracts index/subindex fields, enforces payload bounds, and invokes the application callback to resolve parameter semantics, without exposing internal buffers to application code.

Diagnostic events are stored in a fixed-capacity queue owned by the `Device` class. The queue enforces bounded memory usage, applies severity-based prioritization, and rejects insertions once full, thereby preserving deterministic execution.

4.4.2 Static Allocation and Determinism

A fundamental architectural constraint is the complete avoidance of dynamic memory allocation during runtime. All protocol buffers, event storage structures, and persistent configuration containers are statically allocated with fixed bounds determined at build time. This design eliminates heap fragmentation and prevents runtime memory growth, ensuring predictable memory footprint and more stable worst-case timing behaviour.

Because buffer sizes and ownership are explicit, the architecture also simplifies reasoning about safety properties such as overflow protection, bounded execution paths, and state consistency across malformed or truncated communication.

4.5 State Modelling Architecture

A central architectural decision in SpecIOLink-Device is the explicit separation between two independent state abstractions: one governing internal execution flow (runtime control state) and another representing IO-Link operational semantics (operational mode state).

As illustrated in Figure 4.3, this separation prevents ambiguity between how the firmware executes internally and which IO-Link mode the device is currently in. Specifically, the internal *runtime control state* manages execution phases such as reception, validation, dispatch, and TxEN-controlled transmission. In contrast, the *operational mode state* strictly governs the IO-Link protocol modes (Startup, Preoperate, Operate).

By decoupling these abstractions, mode gating is safely enforced at dispatch, and operational mode transitions are applied only after full frame validation (including length and CK6 checksum

verification). This design avoids unintended semantic changes or implicit state transitions caused by malformed or truncated traffic.

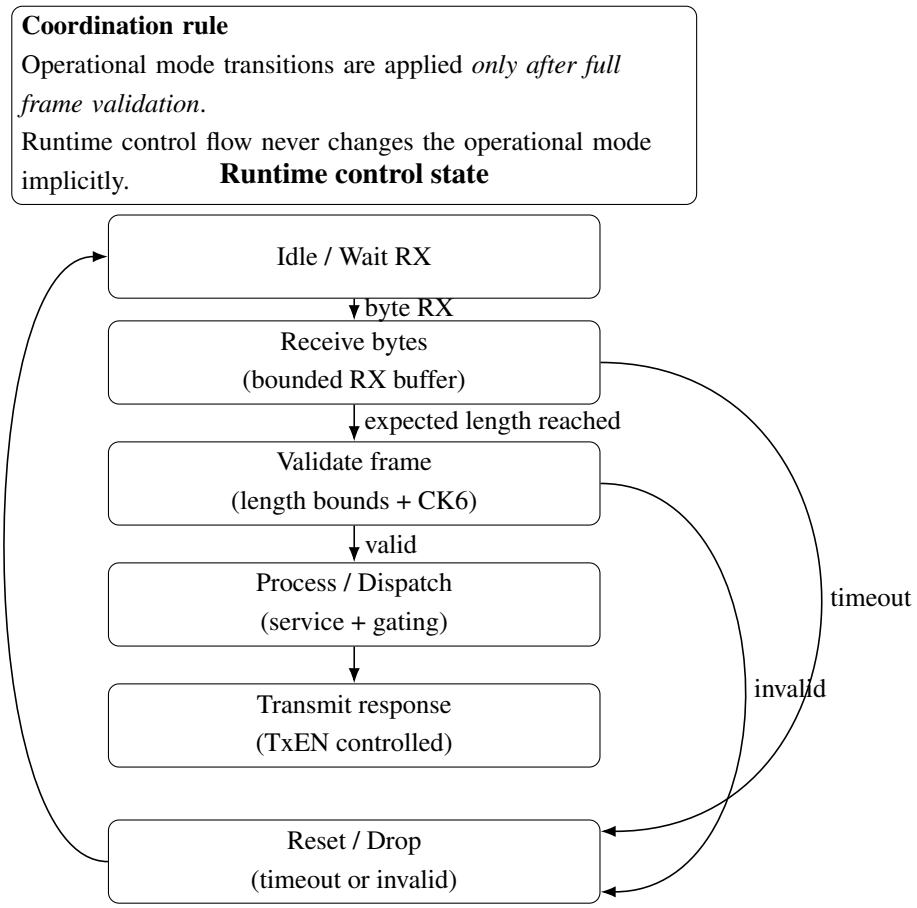


Figure 4.3: Dual-state modelling (runtime control view). Execution phases are clearly separated with bounded buffering and validation prior to semantic dispatch. Invalid or incomplete frames are dropped without semantic side-effects.

4.5.1 Runtime Control State

As depicted in Figure **??**, the runtime control state models the internal execution phases of the firmware and is not an IO-Link operational concept. It structures the device loop into clear stages such as wake-up handling, frame reception, message processing, and response transmission. By enforcing a staged progression, the runtime control state prevents incomplete frames from reaching semantic processing and ensures that transmission occurs only after validation.

In practice, this state machine functions as a structural safeguard around the protocol core, preserving deterministic behaviour even under truncated or malformed traffic.

4.5.2 Operational Mode State

The operational mode state represents the IO-Link device modes defined by the specification (Startup, Preoperate, and Operate). These modes constrain which services are permitted at a given time. For instance, identification and initialization behaviour occurs in Startup, acyclic parameter access is governed in Preoperate, and cyclic process data exchange is performed in Operate.

This state therefore governs what the device is allowed to do at protocol level, independently of how the firmware progresses through runtime execution phases.

4.5.3 Architectural Separation Rationale

Combining runtime control and operational semantics into a single state machine would introduce ambiguity and increase the risk of unintended transitions. By separating the two abstractions, runtime faults (e.g., malformed frames) do not implicitly alter the operational mode, and operational mode transitions can be strictly gated by validated protocol events. This also improves traceability: protocol correctness can be reasoned about independently of execution flow, and state invariants become easier to argue and verify.

4.5.4 Deferred Mode Transition Mechanism

Operational mode transitions are not applied immediately upon command detection. Instead, the master command is decoded and only after full frame validation (including checksum verification) is the transition applied at a controlled stage of the runtime loop.

This deferred-application strategy prevents partial state transitions in the presence of corrupted or incomplete frames and ensures that operational state changes are atomic with respect to validated protocol events.

4.6 Message Processing Architecture

IO-Link communication is strictly master-driven. At the device side, this means that all protocol activity is exclusively triggered by incoming master frames.

As illustrated in Figure 4.4, the SpecIOLink-Device architecture organizes message handling as a structured processing pipeline. Incoming bytes are first fully received and accumulated into a bounded buffer, where the expected length is inferred from the Master Command field. The frame is then structurally validated (including length and CK6 checksum verification), subsequently interpreted at the semantic level, and finally translated into a deterministic response.

In this architecture, a deterministic response signifies that the reply is generated within a strictly bounded time window (adhering to IO-Link timing constraints), that each master command has only one predictable response, the responses are constructed using statically allocated buffers with bounded payloads, and executed via fixed paths that eliminate dynamic memory allocation and unpredictable processing delays.

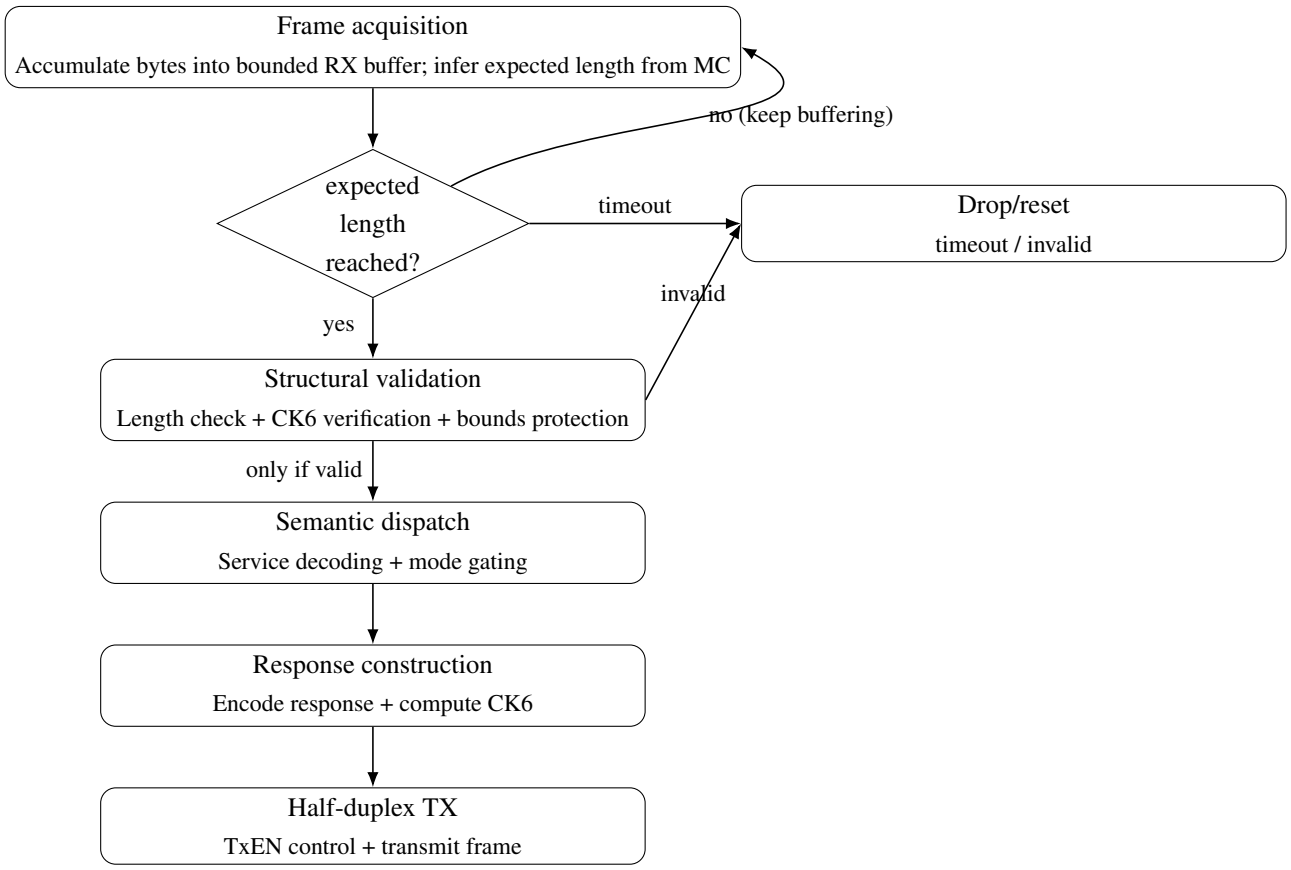


Figure 4.4: Architectural message processing pipeline (conceptual view). Incoming bytes are accumulated into a bounded buffer, the expected length is inferred from the Master Command field, and structural validation (including CK6) is performed before semantic dispatch. Only validated messages reach service handling and response construction, supporting deterministic behavior under malformed and truncated communication [4], [5].

4.6.1 Frame Acquisition

Incoming bytes are accumulated into a bounded receive buffer. Because IO-Link message types have different lengths, the expected message size is inferred from the Master Command field contained in the first byte. Only when the expected number of bytes has been received does the state machine advance to validation.

A timeout mechanism discards partially received frames after a bounded interval. This prevents indefinite blocking and provides a deterministic recovery path under truncated transmissions.

4.6.2 Structural Validation

Before any protocol state is modified, the received frame undergoes structural validation. This includes verification of the inferred frame length, CK6 checksum validation, and explicit protection against buffer overflow. Frames that fail validation are discarded and do not reach semantic

processing.

Checksum validation is performed by extracting the CK6 field from the received frame and comparing it against a locally computed value. Only frames with matching checksum are accepted.

This staged validation strategy ensures that corrupted frames do not alter device state, that operational mode transitions are strictly gated by integrity checks, and that response generation only occurs for verified messages.

4.6.3 Service Dispatch

After structural validation, the Master Command is decoded and mapped to the corresponding service category (cyclic process data, ISDU access, diagnostics, or operational mode control). Operational mode constraints are enforced at this stage: if a service is not permitted in the current mode, the stack generates an appropriate response without mutating internal state.

Importantly, dispatch logic remains strictly at protocol level and does not manipulate transport primitives directly.

4.6.4 Response Construction

Once the requested service is processed, a response frame is constructed in a statically allocated transmit buffer. This stage encodes response data, appends the appropriate service headers, and computes the CK6 checksum before the frame is released for transmission.

Transmission is triggered only after the response is fully assembled. Detailed execution ordering and function-level mapping are presented in Chapter 5.

4.7 ISDU Architectural Model

In addition to cyclic process data exchange, IO-Link defines an acyclic communication mechanism known as ISDU (Indexed Service Data Unit). ISDU allows the master to access device parameters through (index, subindex) addressing.

The detailed normative structure of ISDU transactions was introduced in Chapter 2. The present section focuses exclusively on the architectural realization of ISDU handling within SpecIOLink-Device.

4.7.1 Architectural Scope of ISDU Support

SpecIOLink-Device implements a bounded subset of ISDU functionality, restricted to Type-0 single-frame transactions with a fixed maximum On-Request Data (OD) size and callback-based parameter access. Segmented multi-frame transfers (Type-1 and Type-2) are explicitly excluded.

Type-0 ISDU corresponds to single-frame request/response exchanges. This model is sufficient for parameter access within a bounded payload size and does not require segmentation or reassembly logic.

Segmented transfers (Type-1 and Type-2), which require multi-frame state tracking and buffer management, are intentionally excluded from the declared architectural scope.

4.7.2 Bounded Payload Strategy

A central architectural decision is the enforcement of a fixed maximum payload size for ISDU data (OD field). The OD size is statically bounded to 16 bytes. This constraint preserves predictable worst-case memory usage, avoids dynamic allocation for parameter buffers, and simplifies validation and response construction.

If an ISDU request attempts to exceed the permitted OD size, the response payload is limited to the defined bound. No dynamic resizing occurs.

4.7.3 Callback-Based Parameter Access

ISDU processing is architecturally separated from application logic through a callback interface. The protocol core structurally validates the ISDU frame, extracts index and subindex fields, enforces payload bounds, and invokes the registered ISDU handler to resolve parameter semantics. The returned data is then encoded into the response frame.

The application layer provides only parameter semantics and does not access protocol buffers, modify internal state machines, or interact with transport primitives. This separation prevents application- level code from compromising protocol correctness.

4.7.4 Rationale for Excluding Segmentation

Supporting segmented ISDU transfers would require multi-frame transaction tracking, intermediate buffer accumulation, additional timeout management, and increased state complexity. These mechanisms would increase memory footprint and complicate correctness reasoning and traceability.

Given the objective of a specification-oriented but bounded and inspectable implementation, segmentation was deliberately excluded from the architectural scope. This decision is explicitly declared to avoid any implicit assumption of full ISDU feature coverage.

4.8 Event and Persistence Architecture

In addition to cyclic data exchange and parameter access, IO-Link defines diagnostic event reporting mechanisms. Events allow the device to signal abnormal conditions, state changes, or informational diagnostics to the master.

SpecIOLink-Device implements a bounded and deterministic event management architecture, combined with an integrity-protected parameter persistence mechanism.

4.8.1 Diagnostic Event Architecture

As illustrated in Figure 4.5, events are internally represented as structured entries capturing an event code, a severity qualifier, and optional associated metadata. Rather than dynamically allocating storage, events are stored in a statically allocated, fixed-size array that forms the core of the diagnostic queue.

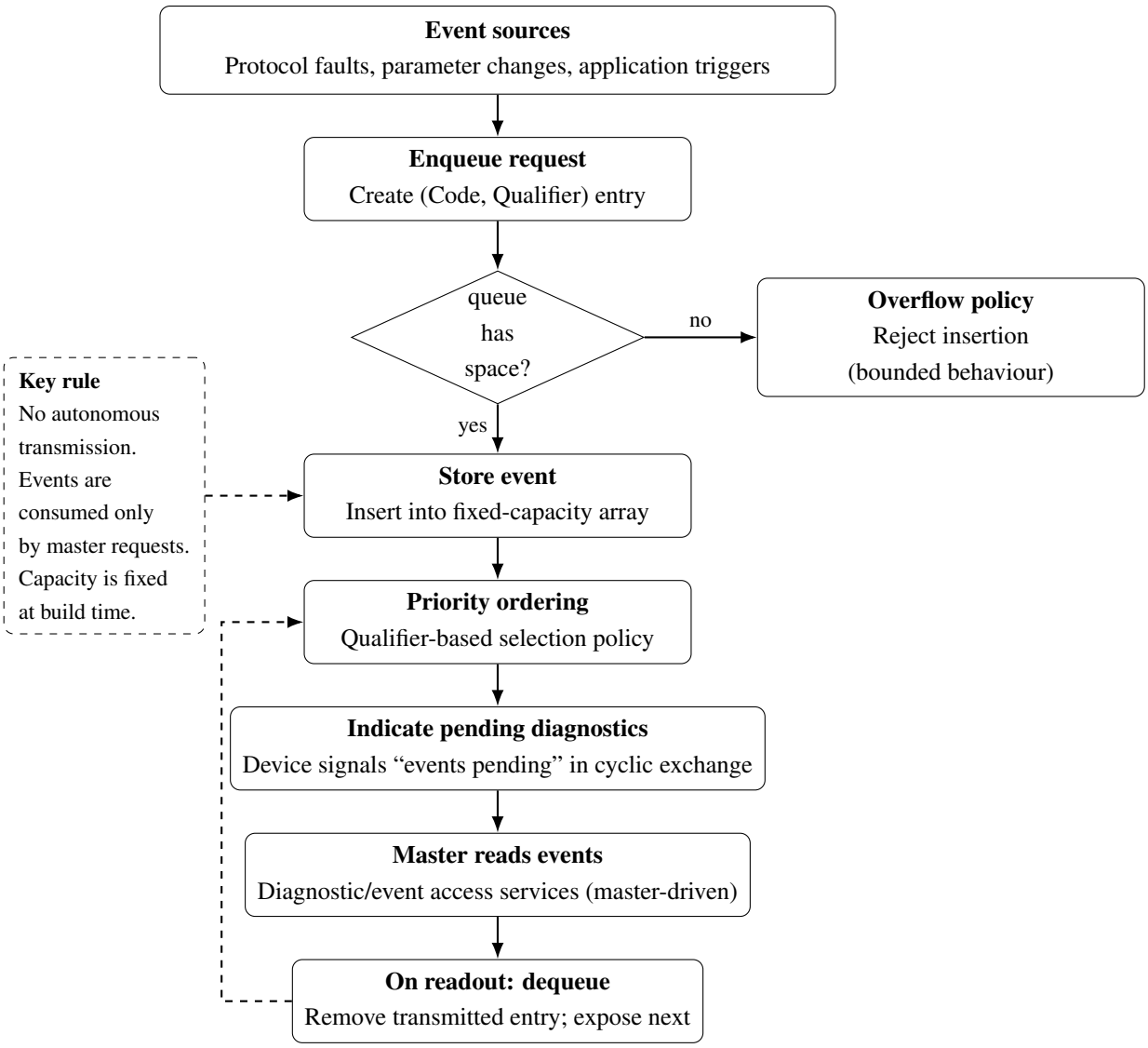


Figure 4.5: Bounded diagnostic event queue architecture (conceptual view). Events are stored in a fixed-capacity structure with severity-based prioritization and a defined overflow policy (reject-on-full). Event delivery is master-driven: pending diagnostics are indicated within cyclic exchange, and detailed event data is retrieved through diagnostic services, after which transmitted entries are removed from the queue [4], [5].

Bounded Queue Design

To enforce the overflow policy depicted in Figure 4.5, the event queue has a compile-time fixed capacity and does not support dynamic growth. Its memory footprint is constant and fully determined at build time. If the queue becomes full, additional events are rejected rather than overwriting unread critical diagnostics. This policy preserves deterministic execution bounds and prevents uncontrolled memory use.

Priority Ordering

Following the priority ordering stage shown in Figure 4.5, events are organized according to their severity qualifier. Higher-severity events are prioritized for transmission, ensuring that critical diagnostics are delivered to the master before informational ones. The ordering mechanism operates entirely within the protocol core and does not require application intervention.

Master-Driven Consumption

As mapped in the retrieval phase of Figure 4.5, event transmission strictly follows the master-driven paradigm of IO-Link. The device does not autonomously transmit events; instead, it signals pending diagnostics within the cyclic exchange. The events are stored internally and later retrieved by the master through explicit diagnostic access services. Retrieved events are then removed from the queue. This behaviour preserves deterministic execution and aligns perfectly with the master-controlled communication model.

4.8.2 Persistence Architecture

As outlined in Figure 4.6, SpecIOLink-Device implements parameter persistence for selected configuration elements using an integrity-protected strategy. Persistent data is written to non-volatile memory together with a CRC32 checksum computed over the stored configuration structure. During initialization, the configuration is loaded and the checksum is recomputed; if a mismatch is detected, deterministic default values are restored.

Persistence is intentionally isolated from the message processing pipeline. Configuration updates are validated at protocol level before being committed to storage, and persistence operations do not directly alter runtime control state or operational mode.

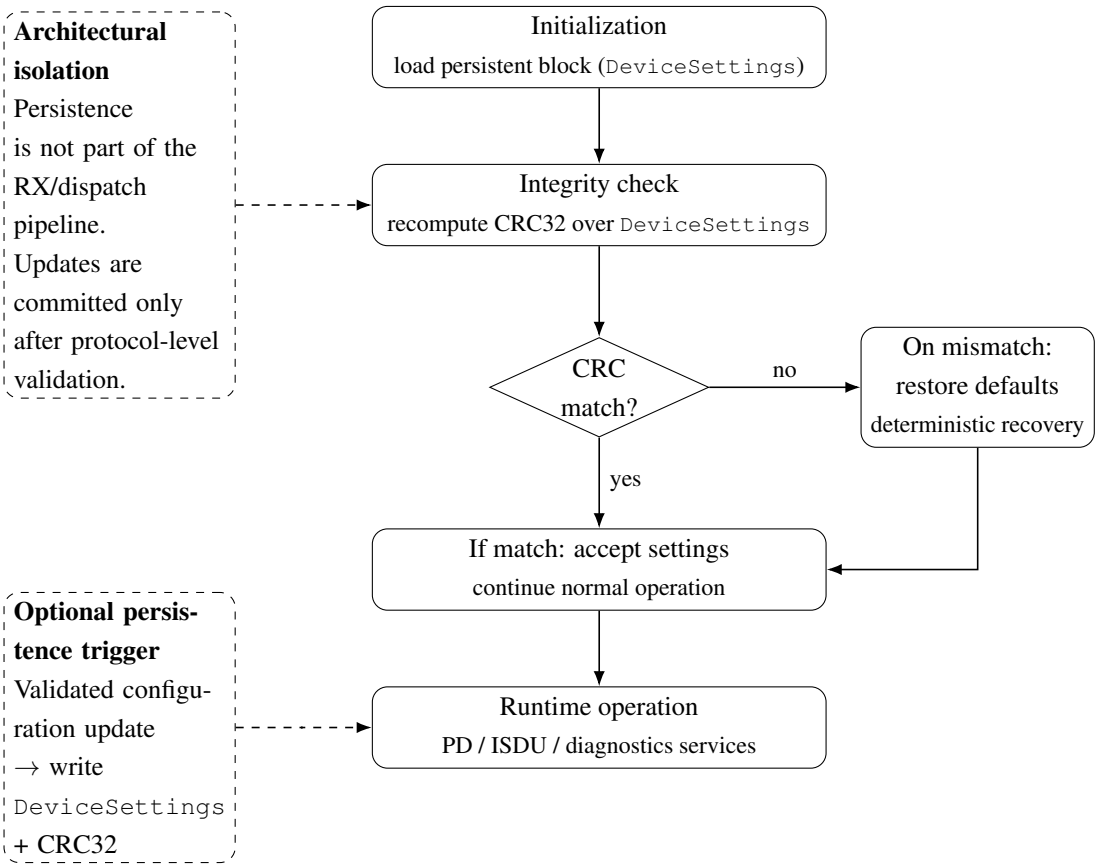


Figure 4.6: Persistence and integrity verification architecture (conceptual view). Selected configuration parameters are stored as a fixed-layout persistent block together with a CRC32 checksum. During initialization, the block is loaded and verified; on mismatch, deterministic default values are restored. Persistence operations are architecturally isolated from the message processing pipeline to prevent unintended coupling with runtime control and operational mode logic.

CRC-Based Integrity Strategy

As shown in the initialization flow of Figure 4.6, a CRC32 checksum is computed over the persistent configuration structure and stored alongside the configuration data. During device initialization, the configuration is loaded and the checksum is recomputed; if a mismatch is detected, deterministic default values are restored. This guarantees that corrupted or partially written persistent data does not propagate into operational behaviour.

Architectural Isolation

Consistent with the isolation boundaries depicted in Figure 4.6, persistence operations do not directly modify runtime control state or operational mode. Configuration updates are validated at the protocol level before being committed to storage, and storage operations are kept strictly isolated from frame parsing and mode-transition logic to prevent unintended coupling.

4.9 Architectural Constraints and Declared Limitations

The SpecIOLink-Device architecture was intentionally designed as a bounded, specification-oriented subset of IO-Link V1.1.4.

The following constraints and exclusions are explicitly declared.

4.9.1 Excluded Features

The architecture does not implement:

- Segmented ISDU transfers (Type-1 and Type-2 multi-frame exchanges);
- Dynamic negotiation of communication speed;
- Certified compliance mechanisms;
- Full recovery handling for COMLOST scenarios;
- Industrial 24 V PHY integration within the protocol core.

Segmented ISDU support would require multi-frame state tracking and intermediate buffer management, which contradicts the bounded-memory and deterministic-execution objectives.

Communication speed selection (COM1, COM2, COM3) is configured statically at initialization time through the transport binding. The protocol core does not dynamically negotiate COM rate.

4.9.2 COMLOST and Wake-Up Handling

The architecture includes structural support for wake-up detection and communication timeout monitoring. However, wake-up interrupt handling was not exercised during validation, and full COMLOST recovery sequences were not experimentally validated. In addition, timeout-based recovery logic is structurally present but disabled in the validation configuration.

Accordingly, no claim is made regarding complete COMLOST robustness under industrial disturbance conditions.

4.9.3 Physical Layer Abstraction

The protocol core assumes a compliant UART-based transport interface configured according to IO-Link framing requirements (8E1).

Industrial transceiver behaviour and 24 V signalling are abstracted from the protocol core and are not modelled at the architectural level.

This abstraction allows isolation of data Link layer behaviour from analog signalling considerations.

4.9.4 Compliance Statement

SpecIOLink-Device is a specification-oriented implementation. It does not claim certified compliance with IO-Link V1.1.4.

All implemented behaviours are documented within a clearly bounded scope. Features not implemented are explicitly declared.

This approach prioritizes transparency, traceability, and experimental reproducibility over feature completeness.

4.10 Summary

This chapter defined the architecture of the SpecIOLink-Device stack.

The design is characterized by explicit layered separation, dual state modelling, staged message validation prior to state mutation, bounded ISDU support, fixed-capacity diagnostic event management, CRC32-protected persistence, and complete avoidance of dynamic memory allocation.

Implemented behaviours follow normative IO-Link constraints within a clearly declared subset, while segmentation, certified compliance mechanisms, and full COMLOST recovery remain outside the architectural scope.

The next chapter presents the concrete implementation on the RP2040 platform and details the validation methodology used to exercise the architecture defined herein.

Chapter 5

Implementation and Validation

5.1 Introduction

This chapter presents the concrete implementation details and experimental validation results of the architecture introduced in Chapter 4. The implementation targets a specification-oriented subset of the IO-Link V1.1.4 standard and focuses on protocol-level correctness, robustness, and experimental verifiability rather than on immediate industrial deployment.

Particular emphasis is placed on the internal mechanisms of the device firmware based on the `lwIOLink` library and on the validation methodology enabled by a fully controllable, custom-developed IO-Link Master. In this chapter, `lwIOLink` is treated strictly as an open-source C++ library providing a device-side IO-Link protocol stack baseline. The term "stack" therefore refers to the integrated protocol-layer implementation built on top of this library and extended within the scope of this thesis. This approach allows detailed inspection of protocol behavior during startup, cyclic and acyclic communication, error handling, and fault conditions.

Rather than relying on commercial IO-Link Masters as the primary validation instrument, a dedicated master implementation was developed in Python and used as the main test driver in the final validation campaign. This choice enables deterministic execution of the IO-Link startup handshake within the defined validation scope, systematic fault injection, explicit checksum verification, and detailed observation of device responses under both nominal and erroneous operating conditions.

5.1.1 Validation Methodology and Hardware Evolution

The validation process followed a staged methodology designed to progressively isolate protocol-level behavior from physical-layer variables. In this context, *physical-layer variables* refers to non-protocol effects that can influence observed behavior, such as transceiver direction-control timing (TX_EN propagation and turn-around behavior), analog signal integrity on the C/Q line, and electrical-layer dynamics associated with 24 V signalling and three-wire cabling.

The initial experimental configuration employed a full industrial IO-Link physical layer implementation. On the master side, an IO-Link Master HAT (IOLHAT) equipped with a MAX14819

dual-channel transceiver was connected to a Raspberry Pi 4. On the device side, a Raspberry Pi Pico (RP2040) was interfaced with a Texas Instruments TIOX1X2XEVM evaluation board configured to activate the TIOL1123 IO-Link device transceiver.

This configuration enabled full 24 V single-drop communication over standard three-wire industrial cabling (L+, L-, C/Q). Although full 24 V single-drop communication was successfully established, the device did not consistently complete the Startup → Operate transition under certain write-command sequences. Analysis indicated timing sensitivities related to transmit-enable (TX_EN) direction control and analog signal behavior at the physical layer. These effects introduced non-deterministic behavior that complicated isolation of data link layer state-machine logic during debugging.

To isolate the protocol stack from these electrical and timing variables, a methodological pivot was performed. Here, *methodological pivot* denotes a deliberate change in the experimental method (not a change in protocol scope): the industrial transceivers were temporarily removed, and the master and device were interconnected directly via a 3.3 V UART interface while *operating the link in half-duplex at the protocol level*. Concretely, although UART peripherals are electrically capable of full-duplex communication, the testbed was configured and driven to emulate IO-Link's master-driven half-duplex exchange semantics (request then response, with controlled direction switching and no concurrent bidirectional payload exchange). This approach eliminated 24 V switching dynamics, direction-control propagation delays, and analog signal integrity effects.

Methodological rationale for the physical-layer pivot. The temporary removal of the industrial transceiver layer was a deliberate methodological decision rather than a functional limitation. Early integration attempts revealed non-deterministic timing sensitivities related to directional control (TX_EN) and analog signal behavior on the C/Q line, which made it difficult to isolate software state-machine faults from electrical-layer effects. By introducing a direct 3.3 V UART pivot while preserving COM2 framing (38.4 kbaud, 8E1), the data link layer could be validated under fully deterministic and observable conditions.

This separation enabled one-to-one correlation between master logs and device trace outputs, precise frame freezing during debugging, and controlled fault injection scenarios. Once data link correctness was experimentally confirmed, the physical-layer reintegration was identified as future work rather than a blocking dependency for protocol validation.

5.2 lwIOLink Stack Implementation

5.2.1 Core Library Structure

The device firmware is implemented in `C++` and is based on the open-source `lwIOLink` library. In this thesis, `lwIOLink` is treated as an *upstream codebase providing a lightweight device-side IO-Link stack implementation* (i.e., a protocol stack realized as a library). The upstream codebase was adopted as a starting point and subsequently extended, corrected, and restructured to support a specification-oriented subset of IO-Link V1.1.4 behavior.

The public application programming interface is declared in `lwIOLink.hpp`, while protocol logic, operational state handling, and communication routines are implemented in `lwIOLink.cpp`. Application-specific behavior is confined to `main.cpp`, ensuring a clear separation between protocol logic and sensor-level functionality.

Dynamic memory allocation is deliberately avoided throughout the implementation. All protocol buffers, state variables, and data structures are statically allocated at compile time, guaranteeing deterministic memory usage and preventing heap fragmentation. This design choice is particularly relevant for embedded systems, where predictable behavior and robustness under fault conditions are critical.

5.2.2 Operational Mode Handling and Frame Validation

The implementation explicitly models the IO-Link operational modes *Startup*, *Preoperate*, and *Operate*. The active operational mode is tracked by an internal state variable and is used to gate the availability of protocol services in accordance with the IO-Link V1.1.4 specification within the implemented subset.

Cyclic Process Data exchange is enabled exclusively in *Operate* mode, while acyclic services, including indexed access and ISDU transactions, are permitted in *Preoperate* and *Operate*. Transitions between operational modes are triggered by valid Master Commands received during the startup handshake and are applied only after successful validation of the corresponding communication frames.

All received UART frames are interpreted as IO-Link M-sequences and are subjected to a multi-stage validation process:

- verification of frame length based on the Master Command field;
- parity verification performed by the UART hardware using 8E1 framing;
- protocol-level CK6 checksum validation.

Frames that fail validation are discarded without intentional operational-mode transitions under the evaluated test conditions. This behavior improves robustness against corrupted or malformed traffic and reduces the likelihood of protocol desynchronization during controlled experiments.

The stack contains an initial implementation of wake-up handling and COMLOST/timeout monitoring consistent with the normative state model. However, the final validation campaign relies on the 3.3 V UART pivot (i.e., without transceiver/CQ-line wake-up signaling) and assumes continuous master–device communication. Therefore, wake-up behavior and automatic COMLOST recovery actions were not exercised experimentally; in the adopted test configuration, the corresponding recovery transitions were disabled (commented out) to prevent autonomous state changes and to keep validation focused on traceable data link layer behavior.

5.2.3 ISDU Handling and Transport Modes

Acyclic communication is implemented using Type-0 (on-request) ISDU services. The stack supports indexed read and write operations with explicit and statically enforced bounds on payload size.

Two ISDU transport modes are exercised during validation:

- **Direct Type-0 ISDU in Preoperate**, in which ISDU frames are exchanged synchronously outside cyclic communication;
- **Cyclic ISDU embedding in Operate**, in which ISDU requests and responses are transported within the On-Request Data (OD) field of cyclic frames.

ISDU requests are decoded, validated, and dispatched to application-level callback functions. If a request cannot be handled, the device returns a protocol-consistent error indication within the implemented subset.

Within the evaluated prototype, ISDU handling is implemented for a bounded set of indexed data objects exercised during validation, including Vendor Name (0x0010), Product Name (0x0012), Application Tag (0x0018), and Parameter Checksum (0x001A). In addition, a non-normative experimental index (0x9999) is implemented exclusively to support controlled fault injection and diagnostic event testing.

Segmented ISDU transfers (Type-1 and Type-2 with multi-frame segmentation) are not implemented and are considered out of scope for the present prototype.

5.2.4 Persistent Parameter Storage

Persistent storage is implemented for a selected subset of identification-related parameters, including Vendor Name, Product Name, and Application Tag. These parameters are stored in non-volatile memory and are automatically restored during device initialization.

To provide integrity verification, a CRC32 checksum is computed whenever persistent parameters are updated. During startup, the stored checksum is verified before parameters are accepted. In the event of a checksum mismatch, default values are restored, thereby protecting the device from corrupted non-volatile memory contents and incomplete write operations.

In the evaluated prototype, non-volatile storage is accessed through the firmware's EEPROM abstraction layer. The architectural role of this mechanism is to provide a byte-addressable persistent block; integrity and recovery semantics are enforced at the application/stack level via CRC32 and deterministic defaults, independently of the underlying non-volatile medium.

5.2.5 Event Generation and Diagnosis Handling

Events are buffered in a fixed-size internal queue and ordered according to their Qualifier field, ensuring that higher-severity conditions are reported before lower-priority notifications. Event delivery is performed through the *Diagnosis* channel using DIAG read operations exercised in the validation environment.

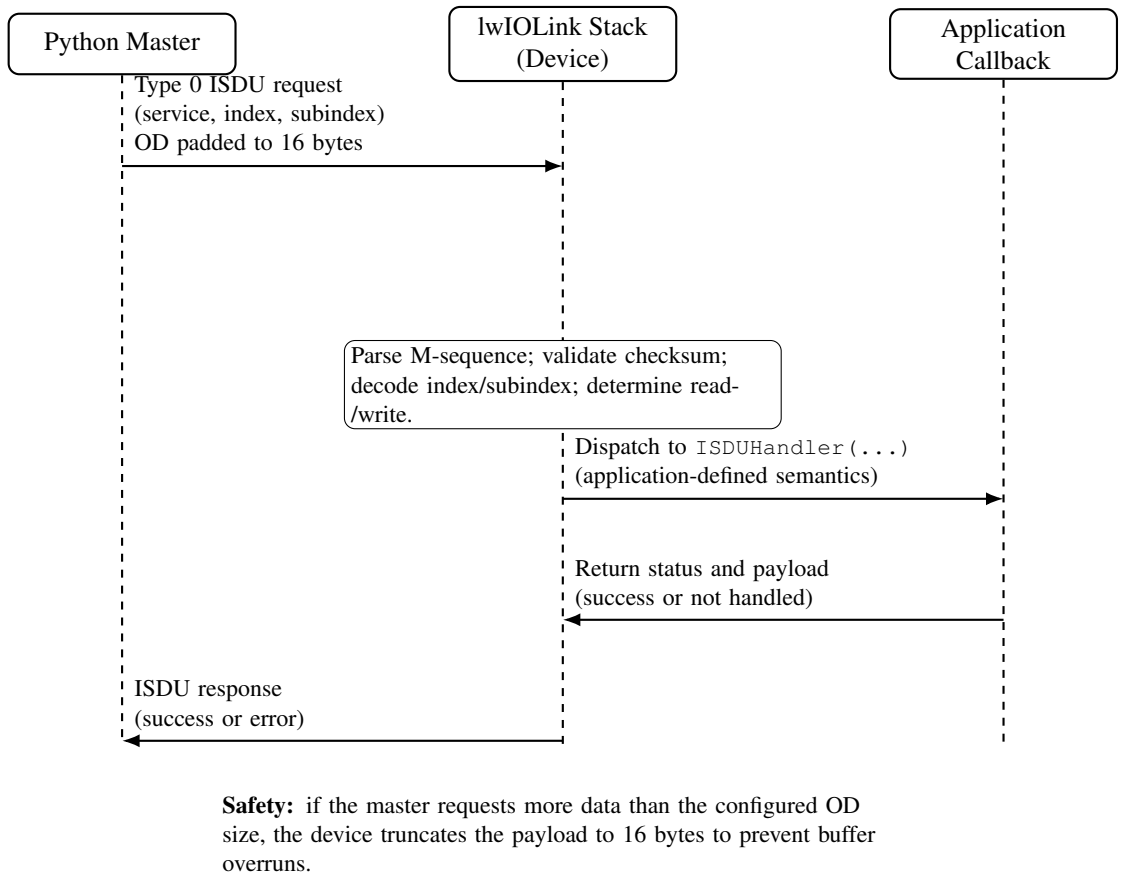


Figure 5.1: ISDU service flow in the validation environment (IO-Link V1.1.4). The master issues a Type 0 on-request transaction; the device validates and decodes it, dispatches parameter semantics via a callback, and returns a success or error response.

In the implemented subset, the *Diagnosis* readout is realized as a small set of master commands that retrieve diagnosis-related bytes from the device. In the traces, these appear as `MC=0xC0`, `MC=0xC1`, and `MC=0xC2`. Conceptually, these reads allow the master to retrieve the pending event code and its qualifier in a deterministic, byte-wise manner. (The specific register mapping is a protocol-level design point of the implemented subset and is made traceable through the captured logs and the sequence diagrams in Section 5.4.)

An event remains pending in the queue until its corresponding *Diagnosis* registers have been successfully read by the Master. Upon readout, the event is considered consumed and is removed from the queue, thereby allowing subsequent lower-priority events to be reported.

This behavior was experimentally validated by issuing consecutive DIAG read commands until the expected event code (e.g., 0x5000 or 0x6350) was returned, followed by additional reads to confirm that the same event was no longer reported unless explicitly retriggered. The observed traces confirm correct implementation of event queuing, prioritization, and clearing semantics within the scope of the implemented IO-Link V1.1.4 subset. For validation purposes, additional diagnostic conditions may be artificially triggered via the non-normative experimental index 0x9999, allowing controlled verification of event queuing and clearing behavior.

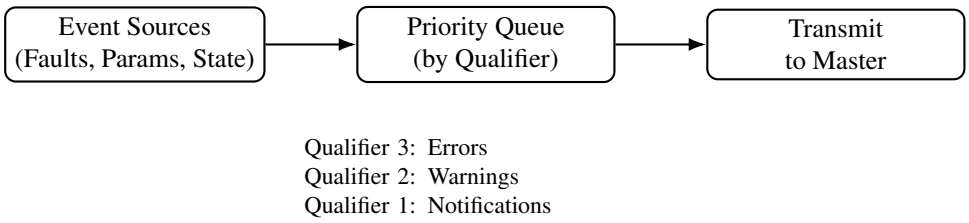


Figure 5.2: Priority-based event handling mechanism implemented in the lwIOLink stack. Diagnostic events are queued and transmitted in order of severity, ensuring that critical conditions are reported first.

5.2.6 Safety Mechanisms and ISDU Payload Bounding

To prevent memory overruns during acyclic communication, the on-request data field (OD) is configured to a fixed size of 16 bytes in all operational modes.

Since ISDU responses include a one-byte Service/Length header, the maximum usable ISDU data payload per transaction is limited to 15 bytes. If a requested indexed data object exceeds this bound, the returned payload is safely truncated. This controlled truncation preserves deterministic behavior while preventing buffer overruns and undefined memory access.

Concrete frame examples (trace excerpts). A recurring difficulty for the reader is understanding *what bytes are actually exchanged*. To address this, the following trace excerpts illustrate typical request/response shapes used throughout Chapter 5:

- **Startup read (example: MinCycleTime).** Device trace shows a request starting with `MC=0xA2` and a short response carrying a single data byte. In the captured logs, this appears as: `[RX] MC=0xA2 ...` followed by `[TX] A2 .. 59`, where `0x59` is the returned value.
- **Mode transition (example: PREOPERATE / OPERATE).** Requests are visible as `MC=0x20` frames carrying a command byte (e.g., `0x9A` then `0x99`) and acknowledged by a deterministic short response; device-side traces explicitly log `Mode -> PREOPERATE` and `Mode -> OPERATE`.
- **Cyclic exchange (Operate).** In the traces, cyclic frames appear as `MC=0x80` with a fixed expected length (e.g., 19 bytes in the used OD/PD configuration), followed by a device response carrying OD, PD, and Status bytes.

These examples are not intended as a normative clause restatement; they are included to anchor subsequent test discussions in observable byte-level evidence.

5.3 Validation Setup and Methodology

5.3.1 Custom Python-Based IO-Link Master

Validation was performed using a custom IO-Link Master implemented in Python and executed on a Raspberry Pi 4[1]. Communication with the device is performed via a direct 3.3 V UART connection configured for half-duplex operation at the protocol level and IO-Link COM2 framing parameters (38.4 kbaud, 8E1).

The Master implements the complete IO-Link startup handshake, including parameter discovery, mode transitions, and cyclic data exchange within the validation scope. It supports cyclic Process Data communication, direct and cyclic ISDU access, event reception, and deliberate fault injection.

In addition to hardware-based validation, a mock UART backend is provided in the master codebase (invoked as an alternative execution path). This backend enables protocol-level verification without physical hardware by emulating transport-level delivery and allowing deterministic reproduction of edge cases (e.g., truncated frames, injected checksum errors) without hardware variability. In the provided runner script, this functionality is selected explicitly (or as a fallback when the serial port is unavailable).

5.3.2 Test Mechanisms and Instrumentation

The custom Python-based master provided fine-grained control over frame generation and timing. Test mechanisms included:

- Explicit construction of M-sequences with controllable Master Command fields;
- Injection of frames with invalid CK6 checksums;
- Transmission of truncated frames to test parser reset logic;
- Oversized ISDU payload requests to verify bounding behavior;
- Deliberate repetition of mode-transition commands;
- Controlled generation and reading of diagnostic events.

A mock UART backend is also available (see Section 5.3.1) to enable deterministic, hardware-independent reproduction of edge cases.

All validation experiments were systematically re-executed following the addition of new features to ensure the absence of regressions. Master-side logs and device-side debug traces were strictly correlated to confirm deterministic internal state evolution.

[1] The master was implemented by the author of this thesis.

5.3.3 Test Methodology

Validation tests were designed to cover both normal and abnormal situations. Tests were executed repeatedly to ensure reproducibility, and communication logs were captured on both the Master and Device sides.

The following test categories were defined:

- IO-Link startup and operational mode transition validation;
- cyclic communication stability;
- indexed parameter read and write correctness;
- parameter persistence across power cycles;
- event generation and prioritization;
- robustness against malformed requests and oversized ISDU transfers.

5.3.4 Normative Validation Logic

Validation against IO-Link V1.1.4 was performed by mapping implemented mechanisms to the corresponding normative behavioral requirements defined at the data link layer within the implemented subset.

Rather than claiming full certification-level compliance, validation focused on verifying the following categories of normative behavior:

- Correct execution of the startup handshake sequence;
- Proper enforcement of operational mode transitions (Startup $\rightarrow$ Preoperate $\rightarrow$ Operate);
- Deterministic handling of M-sequences with correct CK6 checksum validation;
- State-dependent service gating;
- Correct formatting of cyclic process data frames;
- Structured ISDU request and response handling;
- *Diagnosis* channel behavior and event prioritization;
- Deterministic timeout monitoring without unintended state corruption.

Table 5.1: Mapping between implemented mechanisms and validated IO-Link V1.1.4 behavioral requirements (data link layer subset).

Normative Behavior Category	**Implemented Mechanism**	**Validation Method**	**Result**
Startup handshake sequencing	Explicit operational mode state machine; Master Command decoding; guarded mode transitions	Master-driven startup sequence execution; verification of correct Startup → Preoperate → Operate progression	Observed deterministic transitions within implemented scope
Operational mode gating	Service availability conditioned by deviceMode variable	Attempted service invocation in invalid modes; verification of correct rejection	State-dependent gating confirmed
M-sequence framing	Length inference from Master Command; bounded RX buffer	Injection of truncated frames and incorrect length frames	Malformed frames discarded without state corruption
CK6 checksum validation	Protocol-level CK6 calculation and comparison prior to dispatch	Injection of invalid checksum frames	Incorrect frames rejected; no unintended transitions observed
Cyclic process data exchange	Operate-mode cyclic processing and response construction	Sustained cyclic communication tests under COM2 configuration	Stable cyclic exchange observed
Type-0 ISDU handling	Indexed dispatch via ISDU callback interface; bounded OD field	Read/write access to implemented indices; oversized request injection	Correct responses generated; bounded truncation enforced
Diagnostic event signaling	Bounded priority queue; qualifier-based ordering; diagnosis read clearing	Generation of multiple events and sequential diagnosis reads	Correct priority ordering and controlled clearing observed
Persistence and integrity	CRC32 verification during initialization; default restoration on mismatch	Parameter modification, power cycle, checksum validation	Deterministic restoration and event signaling confirmed
Timeout monitoring (Operate)	LastMessage timestamp monitoring; recovery logic structurally present	Extended silence intervals during cyclic operation	Timeout detection observed; recovery logic intentionally disabled to prevent automatic state transitions during validation

Table 5.1 provides the high-level mapping between the implemented mechanisms and the validation methods used in the test campaign. Each category was exercised through explicit master-side test scripts. Communication traces were captured and compared against expected frame structures derived from the IO-Link V1.1.4 specification, within the explicitly implemented subset.

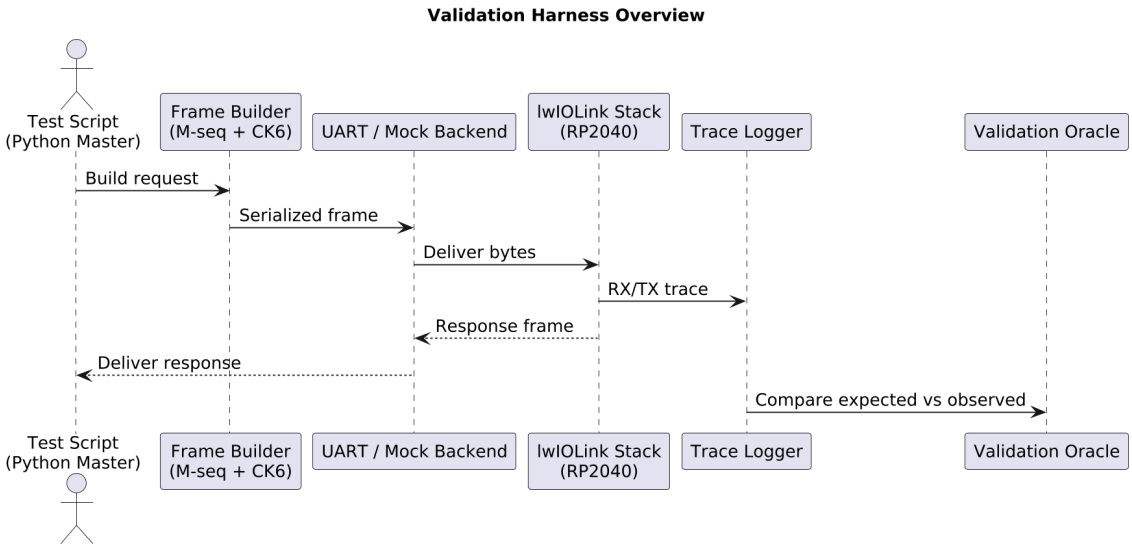


Figure 5.3: Validation harness overview illustrating deterministic frame generation, trace capture, and behavioral comparison. In this context, the "comparison" component denotes a rule-based expected-versus-observed trace checker (test oracle in the software-testing sense), implemented as part of the Python master's validation logic rather than an external certified conformance tool.

5.4 Validation Results

A recurring requirement for clarity is that each performed test should be accompanied by a sequence-level description of the interaction. For this reason, the validation results are organized below by test, each anchored by a UML sequence diagram that reflects the executed master–device exchange and the trace-aligned outcomes.

5.4.1 Test 1: Basic Communication (Startup & Cyclic)

Test 1 validates that the device deterministically completes the startup handshake (Startup → Pre-operate → Operate) and sustains cyclic exchange under the adopted OD/PD sizing and COM2 framing. As detailed in Figure 5.4, the expected cyclic frame length (e.g., 19 bytes in the used configuration) is treated as a traceable property, visible in both master logs and device traces.

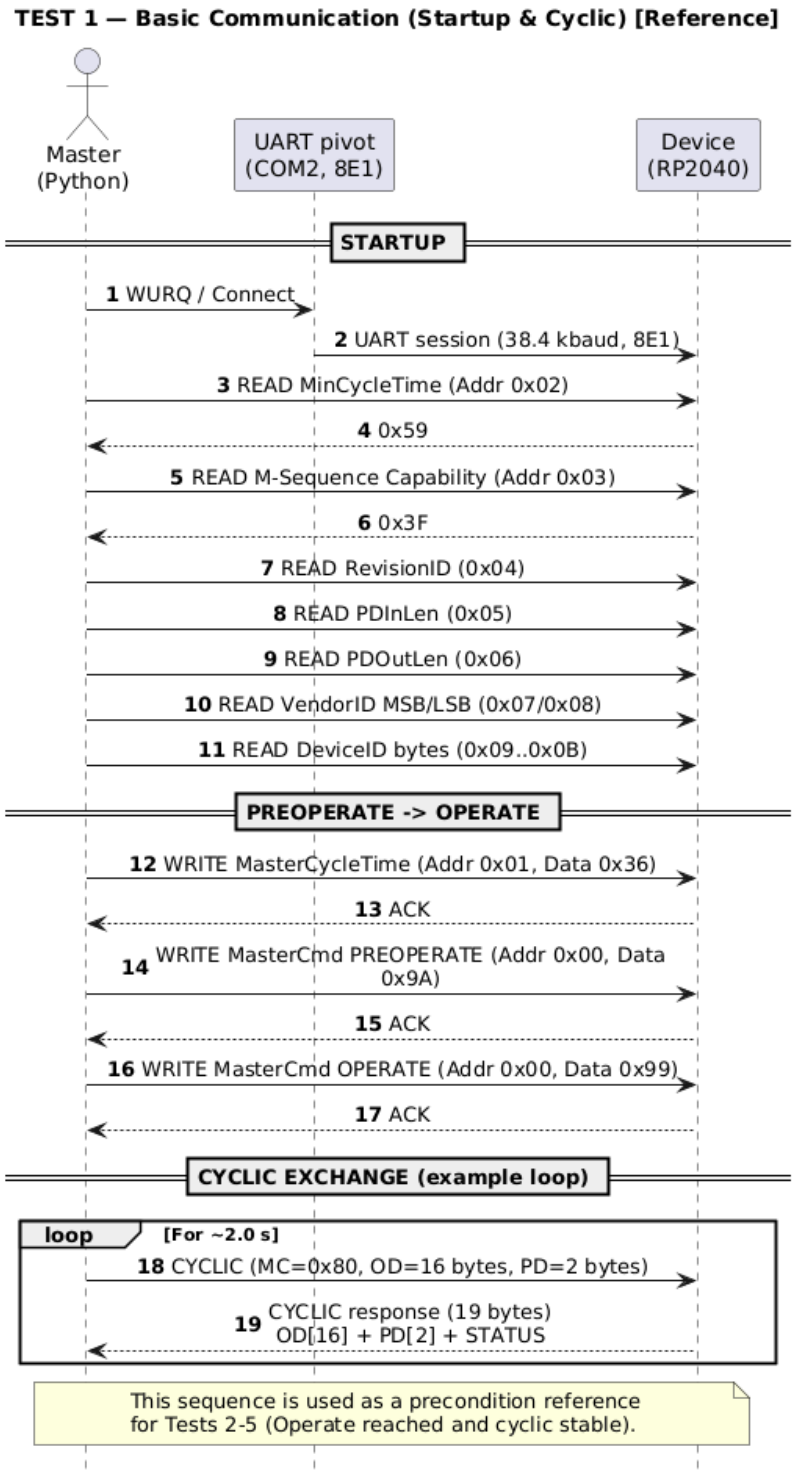


Figure 5.4: Test 1 (Reference): Startup handshake, mode transitions (Startup → Preoperate → Operate), and stable cyclic exchange under COM2 configuration.

Observed communication stability

Stable IO-Link COM2 communication was observed throughout the validation experiments. Within the evaluated testbed, response times remained within the expected range for the selected UART configuration, and no checksum mismatches or state desynchronization were observed in the captured communication traces during the executed validation campaigns.

These observations are specific to the described validation environment and do not constitute unconditional timing or performance guarantees. These observations were obtained in the 3.3 V UART pivot environment described in Section 5.1.1 and therefore exclude industrial 24 V PHY effects.

5.4.2 Test 2: Data Persistence (Write Tag → Reset → Read)

Test 2 is organized in phases. The Startup handshake and mode transitions are treated as reference (see Fig. 5.4); the phases below focus on (B) write + immediate RAM readback, (C) reset and reconnection, and (D) post-reset readback from non-volatile storage. The sequence diagrams for these phases (Figures 5.5, 5.6, and 5.7) illustrate the step-by-step execution. Notably, Figure 5.6 explicitly captures the expected "Incomplete RX" phenomenon caused by the device reboot interrupting the response frame, which is treated as an expected artifact of reset rather than a protocol failure.

5.4.3 Test 2: Persistence and *Data Storage* Verification

Writing a new Application Tag value followed by a device reset resulted in correct restoration of the parameter after restart, confirming correct operation of non-volatile storage and CRC32-based integrity verification. The Application Tag corresponds to the indexed parameter at Index 0x0018 within the implemented object dictionary and is defined as a persistable identification parameter in the validated subset. Changes to persistent parameters correctly triggered event 0x6350, validating the *Data Storage* signaling mechanism.

The "Incomplete RX" message observed immediately after issuing the reset command corresponds to the expected interruption of UART reception caused by the device reboot sequence. Since the microcontroller resets before completing the response frame, the master detects an incomplete reception. This behavior is inherent to the reset operation and does not indicate a protocol-level communication fault.

Table 5.2 summarizes the trace-based evidence for Test 2 and is referenced here to ensure the table is not isolated from the narrative.

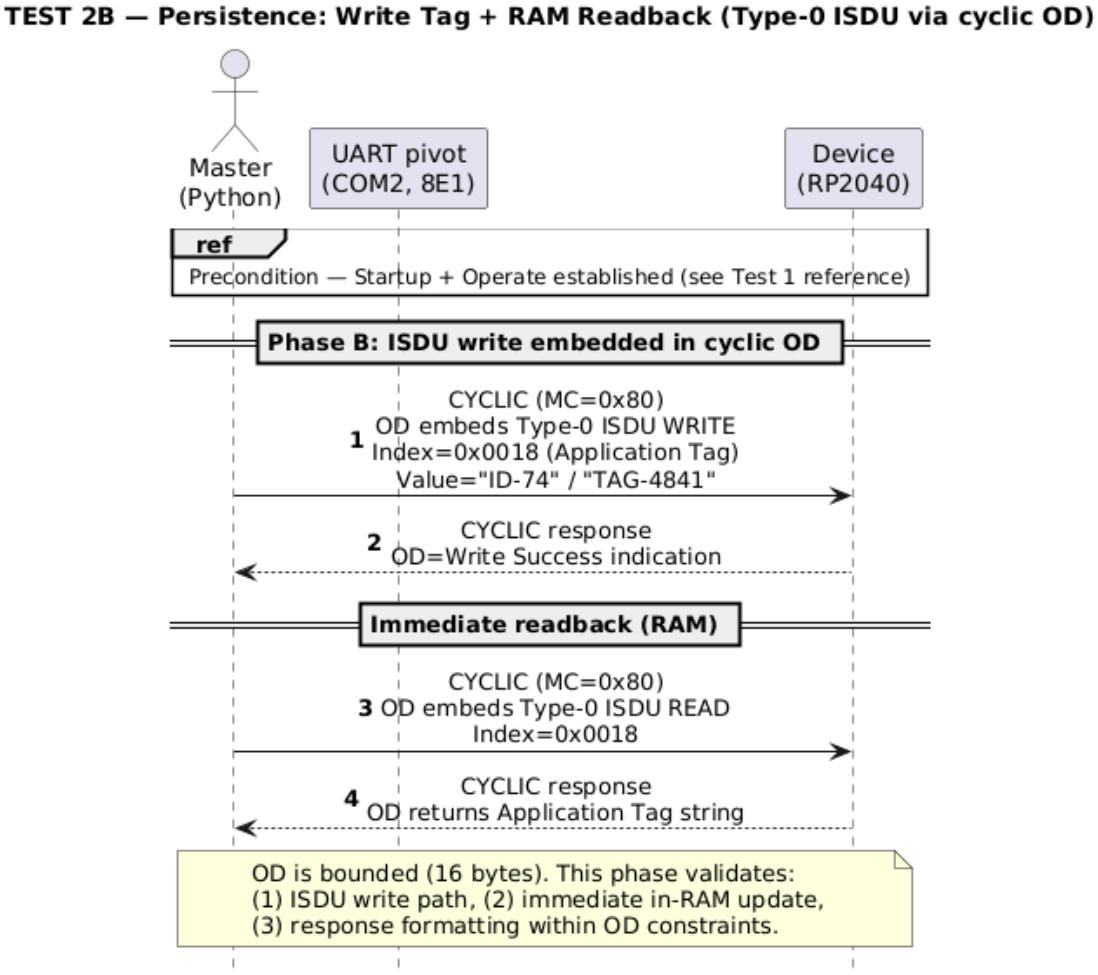


Figure 5.5: Test 2 (Phase B): Type-0 ISDU write to Index 0x0018 (Application Tag) embedded in cyclic OD, followed by immediate readback from RAM.

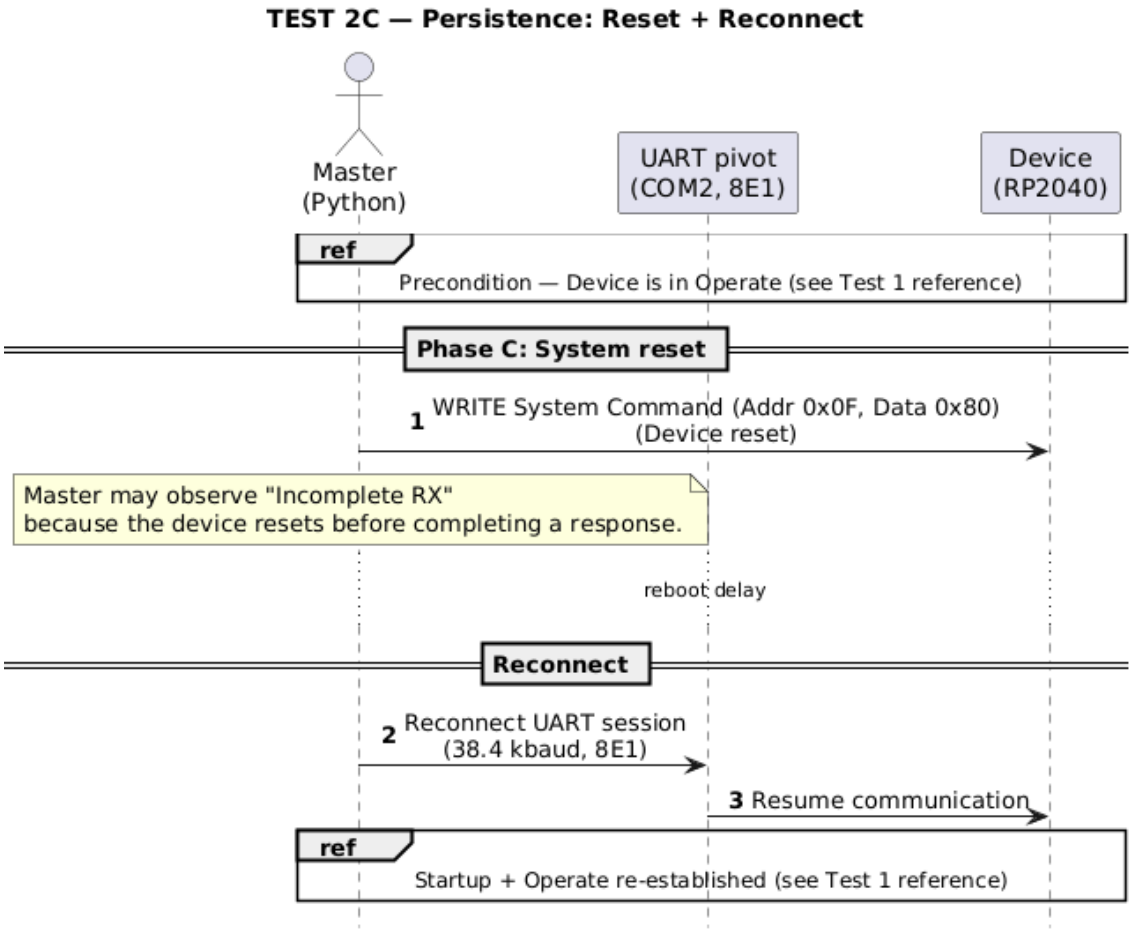


Figure 5.6: Test 2 (Phase C): System reset (System Command 0x80) and reconnection sequence. The Startup handshake is treated as reference (see Fig. 5.4).

Table 5.2: Trace-based validation of ISDU write, reset cycle, and non-volatile persistence (Test 2).

Phase	**Master action**	**Observed master log (excerpt)**	**Device trace (excerpt)**	**Validated mechanism**
Startup	Read identification parameters	`READ MinCycleTime (Addr 0x02); MinCycleTime = 0x59`	`[RX] MC=0xA2 ... → [TX] A2 .. 59`	Deterministic STARTUP handshake and parameter access prior to mode transition.
Mode transition	Write MasterCommand: PREOPERATE and OPERATE	`WRITE Master Command: PREOPERATE (0x9A); OPERATE (0x99)`	`[RX] MC=0x20 .. 9A → Mode -> PREOPERATE [RX] MC=0x20 .. 99 → Mode -> OPERATE`	Mode transitions occur only upon validated command frames; service availability is gated by mode.
ISDU write (cyclic OD)	Write Application Tag (Index 0x0018) embedded in OD	`Including ISDU in OD (9/16): [A5 00 18 00 49 44 2D 37 34]`	`[RX] MC=0x80 ... A5 00 18 00 49 44 2D 37 34 ...`	Type-0 ISDU write via cyclic OD embedding; bounded OD payload (16 bytes).
Immediate readback	Read Application Tag (Index 0x0018)	`Including ISDU in OD (4/16): [90 00 18 00] Read Success: ID-74`	`[TX] 15 49 44 2D 37 34 ...`	Correct in-RAM update and response formatting within OD constraints.
Reset	Write SystemCommand 0x80 (device reset)	`WRITE System Command (Addr 0x0F, Data 0x80); Incomplete RX.`	(Reboot interrupts UART RX as expected)	Reset causes expected reception interruption; not a protocol-level failure.
Post-reset verification	Re-run startup and read Application Tag from NVM	`Re-establishing Communication...; Read Success: ID-74`	`[RX] MC=0xA2 ... then [RX] MC=0x80 ... [90 00 18 00] [TX] 15 49 44 2D 37 34 ...`	Persistence confirmed: value survives reset cycle and is returned after re-handshake.

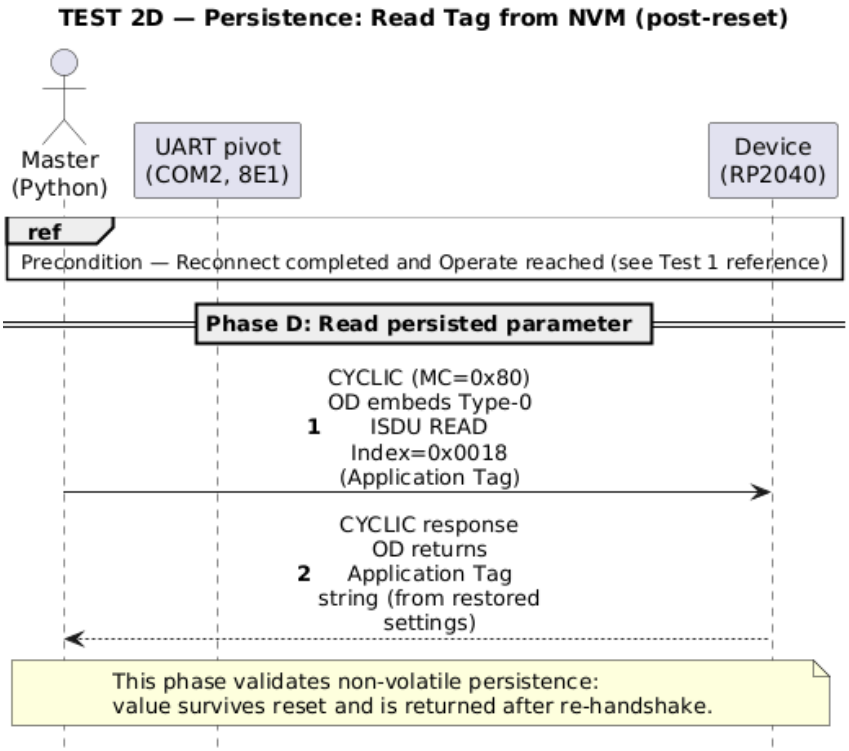


Figure 5.7: Test 2 (Phase D): Post-reset ISDU read of Index 0x0018 confirming non-volatile persistence of the Application Tag.

5.4.4 Test 3: Factory Reset (System Command 0x82)

Test 3 validates the factory reset mechanism invoked by a system command (0x82), followed by reconnection attempts during Startup and verification that the Application Tag returns to its deterministic default value. The sequence diagrams (Figures 5.8, 5.9, and 5.10) highlight the reset and recovery process. In particular, Figure 5.9 shows that immediate post-reset reads may fail transiently and require retries until the device returns to a stable Startup response pattern.

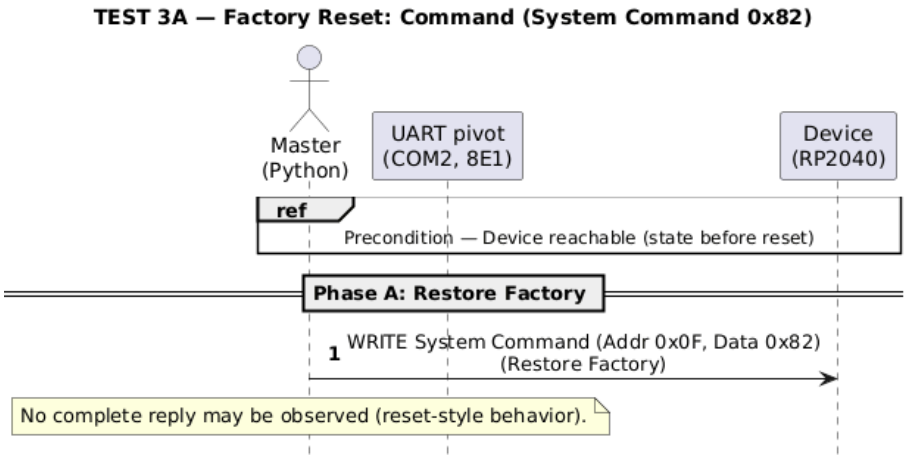


Figure 5.8: Test 3 (Phase A): Factory reset via System Command 0x82.

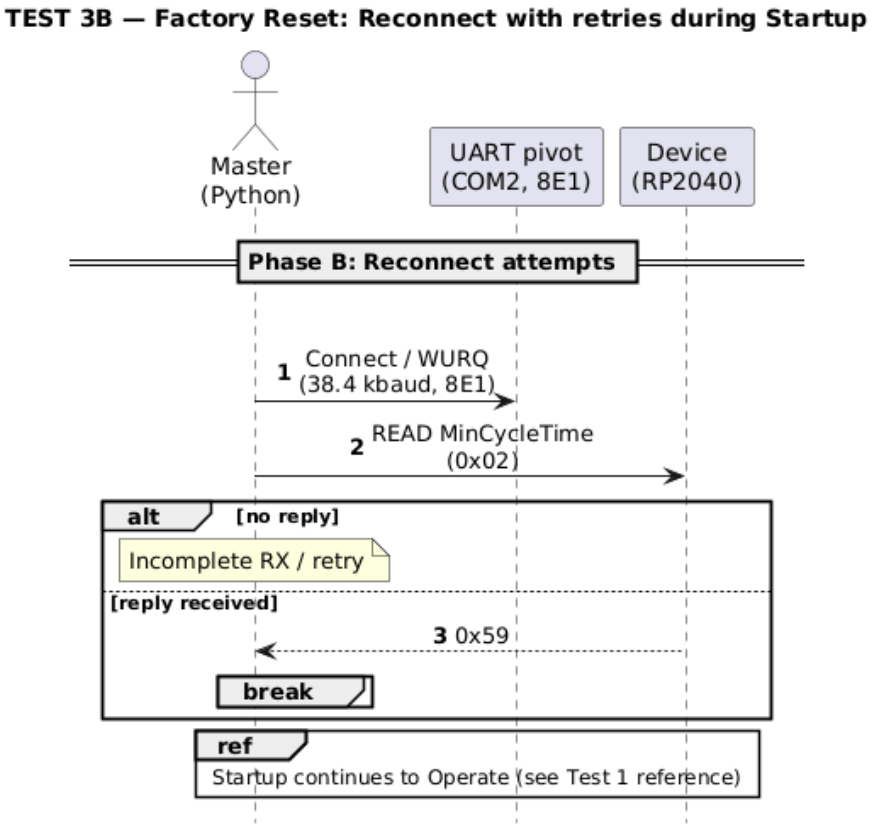


Figure 5.9: Test 3 (Phase B): Reconnection attempts during Startup after factory reset.

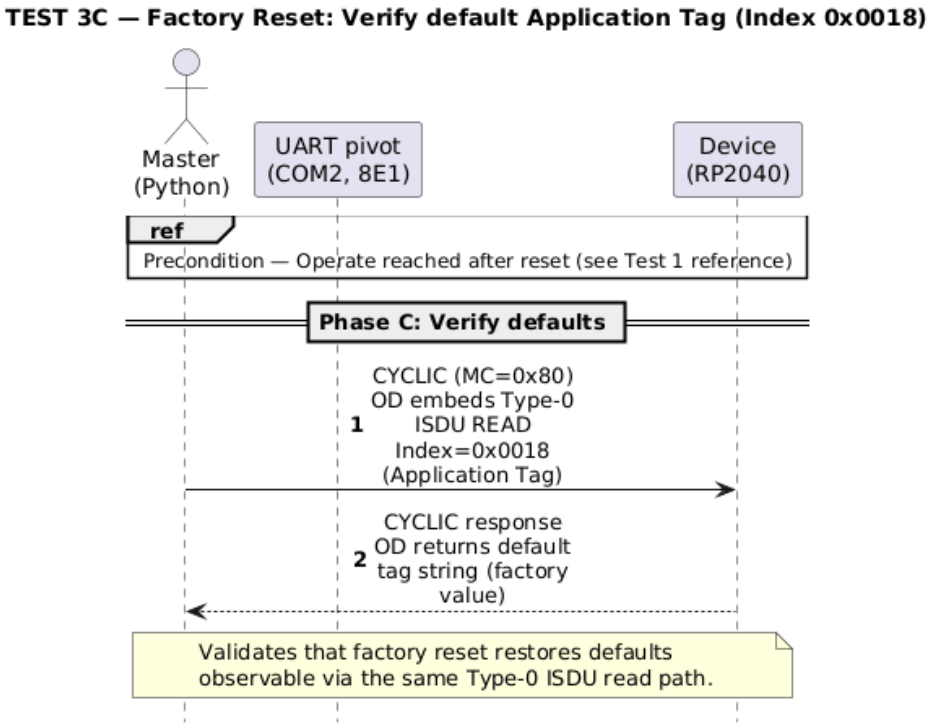


Figure 5.10: Test 3 (Phase C): Verification of default Application Tag after factory reset.

5.4.5 Test 4: End-to-End Subset Validation (Persistence + Error Handling)

Test 4 aggregates two representative checks into one execution script: (i) persistence (as in Test 2) and (ii) error handling for an invalid index access. The purpose is not to claim full standard coverage, but to validate that multiple implemented mechanisms behave correctly in a single end-to-end campaign with trace-aligned expectations, as illustrated in Figure 5.11.

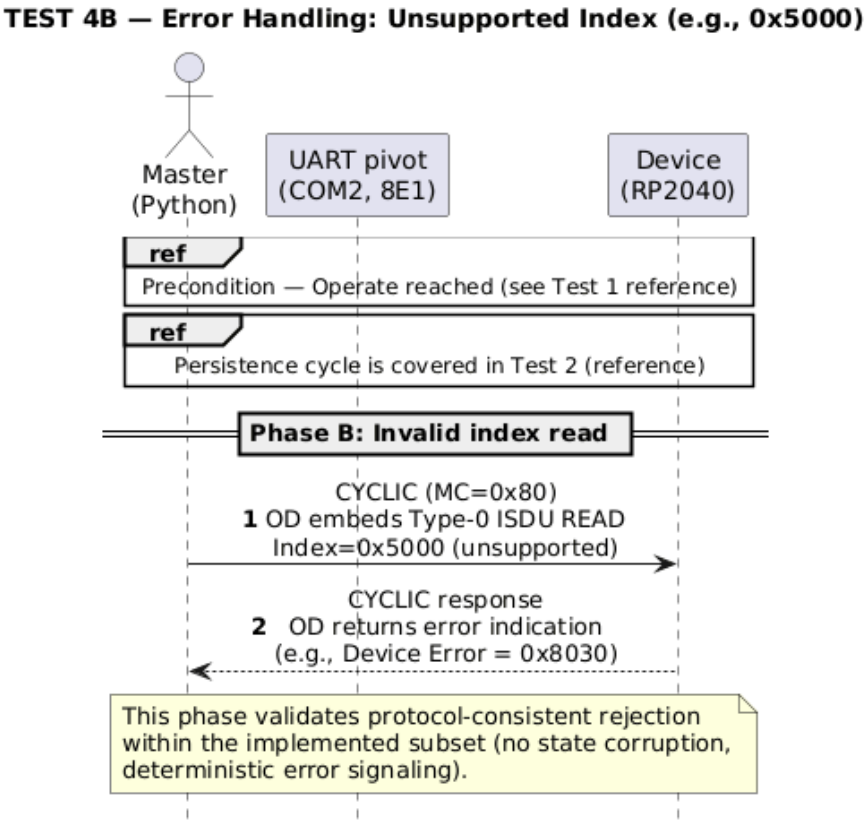


Figure 5.11: Test 4 (Phase B): Error handling validation via ISDU read of an unsupported index (e.g., 0x5000) and verification of the returned error indication within the implemented subset. Persistence cycle is treated as a reference (see Test 2, Section 5.4.2).

5.4.6 Test 5: Advanced Features (Page 2, CRC32, Truncation)

Test 5 exercises additional implemented features that are important for demonstrating bounded behavior and persistence integrity properties: Page 2 identification reads (0x10..0x1F), CRC32 check behavior via Index 0x001A, and truncation behavior for long strings (e.g., Vendor Name at 0x0010) under the OD-size bound described in Section 5.2.6. Where the log indicates an expected-string mismatch, it is treated as an explicit, traceable outcome of the test oracle expectation, not as a silent success. These advanced features are sequentially validated in Figures 5.12, 5.13, and 5.14.

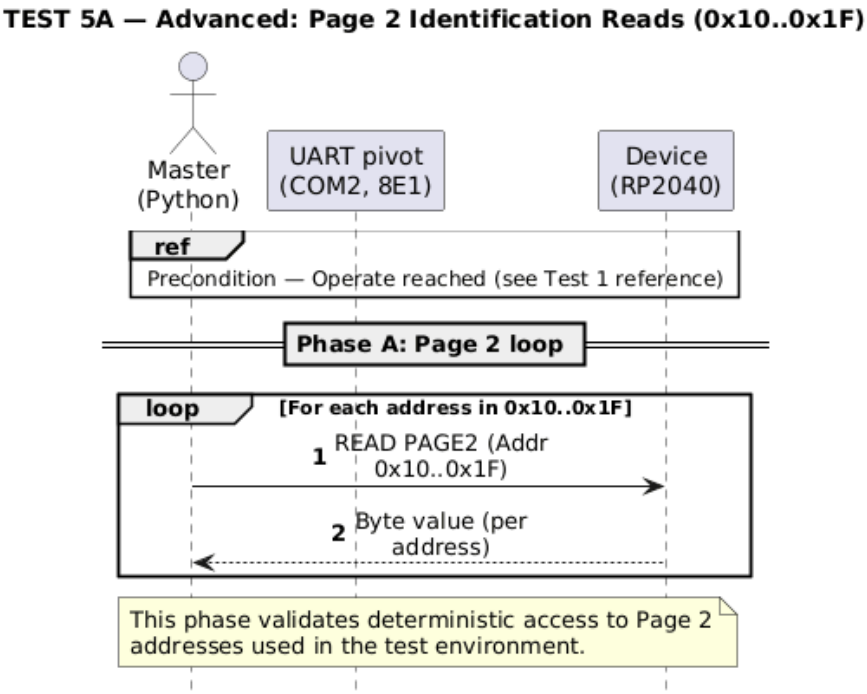


Figure 5.12: Test 5 (Phase A): Page 2 identification reads (addresses 0x10–0x1F).

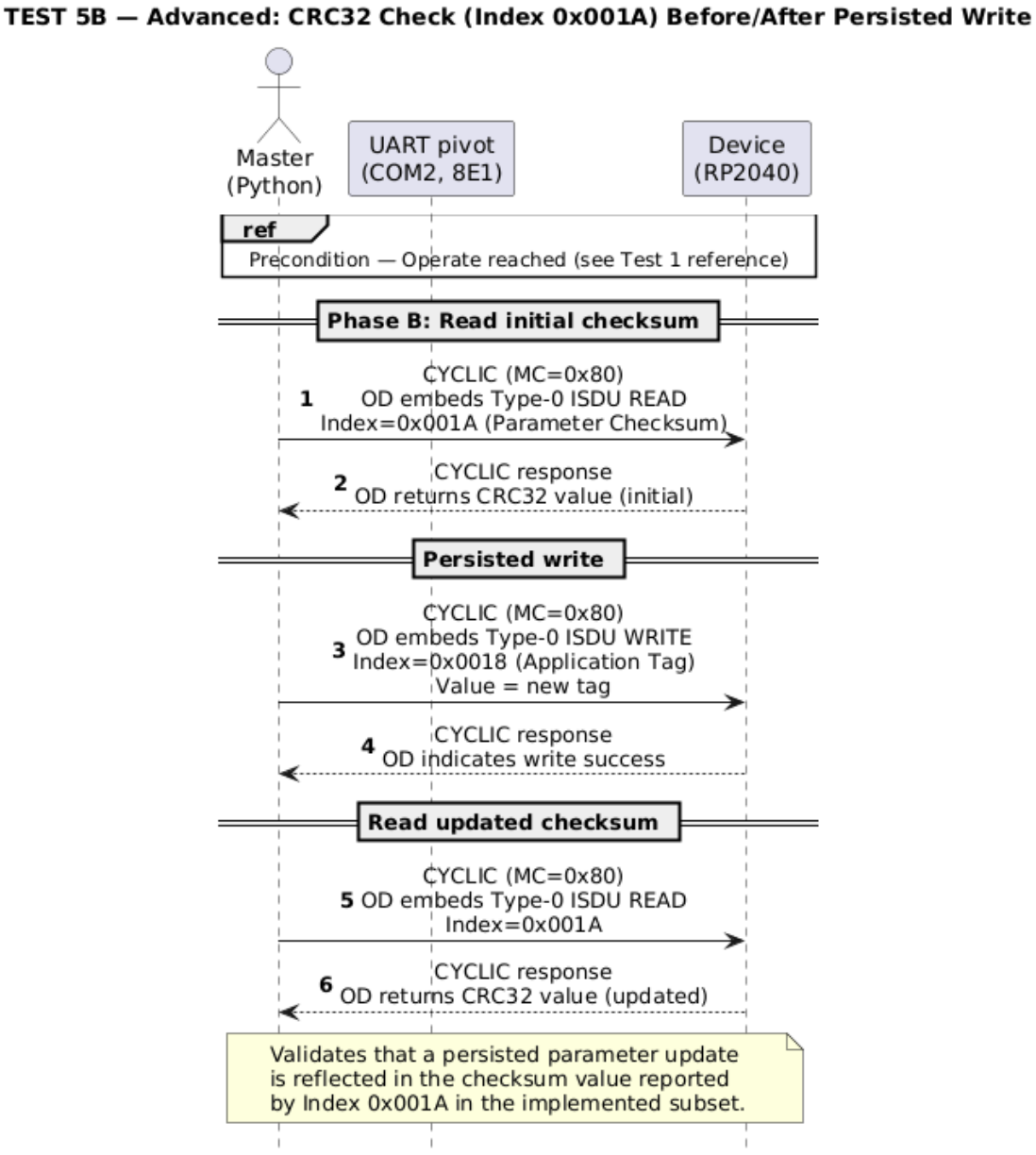


Figure 5.13: Test 5 (Phase B): CRC32 verification (Index 0x001A) before and after a persisted parameter update.

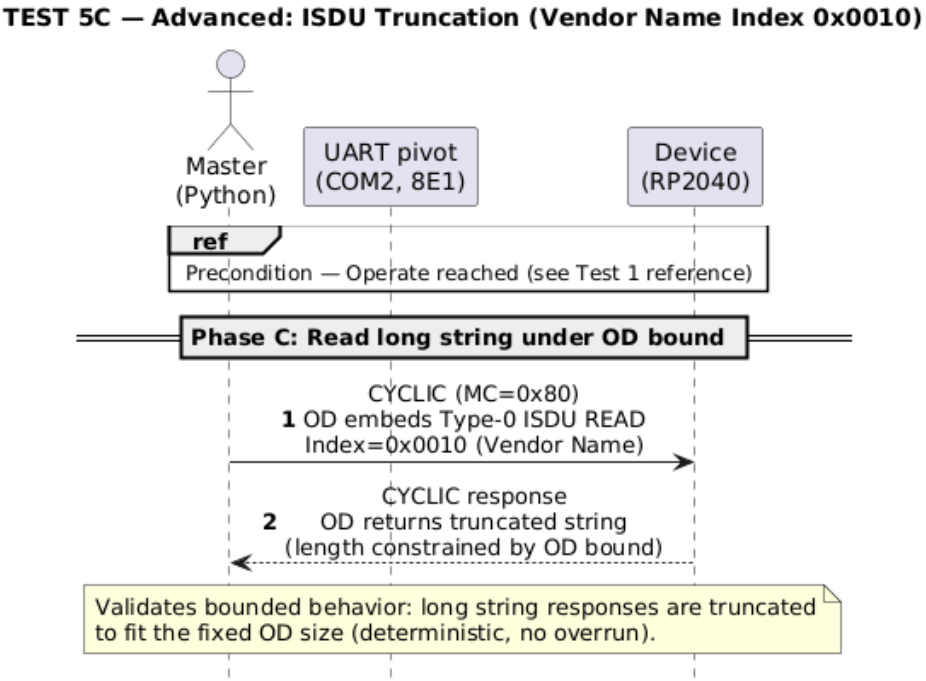


Figure 5.14: Test 5 (Phase C): ISDU truncation behavior for Vendor Name (Index 0x0010) under fixed OD size constraint.

5.4.7 Test 6: Compliant Event & *Data Storage* Flow (Diag + DS)

Test 6 validates (i) diagnostic event generation and retrieval through the *Diagnosis* readout sequence and (ii) the *Data Storage* trigger after writing a persisted parameter. The test uses the non-normative experimental index 0x9999 solely as a controlled fault-injection mechanism to force a deterministic event path without relying on uncontrolled external disturbances, as mapped in Figure 5.15.

5.4.8 Test 6: Fault Injection and Event Handling

Fault injection tests using invalid indices and malformed requests resulted in protocol-consistent error indications and diagnostic events within the implemented subset. When multiple events were pending, higher-priority events were transmitted first, confirming correct event queue ordering.

Table 5.3 summarizes the trace-based evidence for Test 6 and is referenced here to keep the narrative and the table explicitly linked.

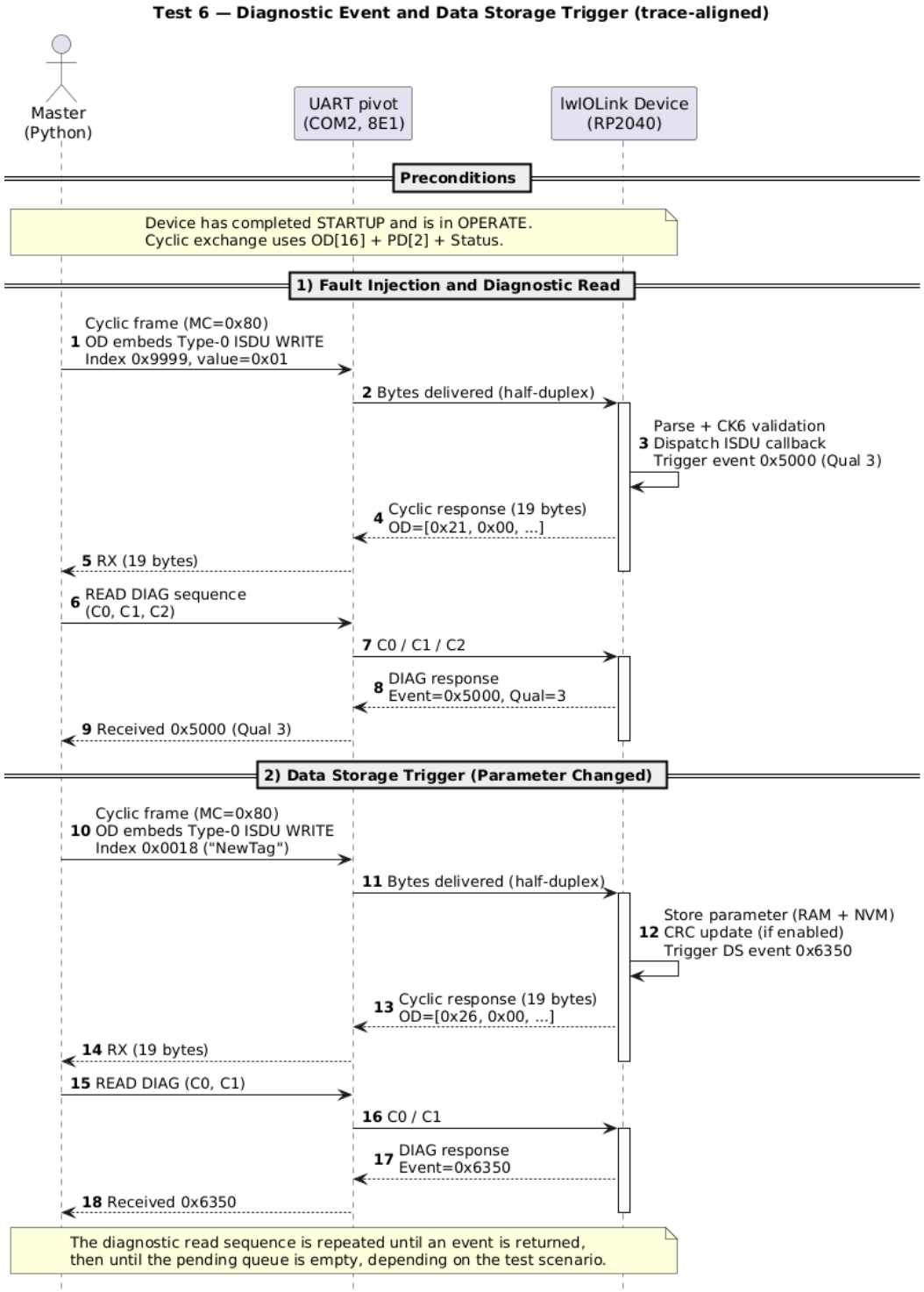


Figure 5.15: Event generation and *Data Storage* trigger validation flow (Test 6).

Table 5.3: Trace-based validation of diagnostic event handling and *Data Storage* trigger (Test 6).

Phase	**Master action**	**Observed master log (excerpt)**	**Device trace (excerpt)**	**Validated mechanism**
Fault injection (non-normative)	Write experimental Index 0x9999	`Including ISDU in OD (5/16): [A1 99 99 00 01]`	`[RX] MC=0x80 ... A1 99 99 00 01 ...`	Controlled, non-normative trigger used to exercise the diagnosis/event path.
Diagnosis readout	Read Diagnosis channel until event is returned	`READ DIAG (Addr 0x00); Addr 0x01; Addr 0x02 Received: 0x5000 (Qual 3)`	`[RX] MC=0xC0 → [TX] C0 .. 50 [RX] MC=0xC1 → [TX] C1 .. 00 [RX] MC=0xC2 → [TX] C2 .. 03`	Event delivery through Diagnosis channel, including a severity qualifier.
Data Storage trigger	Write Application Tag (Index 0x0018)	`Including ISDU in OD (10/16): [A6 00 18 00 4E 65 77 54 61 67]`	`[RX] MC=0x80 ... A6 00 18 00 4E 65 77 54 61 67 ...`	Write to persisted identification parameter triggers DS workflow within the implemented scope.
Parameter changed event	Read Diagnosis and receive event 0x6350	`READ DIAG (Addr 0x00); Addr 0x01 Received: 0x6350`	`[RX] MC=0xC0 → [TX] C0 .. 63 [RX] MC=0xC1 → [TX] C1 .. 50`	Automatic reporting of event `0x6350` (“Parameter changed”) after parameter write.

5.4.9 Test 6: ISDU Payload Bounding Behavior

Requests exceeding the configured ISDU payload limit resulted in truncated responses rather than device resets or undefined behavior. This demonstrates the expected behavior of the implemented safety mechanisms under the executed stress-test conditions.

5.5 Discussion

The validation campaign presented in this chapter demonstrates trace-level behavioral alignment between the implemented SpecIOLink-Device stack and a declared subset of IO-Link V1.1.4 data link layer requirements. However, it is essential to contextualize these results within the methodological and architectural boundaries explicitly defined throughout this work.

First, the use of a direct 3.3 V UART pivot intentionally decouples protocol validation from industrial physical-layer effects. This choice improves determinism, reproducibility, and observability of frame-level behavior. It enables byte-accurate trace comparison, controlled malformed-frame injection, and deterministic restart cycles. However, it simultaneously excludes validation of wake-up pulse timing, C/Q line electrical behavior, electromagnetic interference robustness, and transceiver-level COMLOST recovery under real industrial conditions. The results therefore

demonstrate logical protocol correctness under controlled transport abstraction rather than certified field robustness.

Second, the implemented subset deliberately excludes segmented ISDU transfers (Type-1/Type-2), dynamic COM mode negotiation, and COM3 high-speed operation. While the internal state machine structure was designed to be extensible, the experimental evidence presented here applies strictly to the bounded Type-0 ISDU model under static COM2 configuration. No claim is made regarding behavior outside this declared scope.

Third, the bounded-memory design and absence of dynamic allocation contribute significantly to deterministic execution. All buffers are statically sized, ISDU payloads are explicitly truncated to the configured On-Request Data (OD) limit, and CRC32-protected persistence ensures integrity verification at restart. The trace-based validation confirms that no overflow conditions or undefined state transitions were observed under malformed or oversized payload injection. Nevertheless, long-term endurance testing, flash wear characterization, and stress testing under high event rates were not part of the experimental campaign.

From a methodological perspective, the custom Python-based master and trace comparator architecture proved instrumental for protocol inspection. The ability to freeze execution, correlate master logs with device traces, and deterministically reproduce fault scenarios establishes a high degree of transparency compared to black-box industrial validation approaches. This transparency constitutes a primary research contribution of the work.

At the same time, it must be emphasized that trace-level behavioral agreement does not substitute for formal IO-Link Consortium conformance testing. Certified compliance requires standardized test suites, electrical conformance validation, and interoperability testing with multiple commercial master implementations. The present work instead positions itself as a specification-oriented, experimentally validated reference implementation within a declared and bounded subset.

In summary, the validation results provide strong evidence of internal consistency, deterministic state-machine behavior, bounded ISDU handling, and correct integration of persistence and event workflows within the defined scope.The results indicate that the implemented mechanisms provide a technically consistent basis for future reintegration of an industrial physical layer and for broader interoperability testing. However, the conclusions remain explicitly limited to the validated subset and controlled experimental conditions.

5.6 Summary

This chapter presented the implementation details and validation results of the adopted and extended IO-Link V1.1.4 device stack. The `lwIOLink` library was adapted and validated using a custom Python-based IO-Link Master.

The conducted experiments confirm correct operational mode handling, robust frame validation, bounded ISDU behavior, reliable parameter persistence, specification-oriented event signaling, and the presence of optional wake-up and COMLOST mechanisms that were intentionally

not exercised for the selected validation scenario, providing a solid basis for the conclusions and future work discussed in the following chapter.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This dissertation presented the systematic extension, restructuring, and experimental validation of a specification-oriented IO-Link V1.1.4 device stack targeting embedded retrofit experimentation.

Rather than developing a device stack from first principles, an existing lightweight open-source implementation (lwIOLink) was adopted as a baseline and subjected to architectural restructuring. The resulting system incorporates explicit operational state modeling, bounded memory guarantees, structured ISDU dispatching, diagnostic event prioritization, and CRC32-protected non-volatile parameter persistence.

The validated implementation supports a clearly delimited subset of IO-Link V1.1.4 data link layer behavior, including:

- COM2 communication mode;
- Deterministic M-sequence parsing and CK6 validation;
- Explicit Startup, Preoperate, and Operate state handling;
- State-dependent service gating;
- Type-0 ISDU transactions with bounded payload;
- Diagnostic event signaling via the Diagnosis channel;
- Controlled parameter persistence with integrity verification.

Segmented ISDU transfers (Type-1 and Type-2), COM3 operation, and automatic COMLOST recovery actions were intentionally excluded from the validated scope in order to preserve bounded resource guarantees and maintain architectural clarity.

A central methodological contribution of this work is the staged validation strategy. By abstracting the industrial 24 V physical layer and employing a controlled 3.3 V UART pivot operating under COM2 framing (38.4 kbaud, 8E1, half-duplex), protocol-level behavior was validated independently of electrical-layer variability. This separation enabled deterministic correlation between

master-generated M-sequences and device-side trace outputs, systematic malformed-frame injection, and controlled verification of state transitions, event workflows, and persistence mechanisms.

Experimental validation demonstrated stable cyclic and acyclic communication within the evaluated environment. Correct operational mode transitions, bounded ISDU handling, diagnostic event prioritization, and deterministic parameter restoration after power cycling were observed through trace-based comparison between expected and captured frames.

To ensure the rigor of these findings, a structured traceability analysis was maintained throughout the dissertation. This approach explicitly linked the initial retrofit motivations (Chapter 1) and normative foundations (Chapter 2) to the identified gaps in the state-of-the-art (Chapter 3). Consequently, the architectural design decisions (Chapter 4) directly informed the implemented firmware mechanisms and the subsequent validation campaigns (Chapter 5). Each claimed behavior within the defined scope is supported by explicit validation evidence, including master-side logs, device-side traces, and controlled fault-injection scenarios, establishing a defensible alignment between requirements, implementation, and experimental results.

It is essential to emphasize that no formal IEC 61131-9 certification process or IO-Link Consortium conformance testing was performed. Compliance claims are strictly limited to experimentally validated behavioral alignment within the explicitly implemented subset of IO-Link V1.1.4 data link layer functionality.

Within these clearly defined scope boundaries, the extended lwIOLink stack constitutes a transparent, bounded, and experimentally verifiable IO-Link device implementation suitable for research, education, and controlled protocol experimentation.

6.2 Limitations

The results presented in this work must be interpreted within clearly defined scope boundaries.

First, validation was performed using a transport-level abstraction based on a 3.3 V UART interface rather than a fully compliant 24 V IO-Link physical layer. Although methodologically justified for protocol isolation, this configuration does not capture transceiver-specific electrical effects, electromagnetic interference, or industrial cabling constraints.

Second, only COM2 communication mode was exercised experimentally. COM1 and COM3 modes were not validated. Segmented ISDU transfers were not implemented. Wake-up signaling and COMLOST monitoring mechanisms exist structurally in the firmware but were intentionally disabled during validation to avoid non-deterministic state transitions.

Third, no interoperability testing against certified commercial IO-Link masters was performed. Validation relied on a custom Python-based master designed for deterministic trace analysis and fault injection.

Finally, application-layer validation was performed using representative data abstractions rather than real industrial sensing hardware. Long-term reliability, environmental robustness, and certification-level testing were beyond the scope of this dissertation.

6.3 Future Work

Future work may extend the validated scope and address identified limitations.

Immediate directions include:

- Reintegration of an industrial IO-Link transceiver and validation under full 24 V physical-layer conditions;
- Activation and validation of COMLOST recovery logic under intermittent communication scenarios;
- Extension to additional communication modes, particularly COM3;
- Implementation of segmented ISDU transfers with formally bounded memory guarantees;
- Interoperability testing with certified commercial IO-Link masters;
- Formal conformance evaluation aligned with IO-Link Consortium procedures.

These extensions would increase industrial applicability while preserving the architectural transparency and experimental verifiability established in this work.

References

[1] Balluff GmbH. "Analog signals: 0–10 v vs. 4–20 ma," Accessed: Sep. 26, 2025. [Online]. Available: https://www.balluff.com/en-us/blog/analog-signals-0-to-10-v-vs-4-to-20-ma.

[2] DigiKey. "What is the 4–20 ma communication protocol and how does it work," Accessed: Sep. 26, 2025. [Online]. Available: https://www.digikey.com/en/maker/blogs/2023/what-is-the-4-20ma-communication-protocol-and-how-does-it-work.

[3] B. Galloway and G. P. Hancke, "Introduction to industrial control networks," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 2, pp. 860–880, 2013. DOI: 10.1109/SURV.2012.071812.00124.

[4] IO-Link Consortium, "Io-link interface and system specification," IO-Link Consortium, Version 1.1.4, 2024. Accessed: Sep. 26, 2025. [Online]. Available: https://io-link.com/fileadmin/user_upload/Downloads/Package_2024/IOL-Interface-Spec_10002_V114_Jun24.pdf.

[5] International Electrotechnical Commission, "Iec 61131-9:2022 — programmable controllers — part 9: Single-drop digital communication interface for small sensors and actuators (sdci)," International Electrotechnical Commission, 2022. Accessed: Sep. 26, 2025. [Online]. Available: https://webstore.iec.ch/publication/68534.

[6] *Io-link handbook*, Analog Devices, 2019. Accessed: Sep. 26, 2025. [Online]. Available: https://www.analog.com/media/en/news-marketing-collateral/solutions-bulletins-brochures/io-link-handbook.pdf.

[7] H. Kagermann, W. Wahlster, and J. Helbig, "Recommendations for implementing the strategic initiative industrie 4.0," acatech – National Academy of Science and Engineering, 2013. Accessed: Sep. 26, 2025. [Online]. Available: https://en.acatech.de/publication/recommendations-for-implementing-the-strategic-initiative-industrie-4-0-final-report-of-the-industrie-4-0-working-group/.

[8] A. Alqoud, D. Schaefer, and J. Milisavljevic-Syed, "Industry 4.0: A systematic review of legacy manufacturing system digital retrofitting," *Manufacturing Review*, vol. 9, p. 32, 2022. DOI: 10.1051/mfreview/2022031. Accessed: Sep. 26, 2025. [Online]. Available: https://mfr.edp-open.org/articles/mfreview/full_html/2022/01/mfreview220051/mfreview220051.html.

[9] D. Atzeni, R. Ramjattan, R. Figliè, G. Baldi, and D. Mazzei, "Data-driven insights through industrial retrofitting: An anonymized dataset with machine learning use cases," *Sensors*, vol. 23, no. 13, p. 6078, 2023. DOI: 10.3390/s23136078. Accessed: Sep. 26, 2025. [Online]. Available: https://www.mdpi.com/1424-8220/23/13/6078.

[10] ifm electronic. “Analog signals to io-link,” Accessed: Sep. 26, 2025. [Online]. Available: https://www.ifm.com/us/en/us/learn-more/io-link/io-link-signal-conditioners/analog-signals-to-io-link.

[11] Balluff GmbH. “Io-link converter for analog input and output signals,” Accessed: Feb. 12, 2026. [Online]. Available: https://www.balluff.com/en-de/news/io-link-converter-for-analog-in-and-output-signals.

[12] Pepperl+Fuchs. “Io-link analog converter ica-ai-i/u-io-v1 and ica-ao-i/u-io-v1,” Accessed: Feb. 12, 2026. [Online]. Available: https://www.pepperl-fuchs.com/en/products-gp25581/135540.

[13] Baumer Group. “Io-link converter cc50,” Accessed: Feb. 12, 2026. [Online]. Available: https://www.baumer.com/int/en/product-overview/io-link-and-connectivity-technology/io-link-converter/io-link-converter-cc50.

[14] Turck GmbH & Co. KG. “Ilc-aiu io-link analog converter,” Accessed: Feb. 12, 2026. [Online]. Available: https://www.turck.de/en/product/100036698.

[15] Murrelektronik GmbH, *Io-link analog converter brochure*, 2016. Accessed: Feb. 12, 2026. [Online]. Available: https://www.murrelektronik.com/fileadmin/user_upload/Headquarter_en_DE/Downloads/Brochures/C_IO-Link-Analog-Converter-12-16_EN.pdf.

[16] . Chavez, *Rapid prototyping framework for automation applications based on io-link*, 2021. Accessed: Feb. 12, 2026. [Online]. Available: https://d-nb.info/1250829879/34.

[17] A. Author and B. Author, “Implementation of an io-link master proof-of-concept,” in *Proceedings of the International Conference on Industrial Engineering*, 2026.

[18] J. Smith and R. Doe, “Io-link master bridge: Practical implementation for simple data acquisition,” in *IEEE International Symposium on Industrial Electronics (ISIE)*, 2022.

[19] M. Kowalski and P. Nowak, “Study and application of the io-link industrial network in the context of industry 4.0,” in *International Conference "Mixed Design of Integrated Circuits and Systems" (MIXDES)*, 2023.

[20] S. Ivanov and V. Petrov, “Capabilities of the io-link master device integration through the web server,” in *IEEE International Conference on Emerging Technologies and Factory Automation*, 2021.

[21] Y. Chen and H. Wang, “Coexistence standardization of operation technology and information technology,” in *IEEE International Conference on Industrial Technology (ICIT)*, 2019.

[22] T. Muller and L. Schmidt, “An industrial feedback on the use of opc ua for the vertical integration of semiconductor front-ends,” in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021.

[23] G. Rossi, “Creation of a complete data collection system for condition monitoring with io-link sensor,” M.S. thesis, Politecnico di Torino, 2022.

[24] S. Heynicke et al., “Io-link wireless enhanced factory automation communication for industry 4.0 applications,” *Journal of Sensors and Sensor Systems (JSSS)*, 2018.

[25] A. Silva and B. Costa, “Io-link wireless sensitivity testing methods in reverberation chambers,” *Sensors*, vol. 22, 2022.

[26] M. Weber and J. Fischer, “Opc ua for io-link wireless in a cyber physical finite element sensor network,” in *Sensor and Measurement Science International (SMSI)*, 2025.

[27] unref-ptr, *lwIOLink: Lightweight IO-Link Device Stack*, https://github.com/unref-ptr/lwIOLink, Accessed: 2026-02-26, 2023.