



RasProLink

User & Commissioning Manual

One Windows service, five industrial protocol engines

Turn any Windows PC into a real EtherCAT, EtherNet/IP or PROFINET station — no extra hardware.

RAS Automation — RasProLink v1.0

September 2026

Table of Contents

Chapter 1	Introduction	3
Chapter 2	System Requirements	5
Chapter 3	Installation	6
Chapter 4	The Web Interface	7
Chapter 5	Data Mapping and Tags	10
Chapter 6	EtherCAT Master	13
Chapter 7	EtherNet/IP Scanner	17
Chapter 8	EtherNet/IP Adapter	20
Chapter 9	PROFINET Controller	22
Chapter 10	PROFINET Device	33
Chapter 11	.NET Client Library	36
Chapter 12	Licensing	41
Chapter 13	Performance and Real-Time Behaviour	43
Chapter 14	General Troubleshooting	44
Appendix A	HTTP API Reference	45
Appendix B	Configuration Files	50
Appendix C	Glossary	52

Chapter 1

Introduction

1.1 Who this manual is for

This manual is for automation engineers who need a PC to speak a real industrial fieldbus — as a Master, a Controller, a Scanner, an Adapter, or a Device — and for developers who consume that data from a .NET application, a script, or any HTTP client.

Familiarity with your chosen protocol's basic concepts (station, slave/device, cyclic data) is assumed; no prior experience with RAS Automation products is required.

1.2 What RasProLink is

RasProLink is a single Windows service that turns your PC into a real industrial communication station on five protocols: EtherCAT (as Master), EtherNet/IP (as Scanner/Originator or as Adapter/Target), and PROFINET (as IO-Controller or as IO-Device). You pick the active protocol engine from a web interface — no other hardware, gateway box, or paid third-party stack is required.

Engine	Protocol	Role
EtherCAT Master	EtherCAT	Master (your PC drives the slave ring)
EtherNet/IP Scanner	EtherNet/IP (CIP)	Originator — opens Class 1 connections to Adapters
EtherNet/IP Adapter	EtherNet/IP (CIP)	Target — accepts Forward Open from a Scanner/PLC
PROFINET Controller	PROFINET	IO-Controller — the Master station a PLC normally is
PROFINET Device	PROFINET	IO-Device — a Slave to a real PLC, described by GSDML

Only one engine runs at a time; switching engines from the web interface restarts the underlying service in the new protocol within seconds. Each engine keeps its own configuration, so switching back restores where you left off.

1.3 Why RasProLink

- No hardware fieldbus card, dongle, or gateway box — a standard Ethernet port is enough.
- No expensive third-party connectivity libraries — the protocol stack is built in and driven directly.
- Multi-protocol by design — one installation, five engines, switch whenever a project needs a different fieldbus.
- Drag-and-drop, real-time commissioning in a plain Windows environment — real Master/Controller/Device behaviour, not a simulator.
- .NET-native — any software you already run on .NET becomes the brain behind the Controller/Device, fully under your own command.

Tip: Real-time note: on all five engines, the cyclic exchange loop runs on a dedicated high-priority thread with Windows multimedia-timer resolution and CPU core-affinity pinning, with per-cycle heap allocations eliminated. On a typical Windows PC this keeps cycle jitter in the sub-millisecond range (values around 0.3 ms have been measured at a 4 ms EtherCAT cycle) — see §13.4.

1.4 Ways to use it

Once devices are connected, there are two ways to use the data:

- Web interface — add devices, watch status, read/write raw IO bytes, manage the license and follow the service log. Meant for commissioning and diagnostics, not for driving production logic.
- .NET client library (RasProLink.Client) — a small, dependency-free DLL your own application references to read device status and IO in code, write outputs, and manage devices/tags programmatically. This is what a real application is built on — see Chapter 11.

Chapter 2

System Requirements

- Windows 10 / 11, or Windows Server 2016 or later, 64-bit.
- A free, wired Ethernet adapter on the same physical/logical network as your field devices. Wi-Fi is not suitable for the field-facing side of any of the five engines — real-time cyclic traffic needs a switched, wired segment.
- EtherCAT Master only: a network adapter with a chipset supported by the built-in raw-frame driver (most Intel and Realtek adapters are supported).
- PROFINET Controller and PROFINET Device: Npcap (<https://npcap.com>) for raw frame access — bundled with the installer, or installed beforehand.
- No .NET runtime needs to be installed separately — the service is published fully self-contained.
- Administrator rights to install the Windows service and Npcap/raw-frame driver.

Warning: Do not put the field-facing side of any engine behind a Wi-Fi link, a VPN, or a heavily loaded shared switch: EtherCAT and PROFINET cyclic frames and EtherNet/IP CIP connections are timing-sensitive, and added latency or jitter shows up directly as connection loss or watchdog timeouts.

Chapter 3

Installation

3.1 Installation steps

1. Copy the package to the target PC.
2. If an npcap-*.exe installer is included, run it first (or install Npcap from npcap.com beforehand). Not required for the EtherNet/IP engines.
3. Right-click install.bat and choose Run as administrator.
4. The service installs, starts automatically, and is set to start with Windows.
5. Open <http://localhost:8080> in a browser on the PC to confirm the web interface responds.

3.2 How the service runs

RasProLink runs as a Windows service with no window or tray icon, under the name RasProLink.Service. It starts before anyone logs on and keeps running across user sessions. Use the Windows Services console (services.msc) to check status, or to stop/start it manually while troubleshooting.

3.3 Remote access to the web interface

By default the web interface listens on all local addresses on port 8080, so <http://<pc-ip>:8080> reaches it from another machine on the network — handy when the industrial PC is mounted elsewhere and you commission it from a laptop. If Windows Firewall is on, allow the inbound TCP 8080 traffic the first time it is reached from outside.

3.4 Update and removal

To update, run the new version's install.bat over the existing installation: it stops the running service, replaces the executable and files, and restarts it. Configuration files are not touched by an update, so device lists and network settings survive. To remove, run uninstall.bat; the configuration folder is kept by default (in case you reinstall later) — pass -RemoveConfig to also delete it and the logs.

Chapter 4

The Web Interface

The web interface layout is identical across all five engines: only the active engine's name and its protocol-specific cards change. The screenshots below are taken from the PROFINET Controller engine as a representative example; EtherCAT Master, EtherNet/IP Scanner/Adapter and PROFINET Device use the same frame with their own device/configuration cards (covered in their own chapters, §6–§10).

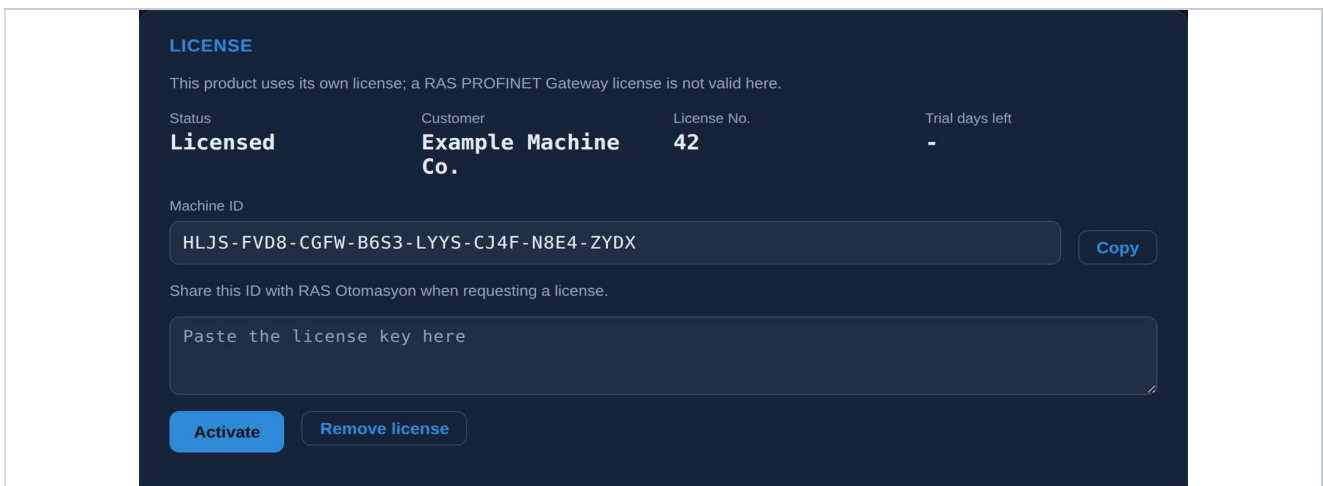
4.1 Top bar and language selector



Top bar — active engine name, language selector, live connection indicator (example: PROFINET Controller)

The top bar shows the product name and the active engine, a language selector, and a connection-status pill that turns green once the browser can reach the service. The whole interface re-renders instantly on a language change; the page does not reload.

4.2 License card



License card during the trial period

License card after a key has been activated

The card shows status, customer, license number and remaining trial days; the Machine ID to share with RAS Automation when requesting a license; and a field to paste a license key into. Full flow in Chapter 12.

4.3 Network adapter selection

Network adapter selection, with an optional manual MAC override

On a PC with more than one network adapter, choose the one physically wired to the field devices here. This selector is shared across every engine, since every engine talks to the field on exactly one adapter at a time.

4.4 Switching the active engine

Every engine's top bar (§4.1) carries a permanent link back to a shared protocol picker — for example Protocol: PROFINET Controller ↗ on the PROFINET Controller pages, Protocol: EtherCAT Master ↗ on the EtherCAT Master pages, and so on. Whichever engine is currently active, this link always opens the same page, served at /engine.html, and is the only way to change which of the five engines the service runs.

The picker

The picker shows one card per engine — EtherCAT Master, EtherNet/IP Scanner, EtherNet/IP Adapter, PROFINET Controller, PROFINET Device. The currently active engine's card carries an AKTİF (active) badge and a disabled **Şu an aktif** (currently active) button; every other card shows either a Kullanıma hazır (ready for use) badge or, for an engine not yet finished, a Geliştirme aşamasında (in development) badge with a short warning line — and a Bu protokolü seç (select this protocol) button. As of this writing, all five engines are production-ready, so in

practice every non-active card currently shows the ready badge; the development badge exists in the interface for whichever engine ships next in that state.

Note: Unlike the rest of the interface (§4.1), this page is not affected by the language selector — its labels stay in Turkish no matter which language is active on the engine pages you came from. The labels quoted above (AKTİF, Kullanıma hazır, Bu protokolü seç, and so on) are the exact text you will see, in every language edition of RasProLink.

Making the switch

Clicking a card's button asks for confirmation — a stronger warning if the target engine is not yet production-ready — then posts the chosen engine's key (for example profinet-controller or ethernetip-scanner) to POST /api/engine. The service writes that choice to Config\enginesettings.json and, about three seconds later, exits on purpose. Installed as a Windows Service (Chapter 3), RasProLink is configured to restart automatically whenever it exits, including this clean exit, so the Service Control Manager brings the process back up within a few seconds; on that fresh start it reads Config\enginesettings.json again and comes up running the newly selected engine.

While the service is down, the picker page polls GET /api/host/info every three seconds, for up to about two minutes, until the response's activeEngine matches the engine just selected — then it redirects the browser to /index.html, now serving the new engine's own interface. If two minutes pass without a match, the page stops polling and asks you to refresh manually instead of retrying forever.

Note: A switch is a real process restart, not a live hand-off: any other browser tab or client still pointed at the previous engine's pages or API loses its connection the instant the service exits, and cyclic data stops until the new engine finishes starting. Let the picker's automatic redirect (or the connection pill of §4.1) confirm the new engine is up rather than continuing to use an old tab. When RasProLink is run interactively (dotnet run) instead of as an installed service, the automatic restart described above does not happen — the process simply exits, and it must be started again by hand.

Configuration is kept per engine

Each engine keeps its own settings in its own file(s) under Config\, following the same convention used throughout this manual (§9.7 and equivalent sections in Chapters 6–10). Switching engines only ever writes Config\enginesettings.json — the service builds the configuration store for whichever engine becomes active and does not open or touch any other engine's files. Confirmed from the service's startup code: switching away from an engine and back later leaves its device list, targets, tags and other settings exactly as they were left.

Engine	Own configuration file(s) under Config\
PROFINET Controller	controllersettings.json, devices.json, tags.json*
PROFINET Device	deviceconfig.json, tags.json*
EtherNet/IP Scanner	ethernetip-scanner-targets.json, ethernetip-scanner-tags.json
EtherNet/IP Adapter	ethernetip-adapter-config.json, ethernetip-adapter-tags.json
EtherCAT Master	ethercat-master-config.json, ethercat-master-tags.json

Note: * One exception, confirmed in the current build: PROFINET Controller and PROFINET Device both read and write the same Config\tags.json file for their tag lists. Unlike their other settings, which are fully separate,

switching between just these two engines does not give each of them an independent tag list. The license card (§4.2) is unaffected by any of this either way — the license itself lives in a single Config\lisans.json that is shared by all five engines, so switching engines never asks you to re-activate.

Chapter 5

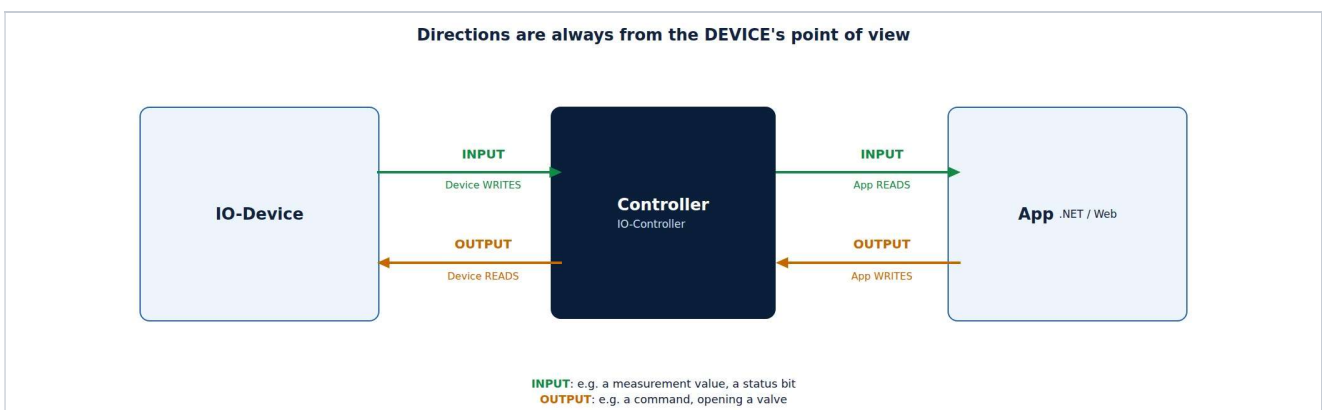
Data Mapping and Tags

5.1 Slots and submodules

Every engine organizes a connected device's process data the same way: a device is a tree of slots, each slot holding one or more submodules, and each submodule carrying its own input and/or output byte length. This applies whether the underlying concept is a PROFINET slot/subslot, an EtherCAT PDO-mapped entry, or an EtherNet/IP assembly — RasProLink normalizes all three onto the same slot/subslot model everywhere in the web interface, the .NET client, and the HTTP API.

5.2 Input/output direction

Direction is always named from the connected device's point of view: Input is data the field device produces and your application reads (a sensor reading, a status word); Output is data your application produces and the field device consumes (a setpoint, a command, a coil). This is the reverse of how a PLC program names its own I/O, and it is a one-time rule, not a per-engine one.



Direction is always named from the DEVICE's point of view — the reverse of a PLC program's own rule, worth checking once

An output submodule's bytes are written by your application (or the field-side PLC/Scanner, depending on the engine) and consumed by the connected side on the next cycle.

5.3 Tags

Tags

In addition to raw data (Live IO above), give a byte/bit location a NAME and TYPE — reading/writing can then be done by that name. Tags can also be defined for devices added via XDB.

NAME	SLOT	DIRECTION	TYPE	LOCATION	VALUE	
Sensor_Input1	2 / 0x0001	Input	bool	@0.0	false	<button>Edit</button> <button>Delete</button>
Motor_Run	3 / 0x0001	Output	bool	@0.0	true	<button>Write</button> <button>Edit</button> <button>Delete</button>

Tag name

1/0x0001 — Interface (in 0B / out 0B) ▼

Input (from device) ▼

Int16 ▼

Offset

Bit

Length

Unit

Add / Update

Tags card — name a byte/bit range so it can be read or written by name (example shown for PROFINET Controller)

A tag is a layer on top of the raw cyclic data — it never copies it. Give a byte/bit position within a slot/subslot a name and a type, and from then on you (or your code) can read/write it by that name instead of raw hex. Tags can be defined on any device on any engine, including ones added automatically (via .xdb, GSDML, or ESI matching) — the module structure only supplies the slot/subslot layout; the naming is always yours to add on top.

Type	Size	Notes
Bool	1 bit	Bit number 0–7 within the byte; other bits are unaffected.
Byte / SByte	1 byte	Unsigned / signed 8-bit.
Int16 / UInt16	2 bytes	Big-endian, as all three fieldbuses use on the wire.
Int32 / UInt32	4 bytes	Signed/unsigned 32-bit.
Int64 / UInt64	8 bytes	Read the raw bytes on the .NET client for the exact value (double loses precision above 53 bits).
Float32 / Float64	4 / 8 bytes	IEEE 754 (Real / LReal).
String	variable	Fixed-length ASCII, padded/accepted with trailing space or NUL.
Bytes	variable	Raw byte array, read/written as hex text.

Validation rules, checked on every add/update (the whole change is rejected if any rule fails): the name must be non-empty and unique within the device; a Bool tag's bit number must be 0–7; a String/Bytes tag's length must be at least 1; two tags may not claim the same bytes in the same slot/subslot and direction (two Bool tags on different bits of the same byte are the one exception — a common way to name each bit of a status word individually). A tag pointing at a slot/subslot that no longer exists, or reaching past its submodule's data, is flagged with a warning rather than blocked outright — harmless until the device is re-imported with a different module layout.

Chapter 6

EtherCAT Master

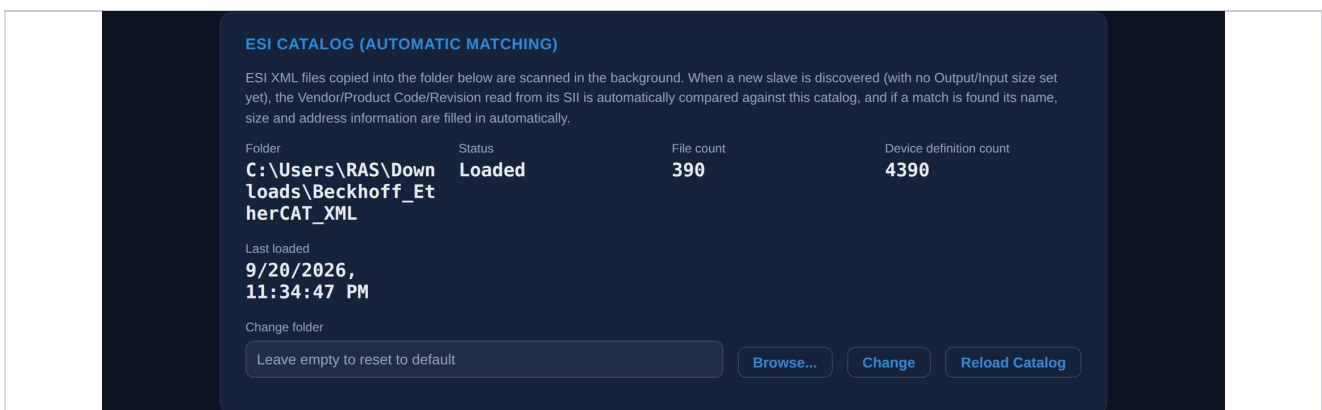
6.1 Overview

The EtherCAT Master engine turns your PC's Ethernet adapter into the Master of an EtherCAT segment: it opens the ring, walks it during the INIT→PREOP→SAFEOP→OP state machine, and then drives the cyclic process-data exchange with every slave on the ring at the configured cycle time, entirely in software — no dedicated EtherCAT master card is used.

Item	Value
Role	Master
Default cycle time	4 ms (configurable)
Slave identification	ESI XML catalog, auto-matched by Vendor/Product/Revision from SII
Measured jitter (typical PC, 4 ms cycle)	≈ 0.3 ms after real-time tuning

6.2 ESI catalog — automatic slave matching

EtherCAT slaves do not need a proxy file the way PROFINET devices do: every slave publishes its Vendor ID, Product Code and Revision itself over the SII (Slave Information Interface) as soon as it is powered and wired into the ring. RasProLink reads that identity automatically and looks it up in a folder of ESI XML files — the same files your slave vendors already publish for TwinCAT, Codesys or similar tools.



ESI catalog card — folder watched for ESI files, device signatures indexed, last reload time

Copy your slave vendors' ESI XML files into the watched folder (a default location is provided, and it can be changed from this card); new files are picked up automatically. When a newly discovered slave has no output/input size configured yet, RasProLink compares its SII-reported signature against every indexed ESI file and, on a match, fills in its name, PDO sizes and process-data offsets automatically — no manual byte-counting.

6.3 Process-data mapping

Each slave's PDO (Process Data Object) map — its input and output bit/byte layout — comes from the matched ESI entry, exactly like the Slot/Subslot table for a PROFINET device (§5 principles apply equally here: direction is always named from the slave's point of view, tags let you name a byte/bit range, all supported types from Bool to Bytes are available). See §5 for the shared data-mapping and tag rules that all five engines use.

6.4 .NET client example

```
using RasProLink.Client;

using var client = new RasProLinkClient("localhost", 8080);
var status = await client.ConnectAsync();

var slaves = await client.GetDevicesAsync();
foreach (var s in slaves)
    Console.WriteLine($"{s.Name} @ {s.State} in={s.InputBytes}B out={s.OutputBytes}B");

// Read one slave's live process data
var io = await client.GetDeviceIoAsync("drive-axis-1");

// Write an output byte range (e.g. a drive control word)
await client.WriteDeviceOutputAsync("drive-axis-1", slot: 1, subslot: 1,
    data: new byte[] { 0x06, 0x00 });
```

This is the same RasProLink.Client library used by every engine — only the target device name changes. See Chapter 11 for the full API surface.

6.5 Troubleshooting

Symptom	Likely cause
No slaves found	Ring not closed (last slave's OUT port must loop back, or an opening topology used consistently), or the selected adapter is not the one wired to the ring.
Slave stuck below OP	No ESI match (unknown PDO sizes) — add the vendor's ESI file to the catalog folder and let it re-scan, or a slave-specific SDO/mailbox startup parameter failed (test the object by hand with the SDO tool, §6.7).
High jitter / occasional frame loss	Another application competing for the same CPU core, or the PC's power plan throttling — see the real-time notes in §1.3 and §13.4.

6.6 ESI file preview and chain builder — configuring topology before it's wired

§6.2 covers automatic matching: a slave that is already powered and wired reports its identity over the SII, and RasProLink looks it up in the watched ESI folder. The ESI File Preview card on the Slave Details card takes the opposite direction — you pick a single ESI XML file by hand, RasProLink parses it in the browser, and you can carry its Sync Manager sizes/addresses and CoE support into the currently selected slave's form, or use it to build a whole run of slaves before any hardware is on the ring at all.

Previewing a single ESI file

Select the slave to edit from the slave table first, then choose an ESI XML file under ESI File Preview. If the file describes several devices, pick the right one from the Device drop-down; the card then shows Vendor, Product Code / Revision, whether CoE (Mailbox) is supported, and the Outputs (Outputs SM) / Inputs (Inputs SM) sizes it declares. Carry these values into the form fills the selected slave's output/input size and physical-address fields from that ESI entry — you still click Save on the form afterwards, exactly as when typing the values in by hand (§6.3).

Building a chain: coupler + terminals

Modular IO on EtherCAT is usually a bus coupler (e.g. Beckhoff EK1100) with a run of terminals plugged into it (e.g. EL1008, EL2008); each terminal is its own slave on the ring, one position after the previous one. Instead of ESI-previewing and saving each of them one at a time, load the coupler's ESI file and click Add to chain, then repeat for every terminal in its physical, left-to-right wiring order — each Add to chain appends the currently previewed device to the Chain (Coupler + Terminals) table below, where its position, Product Code / Revision, name, output/input byte counts and CoE support are listed; a row can still be reordered with Move up/down or dropped with Remove before it is applied.

Set Starting ring position (it defaults to one past the last slave already in the list the first time you add to an empty chain) and click Apply chain to slave list: RasProLink walks the chain in order, assigning each entry the ring position starting position + index and a fixed station address of 0x1000 + ring position, then saves it with a normal PUT to /api/ethercat/slaves — the same call the manual form uses. If a slave already occupies that ring position, its entry is updated in place rather than duplicated, so re-applying a corrected chain over an existing one is safe. The result is an ordinary set of slave configuration entries, indistinguishable from ones entered by hand or matched by the ESI catalog (§6.2) — the chain table itself is only a staging area and is cleared once applied.

Tip: This lets you configure an entire coupler + terminal run from the vendor's datasheet ESI files at your desk, before the panel is wired — useful for pre-staging a configuration file to ship with a machine. Once the real hardware is wired and the bus opens, each slave's actual SII identity is checked against its configured Vendor ID/Product Code/Revision the same way as any other slave (§6.1); a mismatch (wrong terminal in the wrong slot) is visible immediately in the ESM State column of the slave table.

6.7 SDO (CoE) read/write — manual CANopen-over-EtherCAT access

Besides the cyclic PDO exchange that §6.3 describes, a slave that supports CoE (CANopen over EtherCAT) also exposes its object dictionary over the mailbox — individual objects addressed by a 16-bit index and 8-bit subindex, read or written one at a time via SDO (Service Data Object) transfers, independent of the cyclic frame. This is a manual, on-demand path: it does not run every cycle, and it is meant for commissioning and diagnostics, not for control-loop data (use tags/PDO mapping, §6.3, for anything that needs to move every cycle).

Select a slave in the slave table; the SDO (CoE) Read/Write card appears further down the Slave Details card, below Live IO (§6.9), only when that slave's ESI-declared CoE support is set — otherwise the card shows This slave does not support CoE (mailbox) and the fields stay hidden. Enter Index (hex, e.g. 1018 for the Identity Object) and Subindex, then click Read to fetch the current value as hex, or type Data (hex) and click Write to send it. Both go through the slave's mailbox rather than the cyclic PDO frame, so a read/write can take noticeably longer than one bus cycle and can fail on a slave that is not yet past PREOP.

Note: A failed SDO transfer reports the slave's CANopen abort code alongside the error message — the same abort code your slave's manual documents for that object (e.g. object does not exist, or write access denied for a read-only entry).

6.8 Network adapter selection and bus cycle time

§6.1 states the default 4 ms cycle time as a fact about the engine; this section is where both the adapter and that cycle time are actually set, on the Network Adapter Selection card at the top of the web UI.

Network Adapter Selection

EtherCAT works with raw Ethernet frames rather than IP, so the Master must be told which network adapter the ring is physically connected to. The Network Adapter dropdown lists every adapter RasProLink can see; leave it on (auto-select) to let the Master pick, or choose one explicitly. An optional MAC (manual, optional) field lets you pin the adapter by MAC address instead of by its OS-assigned name, which is useful on a PC where adapter names can shift after a driver update or a USB dock is reconnected. Clicking Apply restarts the bus — a NIC change affects the whole segment, so it cannot be applied without a brief reopen of the ring, exactly like a slave add/remove (§6.3) does.

Bus cycle time

The Cycle Interval (ms) field on the same card sets how often the Master drives the cyclic exchange (the 4 ms in §6.1 is only this field's default, not a fixed value). Enter a new interval and click Apply next to it: unlike the adapter, changing the cycle time does not rebuild the ring's FMMU/Sync Manager configuration, so it does not force a restart — the UI's own wording is explicit that Applied, the new value takes effect the next time the bus (re)opens, i.e. it is picked up on the next connection cycle rather than mid-stream.

Note: "Next time the bus reopens" happens sooner than a service restart in practice: stopping and starting the Master from the status card (§6.1), or any change that already triggers a restart (a NIC change above, or adding/removing a slave, §6.3), also reopens the ring and picks up the new cycle time.

6.9 Live IO — manual per-slave read/write in the web UI

§6.4 shows process data read/write from .NET client code; the same raw bytes are also reachable straight from the browser, without writing a line of code, through the Live IO panel on the Slave Details card.

Select a slave in the slave table to open its Slave Details card; below the identity fields (ESM State, Vendor ID, Product Code, Revision, Serial No.) the Live IO block shows Input (from slave, read-only, hex) — the slave's current input image, refreshed automatically and read-only — and Output (to slave, hex), an editable field: type a hex string and click Send to write it straight into that slave's output image on the next cycle. This is the same input/output byte range client code reaches with `GetDeviceIoAsync/WriteDeviceOutputAsync` (§6.4), exposed directly for bench testing without a client project — for example, forcing an output coil while wiring a panel, or confirming an input byte changes when a sensor is triggered, before any tag names are even defined (§6.3).

Note: A value sent this way is overwritten by the next cyclic write from a connected client or by a tag write (§6.3) — Live IO does not hold or latch an output, it only injects one write into the ongoing cyclic exchange. It is meant for bench testing and commissioning, not as a substitute for a running client application.

Chapter 7

EtherNet/IP Scanner

7.1 Overview

The EtherNet/IP Scanner engine makes your PC the Originator of one or more Class 1 (implicit, cyclic) CIP connections: it opens a Forward Open to each configured Target/Adapter, then exchanges input and output assemblies at the negotiated RPI (Requested Packet Interval), and re-opens the connection automatically if it drops.

Item	Value
Role	Originator
Connection class	CIP Class 1 (cyclic, UDP), one Forward Open per Target
Identification	Target IP + assembly instance numbers (input/output/config)

7.2 Adding a Target

Add a Target from the web interface's device card by entering its IP address and the assembly instance numbers from its EDS file or datasheet (input, output and, if used, configuration assembly), plus the desired RPI — or skip the manual lookup entirely by running a Device Discovery scan first (§7.5), which can pre-fill this same form. RasProLink opens the Forward Open immediately and the Target's row shows Connecting → Running once the first cyclic exchange succeeds — the same status progression used by the PROFINET Controller's device list (§9.2).

7.3 .NET client example

```
using RasProLink.Client;

using var client = new RasProLinkClient("localhost", 8080);
await client.ConnectAsync();

var targets = await client.GetDevicesAsync();
foreach (var t in targets)
    Console.WriteLine($"{t.Name} {t.IpAddress} [{t.State}] in={t.InputBytes}B out={t.OutputBytes}B");

var io = await client.GetDeviceIoAsync("vfd-conveyor-3");
await client.WriteDeviceOutputAsync("vfd-conveyor-3", slot: 0, subslot: 1,
    data: new byte[] { 0x01, 0x00 }); // e.g. run/stop word
```

7.4 Troubleshooting

Symptom	Likely cause
Forward Open rejected	Assembly instance numbers or sizes don't match the Target's EDS, or the Target already has the maximum number of connections open from another Scanner/PLC.

Symptom	Likely cause
Connection times out repeatedly	RPI set too aggressively for the network path, or a switch/VLAN is adding latency — see §2's real-time network guidance.

7.5 Device Discovery — finding Targets on the wire

The Device Discovery card sends a UDP ListIdentity broadcast and lists every EtherNet/IP device that answers, with its IP address, Vendor ID, Product Code, revision and product name — the EtherNet/IP counterpart of the PROFINET Controller's DCP network scan (§9.3). It runs for a fixed window (2 s by default) and does not by itself open any CIP connection; it is read-only reconnaissance.

Each result carries an EDS Match column: RasProLink looks the responding device's Vendor ID/Product Code/Device Type/revision up against the EDS Catalog (§7.6) automatically. "Use this IP" fills only the IP field of the Add-a-Target form (§7.2); "Auto-fill" is offered instead whenever a catalog match exists and additionally pre-fills the assembly instance numbers and input/output sizes from that matched EDS, so the Target can be added without looking anything up by hand.

Note: ListIdentity is a broadcast on the local subnet — a Target behind a router or a VLAN boundary will not appear here even though a Forward Open to its known IP would still succeed. Add it manually in that case.

7.6 EDS Catalog — automatic matching

The EDS Catalog card watches a folder of vendor EDS files and scans it in the background — the EtherNet/IP equivalent of the PROFINET GSDML catalog (§9.10). Drop the .eds files supplied by your device vendors into the folder (the default is ethernetip-scanner-eds, next to RasProLink's other configuration files) and RasProLink indexes each device definition's Vendor ID/Product Code/Device Type/revision together with its assembly layout, so Device Discovery (§7.5) can match and pre-fill against it without you opening a single EDS file by hand.

Field	Meaning
Folder	The watched directory; change it with the Change/Browse controls (a folder-picker dialog is offered when RasProLink runs on an interactive desktop, otherwise type the path directly).
Status / File count / Device definition count	Whether the last scan is loaded or still running, how many .eds files were read, and how many device definitions were extracted from them (one file can describe several device variants).
Reload Catalog	Forces an immediate re-scan of the folder without waiting for the background watcher — use it right after copying in new EDS files.

Note: A file that fails to parse is skipped, not fatal — it is reported by name in the catalog's error list so you can fix or replace it, while every other file in the folder still loads normally.

7.7 EDS File Preview — inspecting a file before adding a device

The EDS File Preview card lets you paste the raw text of a single EDS file and parse it on the spot, without adding it to the EDS Catalog folder (§7.6) and without creating a Target. It reports the vendor name and code, product code/type/name, revision, and every assembly instance defined in the file with its byte size and direction

(Input/Output) — useful for checking a file a vendor just emailed you, or for reading assembly numbers out of a device you are not ready to connect yet.

When the file's [Connection Manager] path resolves cleanly, a "Carry these values into the form" button appears and copies the parsed assembly instances and sizes straight into the Add-a-Target form (§7.2). If the direction cannot be determined from the file (no usable Connection Manager path), that button is hidden and you read the instance numbers off the table manually.

Chapter 8

EtherNet/IP Adapter

8.1 Overview

The EtherNet/IP Adapter engine makes your PC a Target: it listens for a Forward Open from a Scanner — typically a PLC — and, once accepted, exchanges the configured input/output assemblies at the RPI the Scanner requested. This is the mirror image of the Scanner engine: here your PC is the device being scanned, not the one doing the scanning.

Item	Value
Role	Target / Adapter
Identification to the Scanner	Configured assembly instances (input/output/config) and their byte sizes

8.2 Configuring the local assemblies

From the web interface, define the input assembly (data your PC produces, read by the Scanner) and output assembly (data the Scanner writes, consumed by your PC) with their instance numbers and byte sizes — matching whatever the connecting Scanner/PLC's own configuration (or EDS) expects. Once saved, RasProLink starts accepting Forward Open requests for that instance.

8.3 .NET client example

```
using RasProLink.Client;

using var client = new RasProLinkClient("localhost", 8080);
await client.ConnectAsync();

// Read what the Scanner/PLC most recently wrote to us
var io = await client.GetDeviceIoAsync("self");

// Publish new input data for the Scanner/PLC to read next cycle
await client.WriteDeviceOutputAsync("self", slot: 0, subslot: 1,
    data: new byte[] { 0x2A, 0x00, 0x00, 0x00 });
```

8.4 Troubleshooting

Symptom	Likely cause
Scanner cannot open a connection	Assembly instance/size mismatch between the Scanner's configuration and this engine's configured assemblies, or the assembly is not yet saved/started.

8.5 Connection status and statistics

Two cards on the Adapter's web page give you live visibility into the connection a Scanner holds with this engine, without needing a REST call or a packet capture — there is no separate device list here as in the Scanner engine (§7), since the Adapter only ever has the one connection a Scanner opens to it.

Connection Status

Shows whether a Scanner is currently connected and, if so, its IP address and the time the connection was established ("Connected at"), plus the two CIP connection IDs the Forward Open negotiated — Input Connection ID (the O→T connection carrying what this engine receives) and Output Connection ID (the T→O connection carrying what it sends) — and the timestamp of the last input received from the Scanner, useful for noticing a Scanner that has gone silent without formally closing the connection.

Statistics

Reports Forward_Open Attempts (every accepted or rejected Forward Open request this engine has handled, including ones from before the current connection), Connection Losses (how many times an established connection has dropped), Frames Sent and Frames Received (cyclic I/O frame counters since the engine started), and Jitter, charted the same way as the PROFINET Controller's device-level jitter chart (§9.4, §13.4) — a wide or growing jitter reading points at the same network-path causes covered in §7.4's troubleshooting table.

Note: There is no REST endpoint or UI list that enumerates other Scanners/PLCs that might also be configured to open a connection to this Adapter — an Adapter Target only ever serves the one Forward Open it currently holds, so "who is connected" is exactly what the Connection Status card above already shows.

8.6 I/O Monitoring / Simulation — bench testing without a Scanner

The I/O Monitoring / Simulation card lets you exercise this engine's I/O without any real Scanner or field hardware attached. It shows the current input assembly as read-only hex (the data most recently written by a connected Scanner, or all zero if none is connected) and offers a hex field plus a Send button for the output assembly, letting you manually inject the bytes a real Scanner would otherwise write — useful for testing tag definitions (§5) or an integration against this engine before a PLC program exists.

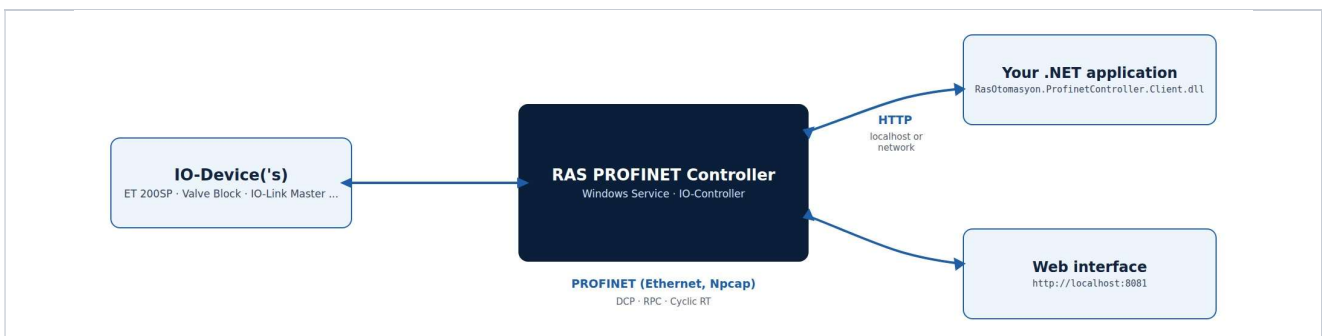
Tip: "Output" and "Input" here follow the Scanner's point of view, matching the field names of §8.2: what you type and Send in this card is exactly the data the connected Scanner reads as its input assembly.

Chapter 9

PROFINET Controller

9.1 Overview

The PROFINET Controller engine turns your Windows PC into a real PROFINET IO-Controller — the role normally held by a PLC. Over a real network adapter, using the same protocols a PLC uses — DCP for discovery, RPC for connection setup, cyclic real-time frames for running data exchange — it connects directly to one or more third-party PROFINET IO-Devices: remote I/O blocks, valve islands, IO-Link masters, GSDML-described devices of any kind. No PLC needs to sit in between.

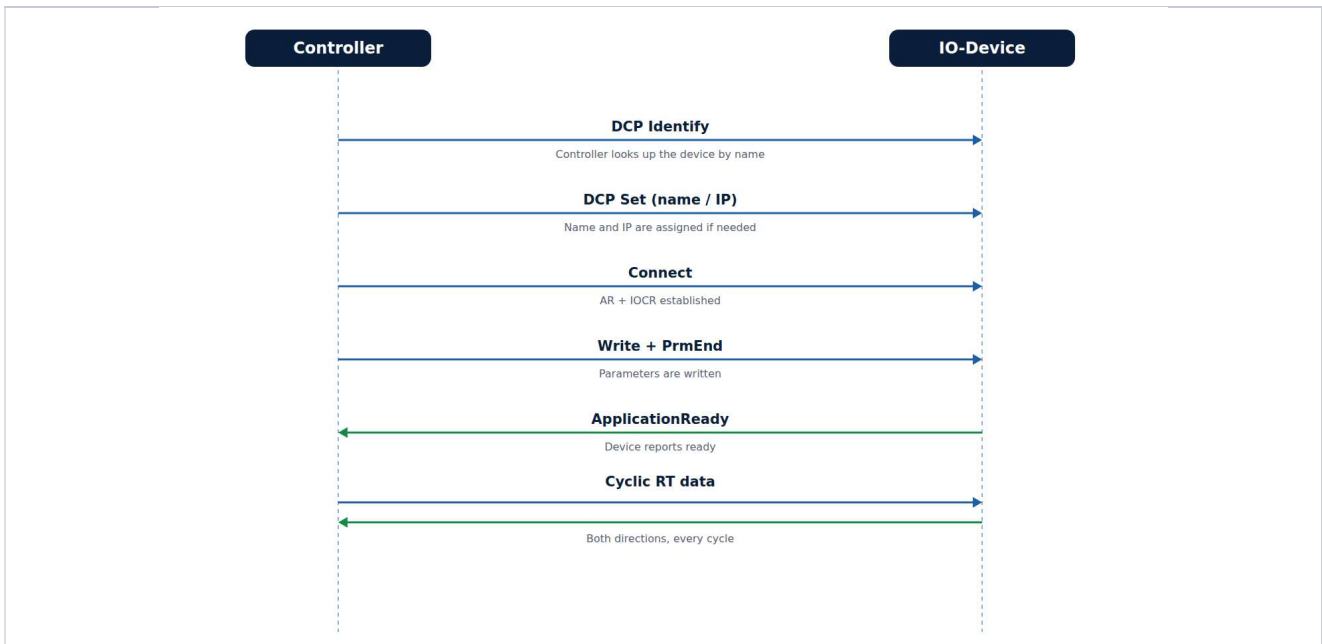


IO-Devices on one side, your application and the web interface on the other — the Controller service sits in the middle

A single Controller instance is not limited to one device: it opens one network adapter and one RPC socket, and every configured IO-Device is served over it — device count is a licensing/practical question, not an architectural one.

9.2 Connection sequence

Every device goes through the same sequence, shown as its status in the device list: Resolving (DCP Identify/Set) → Connecting → Parametrizing → WaitingApplicationReady → Running.



DCP Identify/Set, Connect, Write+PrmEnd, ApplicationReady, then cyclic RT data both ways every cycle

9.3 Adding IO-Devices

IO-DEVICES

Upload the XDB file exported from TIA Portal; every device inside it is added to the list automatically and, if enabled, connection starts right away.

Drop the XDB file here, or click to choose one

DEVICE	IP	STATE	INPUT / OUTPUT	CYCLE	ENABLED	
ET 200SP I/O	192.168.0.21	Running	1B / 1B	4 ms	☒	Delete
Valve Block 2	192.168.0.22	Error	0B / 2B	8 ms	☒	Delete
IO-Link Master 3	192.168.0.23	Offline	32B / 32B	16 ms	☐	Delete

IO-Devices card — drop a .xdb file to add every device it contains in one step

Two independent ways exist to add an IO-Device, and only one of them needs TIA Portal. A Network Scan finds every PROFINET device on the wire via DCP and, for each one, checks its Vendor/Device ID against the GSDML catalog automatically — no .xdb involved at all (§9.10). The steps below cover the second way: importing a .xdb file exported from TIA Portal.

The Controller does not need a GSDML file directly for this path — instead, it reads a device proxy file (.xdb) that TIA Portal already exports for a device configured in a project. This file already contains everything the Controller needs to know: the device's station name, IP configuration, and every module's slot/subslot layout with input/output byte lengths and parameters — none of it needs to be re-entered by hand. See §9.10 for the alternative path that skips TIA Portal and the .xdb file entirely.

Column	Meaning
Device	Display name (falls back to the station name); a device with a module/submodule mismatch also shows a small "Module mismatch" warning underneath (§9.4).
IP	The device's own IP address.
Status	Running (cyclic data flowing), Fault (last retry failed), or Off (disabled / never tried).
In / Out	Total input and output byte count across all the device's submodules.
Cycle	Configured update time.
Enabled	Untick to stop connecting without removing the device's configuration.

9.4 Device details, live IO and tags

DEVICE DETAILS

Module Structure

SLOT	MODULE	SUBSLOT	SUBMODULE	INPUT	OUTPUT	PARAMS
1	IM 155-6 PN HF (Head Module) (0x00000001)	0x0001	Interface (0x00000001)	0B	0B	0
2	8DI DC24V (0x0000AB12)	0x0001	8DI DC24V (0x0000AB12)	1B	0B	3
3	8DO DC24V/0.5A (0x0000AB20)	0x0001	8DO DC24V/0.5A (0x0000AB20)	0B	1B	2

Live IO

SLOT	MODULE	DATA (HEX)	STATUS
2	8DI DC24V	9C	OK
3	8DO DC24V/0.5A	3F Write	OK

Tags

In addition to raw data (Live IO above), give a byte/bit location a NAME and TYPE — reading/writing can then be done by that name. Tags can also be defined for devices added via XDB.

NAME	SLOT	DIRECTION	TYPE	LOCATION	VALUE	
Sensor_Input1	2 / 0x0001	Input	bool	@0.0	false	Edit Delete
Motor_Run	3 / 0x0001	Output	bool	@0.0	true Write	Edit Delete

1/0x0001 — Interface (in 0B / out 0B)

Input (from device)

Int16

Statistics

Sent	Received	Alarms	Connect attempts
812940	812935	0	1
Connection losses	Jitter		
0	0.280 ms		

Device detail — module structure, live IO with a manual hex-write field, tags, and connection statistics including jitter

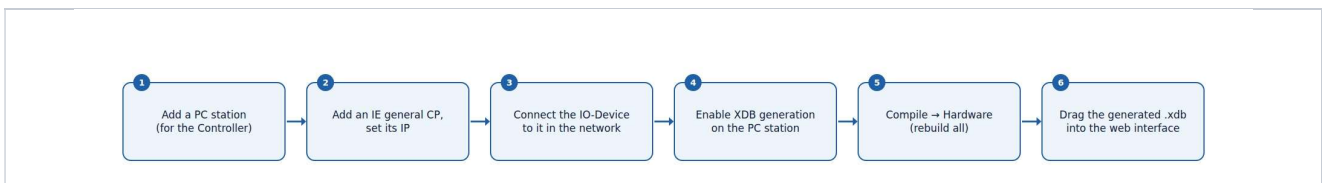
Clicking a device in the list opens three sections: Module structure (the slot/subslot layout as imported from the .xdb or configured from the GSDML catalog, read-only here), Live IO (each submodule's current raw bytes, with a hex field and a Write button on output submodules for quick bench tests), and Tags (name a byte/bit range so it can be read or written by name instead of by raw hex — rules in §5). Statistics at the bottom report frames sent/received, alarms, connection retries, connection losses, and cycle jitter (§13.4) — jitter also gets its own live chart further down the page, tracking running min/max/average instead of just the instantaneous number in the stats row.

On every Connect, the device reports back its actual per-slot/subslot module and submodule idents, and the Controller compares them against the configured layout. A slot with no module fitted, the wrong module fitted,

or a submodule identity that doesn't match flags the device in the list with a small "Module mismatch" warning under its name (§9.3) — the device can still reach Running with a mismatch present, since some submodule states (e.g. a substitute value) don't block data exchange. Opening the device detail view then shows a "Module diff" line listing every slot and subslot the device actually reported next to its state (no module / wrong module / proper module / substitute at module level, or ok / substitute / wrong / none per submodule) — use it to find which physical module was swapped, left out, or wired to the wrong slot without walking the panel.

9.5 TIA Portal — exporting a device proxy file

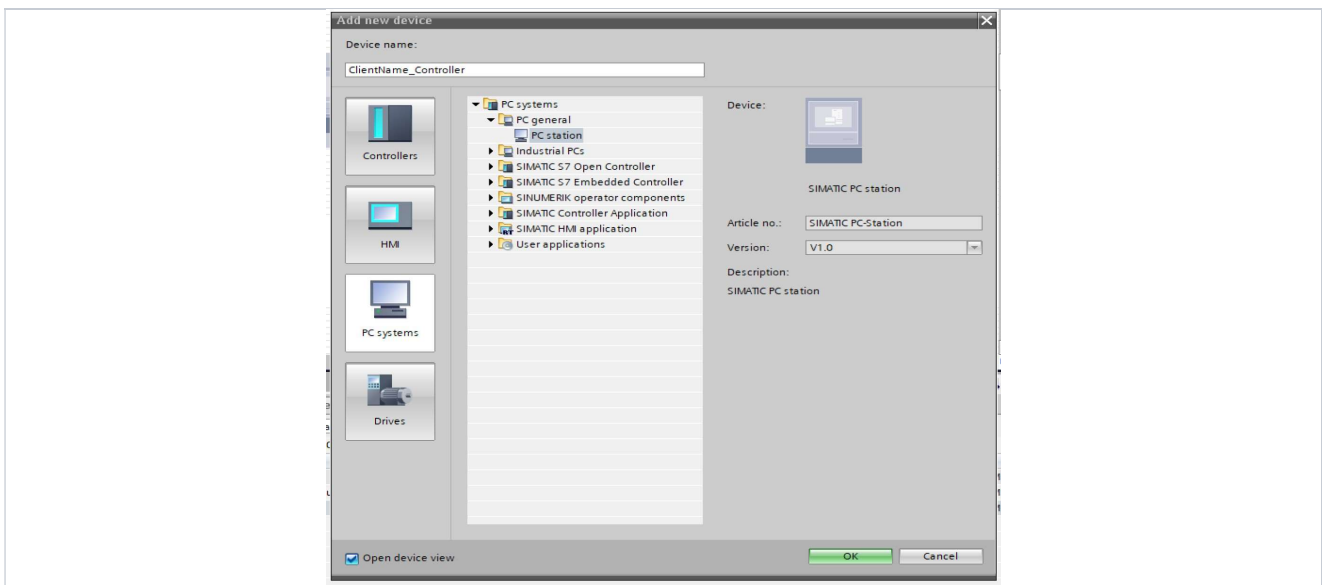
If the IO-Device already sits in a real PLC project (S7-1200/1500 etc.), right-click it and export its device proxy file directly. Most setups, however, have no physical PLC at all — the Controller itself takes that role — so the TIA project needs a station to represent it. The following steps, with screenshots from a real TIA Portal project, show that common path.



Overall flow — details for each step below

1. Add a PC station for the Controller

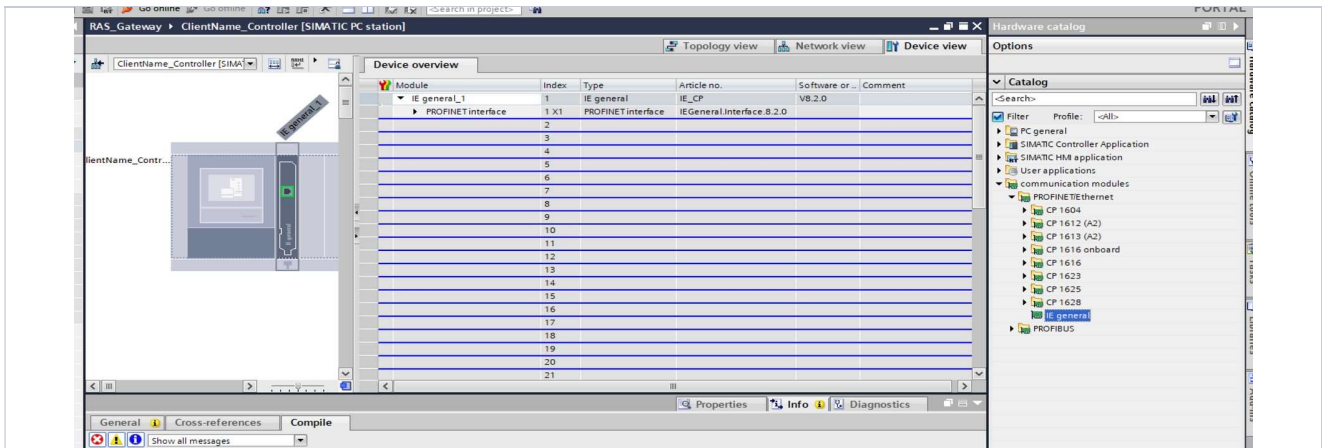
In the project tree, open Add new device, select PC systems → PC general → PC station, and give it a descriptive name (e.g. <Customer/Site>_Controller).



Add new device → PC systems → PC general → PC station

2. Add a PROFINET interface (IE general)

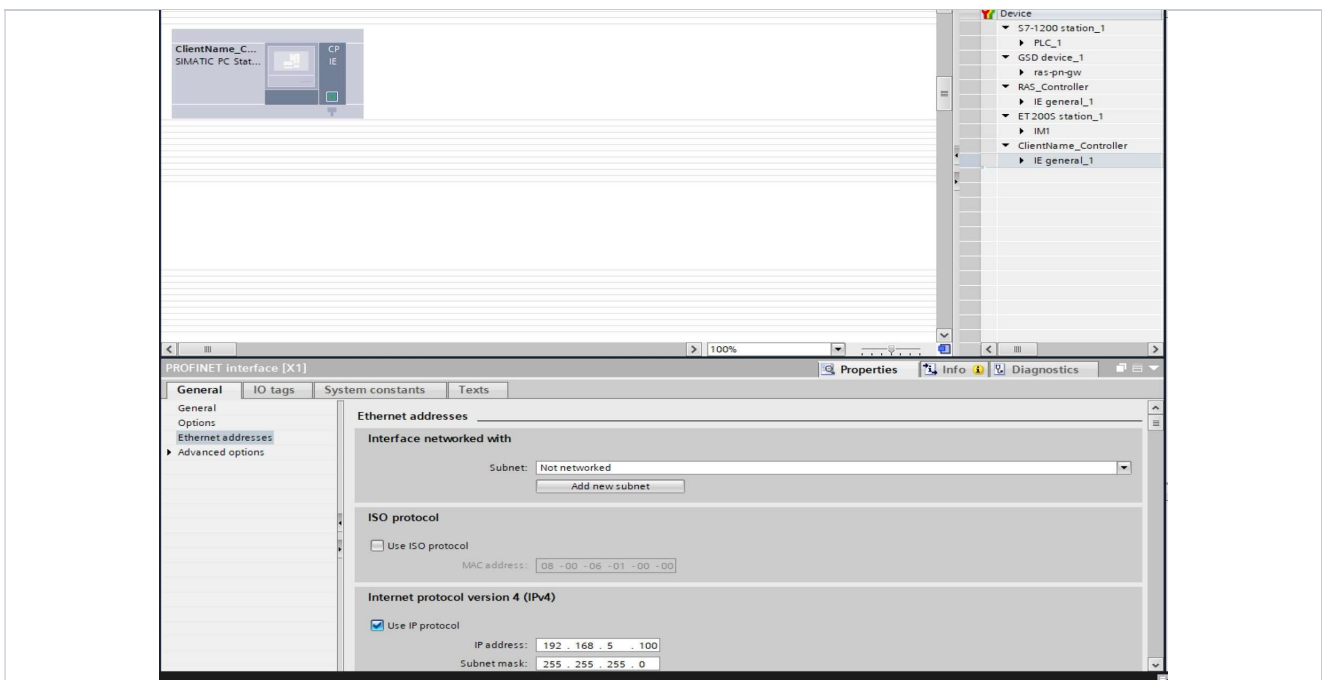
From the hardware catalog, drag Communication modules → PROFINET/Ethernet → IE general onto the PC station's rack — this module represents the Controller's PROFINET-facing network adapter.



IE general module added to the PC station, hardware catalog on the right

3. Set the Controller's IP address

On the new interface's Properties → Ethernet addresses page, enter the IP address and subnet mask this PC will really use (add a new subnet if needed) — it must be on the same subnet as the IO-Devices.

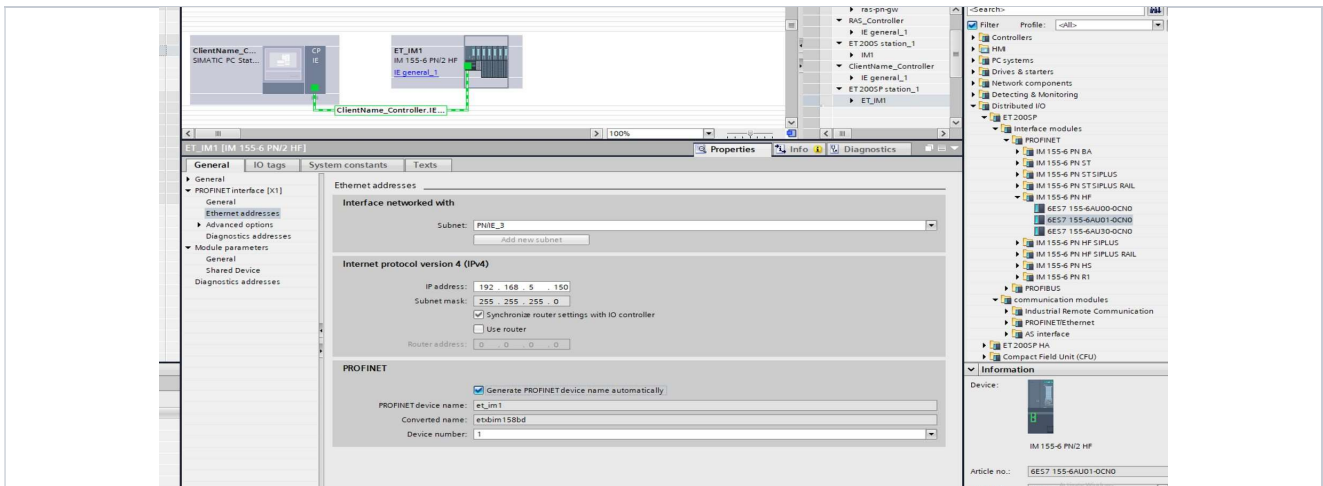


PROFINET interface → Ethernet addresses: IP and subnet

Note: The IP entered here is only TIA's own internal reference — TIA cannot assign it to a real network card. That this IP is really assigned to the Controller PC's network card (and shown as the same IP in the Controller's own settings, §9.6) remains your responsibility.

4. Add the IO-Device and link it in the network

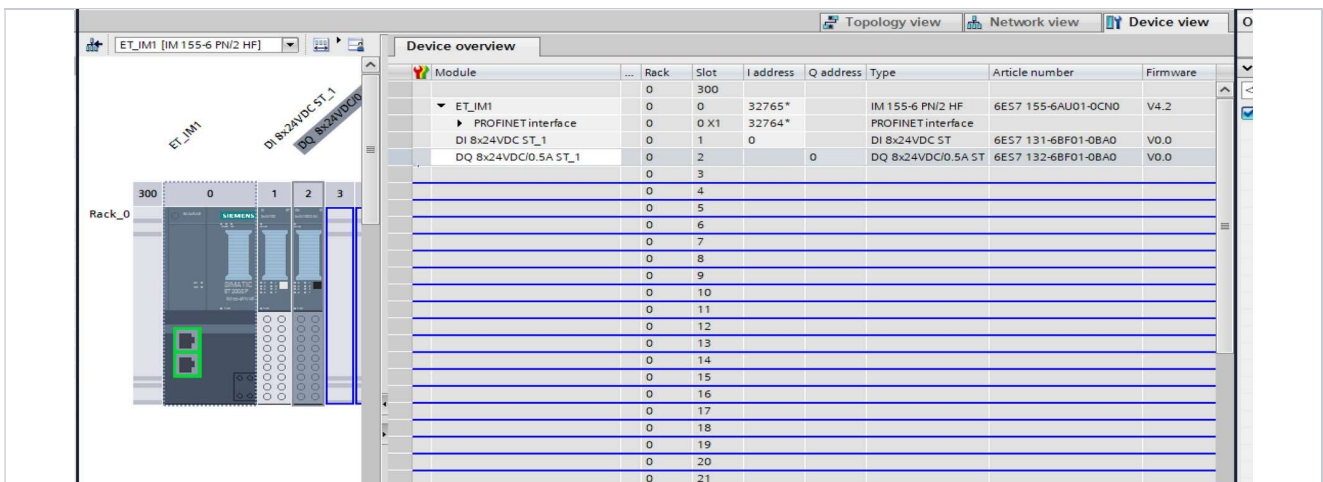
Add the IO-Device to the project (import its GSDML first if the hardware catalog doesn't have it yet), then in Network view draw a PROFINET connection with the mouse from the PC station's interface to the IO-Device's interface — this puts both into the same IO system. Set the IO-Device's own station name and IP (same subnet as the Controller) on its Ethernet addresses page.



Network view: PC station ↔ IO-Device link; the IO-Device's Ethernet addresses page

5. Configure the IO-Device's modules

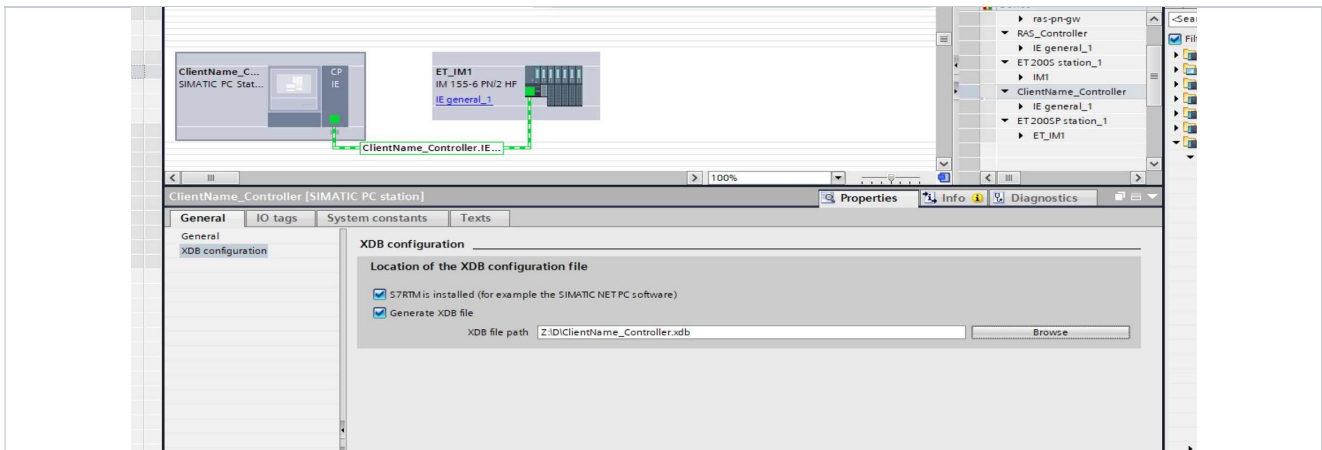
In the IO-Device's own device view, drop in the physically installed modules (DI, DQ, analog, etc.) exactly as fitted — this determines the slot/subslot layout and input/output lengths that end up in the .xdb file.



IO-Device device view: modules installed (e.g. DI 8x24VDC, DQ 8x24VDC/0.5A)

6. Enable XDB generation on the PC station

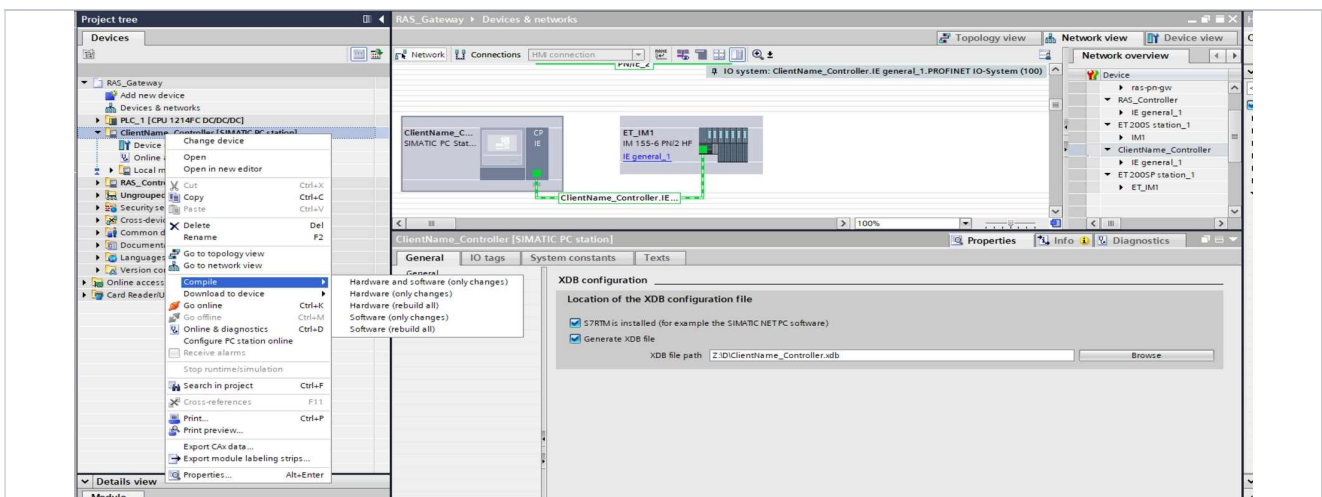
On the PC station (not the IO-Device), open Properties → General → XDB configuration, tick Generate XDB file, and set the path to write it to — a location reachable from the PC running the Controller service.



PC station → Properties → XDB configuration: "Generate XDB file" and the file path

7. Compile to produce the .xdb file

Right-click the PC station and choose Compile → Hardware (rebuild all) (Hardware (only changes) is enough for small later edits). TIA writes the .xdb file to the configured path during compilation.



PC station right-click → Compile → Hardware (rebuild all)

Tip: If the same PC station has more than one IO-Device linked, ONE .xdb file already contains all of them — importing it into the Controller adds every device in one step.

8. Import the .xdb into the Controller

Drop the produced file into the Controller's web interface upload area (§9.3). The import response lists how many devices were imported and any warnings — e.g. a slot the importer could not fully interpret.

9.6 Controller settings and network adapter

CONTROLLER SETTINGS

Station name and IP used when connecting to devices. PN-IO RPC runs over real UDP/IP, so this IP must actually exist on the network adapter.

Vendor ID	Device ID	PROFINET Layer	RPC Endpoint
0x0C00	0x0002	Running	192.168.0.1:34964

Station Name	IP Address	Subnet Mask
ras-controller	192.168.0.1	255.255.255.0

Gateway	
0.0.0.0	

☒ Also apply this IP to the Windows adapter (netsh ... add address, as a secondary address)

Apply

Controller settings — station name, IP/subnet/gateway, and the option to also apply the IP to the Windows network adapter

This is the Controller's own identity on the PROFINET network: station name and IP/subnet/gateway. This IP does not change — it is the address the Controller uses to run DCP and RPC. Since RPC needs a real address on the wire, tick "Also apply the IP to the Windows network adapter" so the address really exists on the selected card (added as a secondary address; existing addresses are not removed).

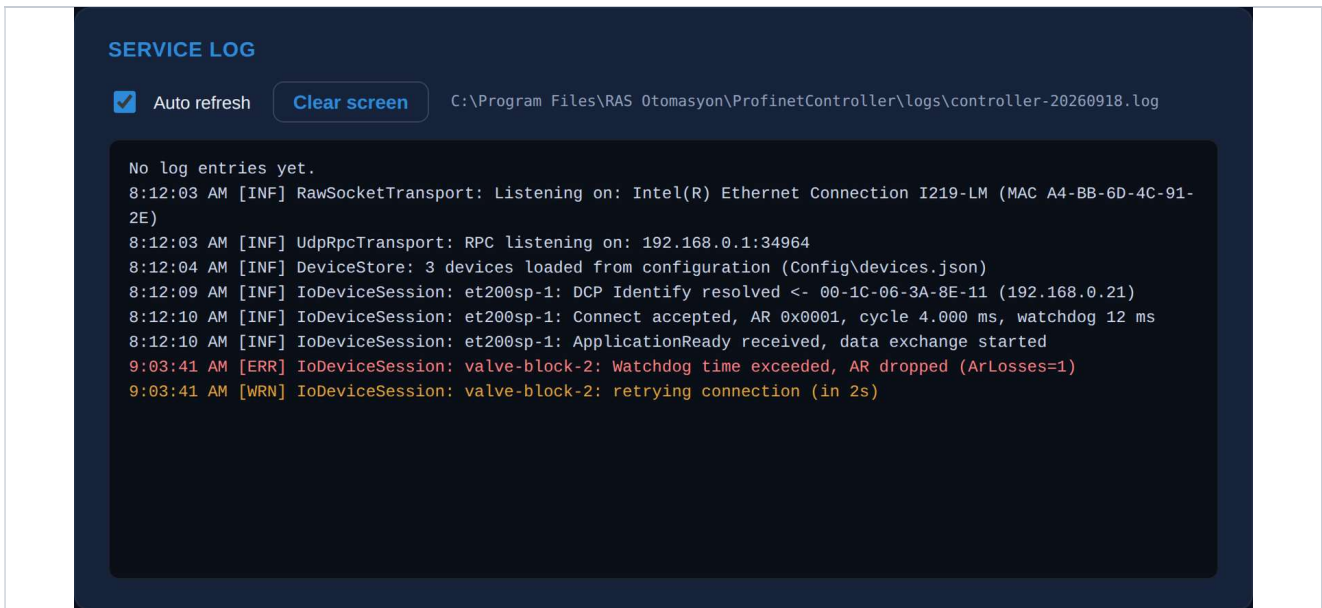
Note: Changing the IP here restarts the PROFINET layer — devices reconnect automatically but drop briefly. Do this in a planned window, not on a running production line.

Manually stopping and starting the PROFINET layer

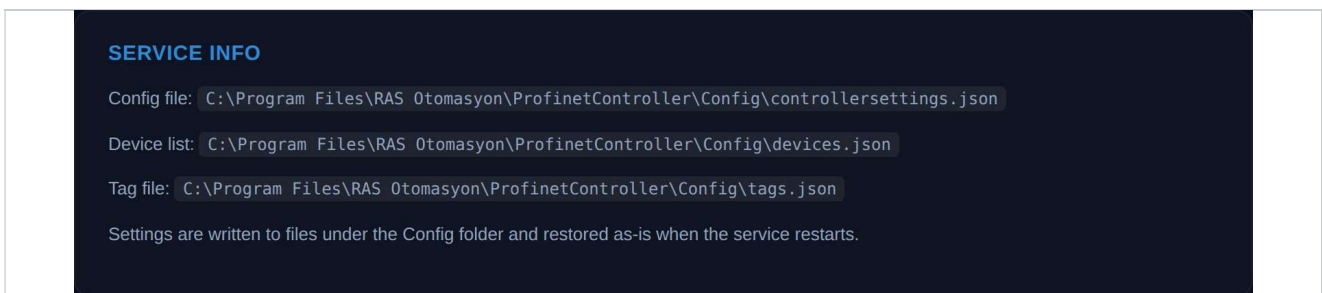
The same card carries a single Stop/Start toggle button, independent of any individual device's Enabled checkbox (§9.3): Stop tears down every running device session and then closes the network adapter and RPC socket entirely (POST /api/stop), while Start reopens them and reconnects every enabled device (POST /api/start). Confirming Stop cuts the connection to all IO-Devices immediately, and no automatic reconnection is attempted until Start is pressed again — the layer stays down until an operator explicitly restarts it. Use this before rewiring or power-cycling the PROFINET segment, so the Controller doesn't spend that time retrying connections that cannot succeed.

Warning: Stop is a hard cut, not a pause: it does not wait for a safe point in the IO-Devices' cycle. Treat it the same as pulling the PROFINET cable — make sure the connected equipment's own logic is in a state that tolerates a sudden loss of outputs before pressing it.

9.7 Service log and paths



Service log card — live, scrolling view of connection attempts, DCP resolution, errors and reconnects



Service info card — exact paths to the settings, device list and tag files on disk

Auto-refresh polls the log every two seconds; untick it to freeze the view while reading (this only clears the on-screen list, never the log file on disk). The Service info card is handy when attaching the log file to a support request or editing devices.json by hand.

9.8 .NET client example

```
using RasProLink.Client;

using var client = new RasProLinkClient("localhost", 8080);
var status = await client.ConnectAsync();
Console.WriteLine($"{status.StationName}: {status.DeviceCountRunning}/{status.DeviceCount} devices running");

var detail = await client.GetDeviceAsync("et200sp-1");
var io = await client.GetDeviceIoAsync("et200sp-1");

await client.WriteDeviceOutputAsync("vana-blok-2", slot: 1, subslot: 1,
    data: new byte[] { 0x01 });

client.DevicesChanged += (s, list) => Console.WriteLine($"{list.Count} devices");
await client.StartPollingAsync();
```

9.9 Troubleshooting

Symptom	Check
Web interface never loads	Is the service running (services.msc)? Is port 8080 already used by something else?
No network adapters listed	Npcap not installed, or it was installed after the service started — install it and restart the service.
Device never leaves Resolving	No DCP Identify response — check cabling, and that the selected adapter (§4.3) is really wired to it.
Device cycles Running → Fault → Running	Watchdog time being approached — check jitter in the device detail card (§9.4) against the configured update time; usually a busy/overloaded network segment.
XDB import rejected	Not a device proxy export, or exported by an incompatible TIA Portal version — re-export and retry.

9.10 GSDML catalog — adding devices without TIA Portal or an .xdb

Every discovered or catalog-added device still goes through the same DCP/RPC/cyclic exchange as §9.2 — the GSDML catalog only changes how a device's slot/subslot layout gets into the Controller, replacing the TIA Portal + .xdb round-trip (§9.3, §9.5) with a folder of the vendor's own GSDML files.

Network Scan (DCP)

Network Scan sends a DCP Identify-All broadcast and lists every PROFINET station that answers — third-party IO-Devices, other Controllers, even the Controller's own interface — without needing an XDB or GSDML for any of them, similar to TIA Portal's Accessible devices. Each row shows the device's name, IP, MAC and Vendor/Device ID; the GSDML Match column shows the catalog's best match for that Vendor/Device ID, with its input/output byte totals, or No catalog match if nothing fits. A device already in the configuration shows Reconfigure instead of Add.

GSDML catalog folder

The catalog scans a folder for .xml/.gsdml files in the background — by default Config\profinet-controller-gsdml next to the other files shown on the Service info card (§9.7), created automatically the first time the Controller starts. Drop the vendor's GSDML file(s) into it and click Reload (or wait for the next automatic rescan) — no service restart needed. The GSDML Catalog card reports file count, how many device definitions were parsed, when it last loaded, and any per-file parse errors. A different folder can be set from the same card (type the path, or Browse — the folder picker only works when the service has an interactive desktop session; typing the path always works) and is remembered across restarts.

Adding a device straight from the catalog

Clicking Add next to a GSDML match — from either the Network Scan results or the catalog's own device list — opens a form pre-filled with that device's type: enter the station name, IP/subnet/gateway, and assign the catalog's optional plug-in modules to slots (Slot 0 is always the fixed DAP itself). Submitting builds a complete device configuration on the spot; the connection parameters an .xdb would normally carry (AR, IOCRs, Alarm CR)

are filled in with the same defaults the Controller uses for any manually-built device, so the new device connects exactly like an XDB-imported one. A compact device with no plug-in modules skips the slot table entirely.

Note: The GSDML catalog only ever supplies the module/slot layout — it never overrides a device already added via .xdb, and the two paths can be mixed freely: some devices imported from an .xdb, others added straight from the catalog, side by side in the same IO-Devices list (§9.3).

Chapter 10

PROFINET Device

10.1 Overview

The PROFINET Device engine is the mirror image of the Controller: it makes your PC a PROFINET IO-Device — a Slave — to a real PLC. It simulates a device whose module/slot layout matches the GSDML file shipped with your RasProLink Device license; once your slot configuration mirrors that GSDML, PLC ↔ IO communication is ready out of the box, with no PLC-side programming beyond a normal hardware configuration import.

Item	Value
Role	IO-Device (Slave)
Described by	A GSDML file shipped with the Device license (maintained outside RasProLink, not generated by it)
Discovered by the PLC via	DCP (station name, then IP if needed)

10.2 GSDML — what it is and how it's used

GSDML (General Station Description Markup Language) is the XML dialect every PROFINET device publishes to describe itself: its identity (Vendor ID, Device ID), its module catalog, and each module's input/output data layout. A PLC engineer imports this one file into TIA Portal (or any other PROFINET engineering tool) and the device appears in the hardware catalog exactly like a real Siemens/vendor part — draggable onto the network, with its modules ready to configure.

RasProLink does not generate or export a GSDML file — the file is a fixed artifact maintained outside the product and shipped with your RasProLink Device license. What RasProLink does is let you build a slot configuration that must mirror that file's module list exactly: the module catalog below is fixed on the RasProLink side, and it is only usable if it matches what the same modules describe in the GSDML that TIA Portal (or any other engineering tool) has imported. Adding a genuinely new module size is not a configuration-screen action — it requires the GSDML file itself to be updated and re-imported into TIA Portal first, after which the matching size becomes selectable here.

Module	Input / Output
8 Byte In/Out	8 bytes in, 8 bytes out
16 Byte In/Out	16 bytes in, 16 bytes out
32 Byte In/Out	32 bytes in, 32 bytes out
64 Byte In/Out	64 bytes in, 64 bytes out

These four symmetrical modules are the entire catalog — slot 0 is always the fixed DAP itself, and up to 8 data slots (slots 1–8) can each take one of these four sizes. This covers most point-to-point data-exchange needs out of the box; pick whichever size fits the payload for each slot. If your application needs asymmetrical or larger

blocks, that is a GSDML change, and — as above — has to be requested and rolled out at the GSDML level, not configured from RasProLink alone.

10.3 Commissioning against a real PLC

6. Get the GSDML file for your RasProLink Device license (see §10.2) and import it into TIA Portal, if it isn't already in the hardware catalog: Options → Manage general station description files (GSD).
7. On RasProLink's web interface, set the Device's station name and pick, for each slot, the module size that matches the corresponding slot in the imported GSDML — see the module catalog in §10.2.
8. In the PLC project, drag the device from the hardware catalog onto the network, connect it to the PLC's PROFINET interface, and assign the same station name and an IP on the PLC's subnet.
9. Download the hardware configuration to the PLC. It resolves the Device by station name over DCP, connects, and cyclic data starts flowing — visible on RasProLink's own device status the same way an IO-Device shows on the Controller (§9.2).

10.4 .NET client example

```
using RasProLink.Client;

using var client = new RasProLinkClient("localhost", 8080);
await client.ConnectAsync();

// Read what the PLC most recently wrote to us (its outputs, our inputs)
var io = await client.GetDeviceIoAsync("self");

// Publish new input data for the PLC to read next cycle
await client.WriteDeviceOutputAsync("self", slot: 1, subslot: 1,
    data: new byte[] { 0x00, 0x2A });
```

10.5 Troubleshooting

Symptom	Likely cause
PLC never finds the device	Station name mismatch (case-sensitive, DNS-safe characters only), or PLC and Device not on the same subnet.
Module mismatch alarm in the PLC	The slot layout configured on RasProLink no longer matches the GSDML that was imported into the PLC project — re-check each slot against the GSDML file's module list (§10.2) and correct the mismatched slot(s).

10.6 Diagnostics

The Diagnostics card is a commissioning and wiring-verification tool: it lets you inject known test patterns into the data the Device sends to the PLC, pin a single tag or byte to a fixed value, simulate a module fault, and reset the running cycle statistics — all without touching the PLC program. These tools affect what the PLC actually receives, so treat them as bench/commissioning aids: while a force or test pattern is active, the values your own

application writes are overridden, and RasProLink does not turn them off automatically when you close the page.

Test patterns

To verify from the PLC side that data is genuinely flowing end to end, pick a pattern, a slot (or leave it on the first available slot), and an update interval, then apply it: Counter increments the first byte of the slot's input data every interval; Ramp writes byte $n = \text{counter} + n$, which also reveals an offset shift if the PLC-side mapping is wrong; Toggle alternates the whole submodule between 0x00 and 0xFF; Random writes random bytes, useful purely to confirm the data path is transparent (no byte getting stuck or masked somewhere in between). Only one pattern runs at a time; switching to Off restores normal application data.

Signal force

Forcing pins a chosen input tag to a value you enter (as hex), independent of the test pattern and of whatever your own application is writing underneath. The underlying value is not erased — it keeps being written and tracked, only overridden on the wire — so removing the force (per-tag, or Clear All) makes the true value reappear immediately, with nothing to reconcile. The active forces table lists each one's slot, offset, tag name and forced value with a Remove action; this is the tool for confirming a specific downstream signal (an alarm bit, an analog reading) drives the expected PLC behavior before real field wiring is connected.

Simulated fault and statistics reset

"Simulate a module fault (IOPS = Bad)" forces the submodule's IOPS status byte to Bad without disconnecting anything — this is how you check that the PLC program actually reacts to a bad data-quality indication (e.g. a module-fault OB, a quality bit gating downstream logic) instead of silently trusting whatever bytes happen to be in the process image. Turn it back off to restore normal (Good) status. Reset statistics zeroes the cycle counters (frames sent/received, jitter, etc.) shown elsewhere in the interface, which is useful right before a timed test run so earlier startup transients don't skew the numbers.

Note: The diagnostics stat tiles (Forced Bytes, Test Pattern) at the top of the card give an at-a-glance summary of what is currently overridden — check them before handing a station over, so nothing forced during commissioning is left active in production. The underlying REST endpoints (GET/POST /api/diagnostics and its sub-routes) are listed in Appendix A.

Chapter 11

.NET Client Library

RasProLink.Host can host any of the five engines (EtherCAT Master, EtherNet/IP Scanner, EtherNet/IP Adapter, PROFINET Controller, PROFINET Device) in one process, but only one is active at a time — switching engines from the web interface's protocol picker restarts the process with that engine (§4.4). The active engine's REST surface differs from engine to engine, most visibly in how a tag write specifies which target (device/slave/scanner target) it goes to:

Engine	Tag write endpoint
EtherCAT Master	POST /api/ethercat/slaves/{id}/tags/write — {id} is the slave's internal GUID.
EtherNet/IP Scanner	POST /api/ethernetip/scanner/targets/{id}/tags/write — {id} is the target's internal GUID.
PROFINET Controller	POST /api/devices/{name}/tags/write — {name} is the PROFINET station name, no GUID involved.
EtherNet/IP Adapter	POST /api/ethernetip/adapter/tags/write — no target/id, one fixed local IO set.
PROFINET Device	POST /api/io/write-tags — no target/id, one fixed local IO set.

RasProLink.Client hides this difference completely. Every common method — including WriteTagAsync — keeps the exact same signature no matter which engine is active: internally, ConnectAsync (GET /api/engine) learns which engine is active, and each call is translated to that engine's real REST path (the name→GUID lookup for EtherCAT/Scanner is resolved once via GetTargetsAsync and cached). A line such as

```
await client.WriteTagAsync("IM1", "Motor_Run", true);
```

keeps working unchanged no matter which engine RasProLink.Host is running (on EtherNet/IP Adapter and PROFINET Device, the "IM1" target name is simply ignored, since those two engines have one fixed local IO set). An engine-specific capability that does not fit this common flow (e.g. raw EtherCAT SDO access, PROFINET .xdb import) is added as a separate, engine-specific method instead — it never changes WriteTagAsync/ReadTagsAsync's own behavior.

Beyond that design, the library is a small, dependency-free package your own .NET application references to connect to the service, read engine status and targets, and read/write tags in code — instead of driving the web interface. It ships as two DLLs:

Assembly	Contains
RasProLink.Client.Abstractions.dll	Shared, engine-independent data types (EngineKind, EngineStatus, TargetSummary, TagDefinition, TagValue...) and a small self-contained JSON reader/writer.
RasProLink.Client.dll	The real client (RasProLinkClient), built on HttpClient.

Note: Neither DLL has any NuGet dependency, and both target netstandard2.0 and net8.0 — the same library works unchanged from an old .NET Framework 4.7.2 WinForms application to a modern .NET 8 service, with no binding-redirect surprises.

11.1 Setup and quick start

Reference both DLLs from your project (no NuGet feed needed — plain file references, or compile the projects from source and add a project reference).

```
using RasProLink.Client;
using RasProLink.Client.Abstractions;

using var client = new RasProLinkClient("localhost", 8080);

var active = await client.ConnectAsync();           // GET /api/engine — which engine is
active
Console.WriteLine($"Active engine: {active}");

var status = await client.RefreshStatusAsync();
Console.WriteLine($"{{status.RoleDisplayName}}: enabled={{status.Enabled}}
connected={{status.Connected}}");

var targets = await client.GetTargetsAsync();
foreach (var t in targets)
    Console.WriteLine($"{{t.Name}} @ {{t.IPAddress}} [{{t.State}}] in={{t.InputBytes}}B
out={{t.OutputBytes}}B");
```

ConnectAsync does not have to be called explicitly — every other method calls it automatically on first use if needed. Calling it yourself is only useful to know or log the active engine up front; after that, IsConnected and ActiveEngine read the cached result without another request.

11.2 Engine status and targets

```
var info = await client.GetEngineInfoAsync();           // full engine-selection list, not just the
active one
Console.WriteLine($"Active: {info.Active}");
foreach (var opt in info.Available)
    Console.WriteLine($" {{opt.DisplayName}} ({{opt.Kind}}) productionReady={{opt.ProductionReady}}");

var status = await client.RefreshStatusAsync();           // normalized snapshot of the ACTIVE
engine
Console.WriteLine(status.Message ?? "(no message)");
```

GetEngineInfoAsync returns the same list the web interface's protocol picker shows (§4.4) — every engine RasProLink.Host could switch to, not only the active one; use it to build an engine picker of your own, or to check ProductionReady before offering a switch. RefreshStatusAsync, by contrast, always describes the currently active engine only, normalized into the common EngineStatus shape: Enabled reflects whether the engine was deliberately stopped (see §11.8); Connected means something different per engine (bus/PROFINET layer up for EtherCAT/PROFINET, a scanner connection present for the Adapter) — it summarizes the whole engine, not any one target. Per-target status lives on TargetSummary.State instead.

GetTargetsAsync returns the engine's device/slave/scanner-target list normalized into TargetSummary — Name is what you pass as the target argument to the tag methods below. EtherNet/IP Adapter and PROFINET Device have no real target list (one fixed local IO set), so GetTargetsAsync returns a single synthetic row for them instead.

11.3 Writing tags

```
// "IM1" is a target Name from GetTargetsAsync (§11.2). On EtherNet/IP Adapter
// and PROFINET Device it is accepted but ignored — there is only one target.
await client.WriteTagAsync("IM1", "Motor_Run", true);

// Write several output tags on one target in a single round trip
await client.WriteTagsAsync("IM1", new Dictionary<string, object?>
{
    ["Motor_Run"] = true,
    ["Speed_Setpoint"] = 1500,
});
```

WriteTagAsync writes one tag; it is a thin wrapper over WriteTagsAsync with a single entry. Prefer WriteTagsAsync when setting more than one tag on the same target — one request instead of several means both less network traffic and no risk of the values landing in different engine cycles. Only Output-direction tags may be written, exactly like the web interface's own rule (§5.3) — writing an Input tag, or one that does not exist for that target, throws RasProLinkClientException (§11.6).

11.4 Live updates — polling

```
client.StatusChanged += (s, status) => Console.WriteLine($"status: {status}");
client.TargetsChanged += (s, list) => Console.WriteLine($"{list.Count} targets");
client.PollError += (s, ex) => Console.WriteLine($"poll error: {ex.Message}");

client.PollIntervalMs = 500; // default 1000
await client.StartPollingAsync();
// ...
client.StopPolling();
```

The service exposes only REST, no push channel, so the client library polls: StartPollingAsync runs a background loop that calls RefreshStatusAsync and GetTargetsAsync on your behalf every PollIntervalMs, raising StatusChanged and TargetsChanged after each round — the same information the web interface itself polls for. A failed round (service briefly unreachable) raises PollError instead of stopping the loop; polling keeps retrying on the next tick. Tag values are not part of polling — call ReadTagsAsync yourself, on whatever schedule your tags need, since which tags matter and how often is application-specific.

11.5 Tags from code

```
var defs = await client.GetTagDefinitionsAsync("IM1");

await client.AddOrUpdateTagAsync("IM1", new TagDefinition
{
    Name = "Sensor_Input1",
    Direction = IoDirection.Input,
    Type = TagDataType.Bool,
    Offset = 0,
    BitOffset = 0
});

await client.RemoveTagAsync("IM1", "Obsolete_Tag");

// One round trip fills a local table for GetTag/ReadTag<T> below
var values = await client.ReadTagsAsync("IM1");
```

```
bool running = client.ReadTag<bool>("IM1", "Motor_Run");
TagValue? raw = client.GetTag("IM1", "Sensor_Input1");
Console.WriteLine(raw?.Text);
```

GetTagDefinitionsAsync, AddOrUpdateTagAsync, SetTagDefinitionsAsync (replaces a target's whole tag table in one call) and RemoveTagAsync mirror the tag editor in the web interface (§5.3) — Slot/Subslot only matter for the PROFINET engines and are left at their defaults (0/1) elsewhere.

ReadTagsAsync is the only call that reaches the network for values; it fills a local table keyed by target and tag name. GetTag and the generic ReadTag<T> then read from that local table — they never make a request, so call ReadTagsAsync again whenever you need a fresher value. ReadTag<T> converts TagValue's stored number/text/raw bytes to bool, string, byte[] or a numeric type; for Int64/UInt64 tags read TagValue.Raw directly instead, since a double cannot hold the full 64-bit range without loss.

11.6 Error handling

Every failed call throws RasProLinkClientException. When the service replies with a structured error body, its message is the real reason; when a target name passed to a tag method cannot be resolved (EtherCAT Master / EtherNet/IP Scanner only — see §11.2), the client reports that directly without a round trip to the server; when the service cannot be reached at all, the message says so and wraps the underlying transport exception.

11.7 Without the library — plain HTTP

The library is a thin wrapper — every operation is a plain HTTP call, reachable from any language. Full endpoint list in Appendix A; here is the same "list devices" call in curl and Python for a PROFINET Controller — the underlying path differs for the other four engines exactly as summarized at the start of this chapter.

```
curl http://localhost:8080/api/devices
```

```
import requests
devices = requests.get("http://localhost:8080/api/devices").json()
for d in devices:
    print(d["name"], d["ipAddress"], d["state"])
```

The examples folder shipped with RasProLink has a full 16-channel walking-bit test client in five languages — C#, VB.NET, Python, PowerShell and Node.js — each self-contained with its own copy of the client library, ready to build and run standalone.

11.8 Starting and stopping the engine

```
await client.StopAsync(); // same effect as the web interface's Stop control – halts ALL targets
// ...
await client.StartAsync(); // resumes it
```

StopAsync and StartAsync are the client-side equivalent of the Stop/Start control shown on the active engine's own status page in the web interface — they halt or resume the entire active engine, all of its targets at once, not a single tag or target. After StopAsync, RefreshStatusAsync's EngineStatus.Enabled becomes false;

Connected drops too, since nothing is exchanging cyclic data any more. There is no client method to stop, disable or remove a single target — that level of detail is configured through each engine's own device/target editor in the web interface, not through this library.

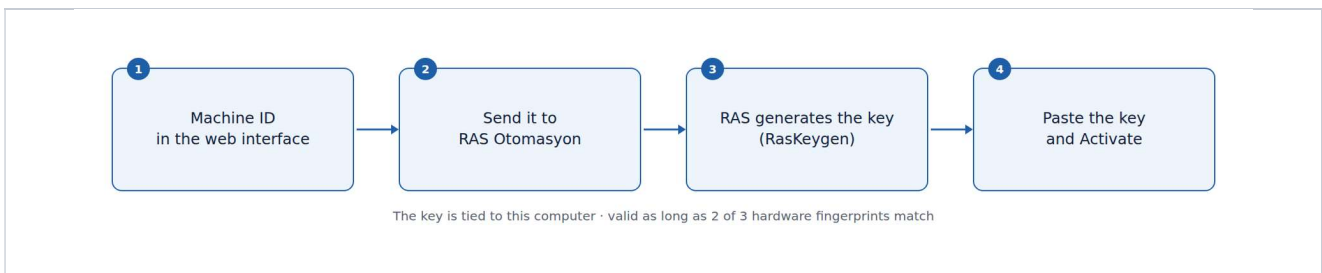
Chapter 12

Licensing

12.1 Trial period

RasProLink runs fully featured, on any of the five engines, for 30 days from first start with no license entered — every engine, every device, unlimited. Once the trial ends without a key, the active protocol layer stops running (devices disconnect) but the web interface keeps working so a key can still be pasted in.

12.2 Obtaining and activating a license



The key is tied to this computer — valid as long as 2 of 3 hardware fingerprints still match

10. Open the License card (§4.2) and copy the Machine ID.
11. Send it to RAS Automation together with your order/customer reference.
12. RAS Automation issues a key signed for that Machine ID (via RasKeygen — an internal tool that never leaves RAS Automation).
13. Paste the key into the License card and click Activate.

A license can also be removed again, from the same License card or via `DELETE /api/license` — this deletes the stored key and reverts the service to trial status (§12.1). It exists for support and hand-off cases (e.g. transferring the PC to someone else) and is not needed for routine activation.

12.3 What the license is tied to

A key is a fingerprint over several independent hardware/OS identifiers of the specific PC. It stays valid as long as 2 of 3 of those sources still match — tolerant of a routine change (such as an OS reinstall alone) without being transferable to another machine wholesale.

One license activates every engine on that PC — there is no per-protocol license: activating once on the Machine ID covers EtherCAT Master, EtherNet/IP Scanner, EtherNet/IP Adapter, PROFINET Controller and PROFINET Device alike, whichever engine is switched to active.

12.4 License error messages

Message	Meaning
Key does not belong to this machine	The key was produced for a different Machine ID — double-check what was sent when requesting it.

Message	Meaning
License file could not be written	The service account cannot write to its Config folder — check folder permissions.
Empty key	Nothing was pasted into the field before clicking Activate.

Chapter 13

Performance and Real-Time Behaviour

13.1 Why it matters

Windows is not a real-time operating system: a naive cyclic loop built on a generic timer can drift several milliseconds from its intended period under normal desktop scheduling, which is often larger than the cycle time itself on a fast fieldbus. RasProLink's five engines share a common set of techniques, applied identically to every one of them, to bring that drift down to a fraction of the cycle time on ordinary PC hardware — no real-time OS patch or dedicated hardware timer card required.

13.2 The four techniques

- Dedicated high-priority thread — the cyclic exchange runs on its own Highest-priority thread instead of sharing the thread pool with everything else the process does.
- Windows multimedia-timer resolution — the system-wide scheduler tick is temporarily raised to 1 ms (from the default ≈ 15.6 ms) for the lifetime of the cyclic loop, so any coarse wait actually wakes up close to when it should.
- CPU core-affinity pinning — the cyclic thread is pinned to the last logical CPU core, so the OS never migrates it mid-cycle; each migration resets cache/TLB state and can add hundreds of microseconds of jitter right when timing matters most.
- Zero per-cycle heap allocation — IO buffers and device lists are allocated once and reused every cycle (cleared and rewritten in place), removing garbage-collector pauses from the timing-critical path.

13.3 What to expect

On a typical Windows PC with these four techniques applied, EtherCAT Master jitter has been measured around 0.3 ms maximum at a 4 ms cycle — down from 9–10 ms before tuning. Results depend on the specific PC (CPU generation, core count, power plan) and on what else is running, but the same four techniques apply identically across all five engines, so comparable improvement is expected on PROFINET and EtherNet/IP cycles too.

13.4 Reading jitter in the interface

Cycle jitter, shown in each device's statistics (§9.4 for PROFINET Controller; the same field exists in every engine's device detail), measures how far a device's actual cyclic timing has drifted from the configured update time. As jitter approaches the configured watchdog/timeout, an occasional dropped connection becomes more likely — the fix is usually a longer update time on that device (more slack per cycle) or reducing congestion on the field-side network segment (a hub-like shared switch serving fewer devices, or a move to a properly QoS'd switch).

Chapter 14

General Troubleshooting

Protocol-specific issues are covered in each engine's own chapter (§6.5, §7.4, §8.4, §9.9, §10.5). The items below are shared across all five engines.

Symptom	Check
Web interface unreachable from another PC	Windows Firewall blocking inbound TCP 8080 — allow it (§3.3).
Engine switch seems to hang	The service restarts internally on a switch — allow a few seconds, then refresh; check services.msc if it takes longer than about 30 s.
License says trial expired right after a clean reinstall	uninstall.bat was run with -RemoveConfig, deleting the trial counter along with everything else — contact RAS Automation for a license instead of repeating the trial.
Need to attach diagnostics to a support request	Use the Service info card's paths (§9.7) to attach the current log file and devices.json.

Appendix A

HTTP API Reference

Every endpoint below is a plain JSON REST call under the engine's base URL (`http://<pc>:8080` by default), with camelCase field names and no built-in authentication — the service is designed to run on a trusted local network. If it must be reachable from an untrusted network, put it behind a reverse proxy that adds authentication. The endpoint set is identical across restarts and engine switches; only the Devices section differs per engine, reflecting each protocol's own device model.

A.1 Shared endpoints — all five engines

Endpoint	Purpose
GET /api/status	Engine identity, active protocol, and summary counters.
GET /api/host/info	Product name, build version, which engine is active and whether it is production-ready, service start time — independent of which engine is running (§4).
GET /api/engine POST /api/engine	List all five engines with their production-readiness, or switch the active one; a switch saves the choice to enginesettings.json and restarts the service into the new engine within a few seconds (§4).
GET/POST /api/settings/language	Read or set the UI/log language (tr/en/de); present identically on every engine and kept in sync with the setting saved in that engine's own config file.
GET /api/nics	List network adapters, show which is selected.
POST /api/nics	Select a network adapter (by MAC, optionally validated).
GET /api/health	Simple liveness check — {"status":"ok"}.
GET/POST/DELETE /api/license	Read license status, activate a key, or remove it. One license covers all five engines (§12.3).
GET /api/logs?after={id}	Service log records newer than the given id.
GET /api/paths	Configuration and log file paths on disk (§B).
GET /api/devices/{name}/tags PUT /api/devices/{name}/tags POST /api/devices/{name}/tags DELETE /api/devices/{name}/tags/{tagName} GET /api/devices/{name}/tags/values POST /api/devices/{name}/tags/write	List / replace / add-or-update / remove a tag; read current values; write one or more Output-direction tags (§5.3).

Note on /api/logs and /api/paths above: only PROFINET Controller and PROFINET Device expose them at those bare paths. EtherCAT Master and both EtherNet/IP roles expose the same information under an engine-prefixed route instead — /api/ethercat/logs and /api/ethercat/paths on EtherCAT Master (§A.2), /api/ethernetip/logs and /api/ethernetip/paths on both EtherNet/IP Scanner and Adapter (§A.3, §A.4). PROFINET Device additionally has no /api/paths endpoint at all — only /api/logs.

A.2 EtherCAT Master — devices

Every route in this section lives under the `/api/ethercat/` prefix — the EtherCAT engine does not use the generic `/api/devices` shape.

Endpoint	Purpose
GET <code>/api/ethercat/status</code> GET <code>/api/ethercat/bus</code>	Engine status summary (running/stopped, message), or the full bus snapshot (NIC, slaves, cycle time).
POST <code>/api/ethercat/start</code> POST <code>/api/ethercat/stop</code>	Manually start or stop the EtherCAT bus/slave communication — the web interface asks for confirmation before stopping (§6).
GET <code>/api/ethercat/nics</code> GET <code>/api/ethercat/nic</code> POST <code>/api/ethercat/nic</code>	List available network adapters; read or select which one the ring uses (by name, with an optional MAC override) (§6).
POST <code>/api/ethercat/bus/settings</code>	Set the cyclic exchange interval (cycle time, milliseconds) (§6).
GET <code>/api/ethercat/slaves</code> PUT <code>/api/ethercat/slaves</code> DELETE <code>/api/ethercat/slaves/{id}</code>	List the configured slaves, add-or-update one (upsert), or remove one from the configuration.
GET <code>/api/ethercat/slaves/status</code>	Live per-slave status: ESM state and statistics for every configured slave.
GET <code>/api/ethercat/slaves/{id}/input</code> POST <code>/api/ethercat/slaves/{id}/output</code>	Read the slave's current raw input PDO bytes (hex), or write its output PDO bytes.
GET <code>/api/ethercat/slaves/{id}/tags</code> PUT <code>/api/ethercat/slaves/{id}/tags</code> POST <code>/api/ethercat/slaves/{id}/tags</code> DELETE <code>/api/ethercat/slaves/{id}/tags/{tagName}</code> PUT <code>/api/ethercat/slaves/{id}/tags/{tagName}</code> GET <code>/api/ethercat/slaves/{id}/tags/values</code> POST <code>/api/ethercat/slaves/{id}/tags/write</code>	List / replace / add-or-update a slave's tags; remove or rename-and-update one tag; read current values; write one or more Output-direction tags (§5.3).
POST <code>/api/ethercat/slaves/{id}/sdo/read</code> POST <code>/api/ethercat/slaves/{id}/sdo/write</code>	Read or write a CoE object dictionary entry (index/sub-index, hex payload) — the SDO tool used for startup parameters and diagnostics (§6).
GET <code>/api/ethercat/esi-catalog</code> POST <code>/api/ethercat/esi-catalog/reload</code> POST <code>/api/ethercat/esi-catalog/directory</code> POST <code>/api/ethercat/esi-catalog/browse-folder</code>	Report the watched ESI folder, indexed signature count and last reload time; force a re-scan; change which folder is watched; open a folder picker (interactive desktop sessions only) — the ESI chain builder (§6).
GET <code>/api/ethercat/logs</code> GET <code>/api/ethercat/paths</code>	Engine-prefixed equivalents of the shared <code>/api/logs</code> and <code>/api/paths</code> described in §A.1, including the ESI catalog directory.

A.3 EtherNet/IP Scanner — devices

Every route in this section lives under the `/api/ethernetip/scanner/` prefix (Target management, connections, IO) or `/api/ethernetip/` (discovery, EDS catalog) — the Scanner engine does not use the generic `/api/devices` shape.

Endpoint	Purpose
GET /api/ethernetip/status	Role identity and status message ("EtherNet/IP Scanner", running).
GET /api/ethernetip/scanner/targets PUT /api/ethernetip/scanner/targets DELETE /api/ethernetip/scanner/targets/{id}	List configured Targets; add-or-update one (IP, T→O/O→T assembly instances and sizes, RPI — §7.2); remove one, which sends Forward_Close first.
GET /api/ethernetip/scanner/connections GET /api/ethernetip/scanner/engine-status	Per-Target connection state, RPI, assembly sizes and statistics; or the whole engine's enabled/disabled flag (separate from any single Target's connection).
POST /api/ethernetip/scanner/stop POST /api/ethernetip/scanner/start	Manually stop or start communication with every configured Target.
GET /api/ethernetip/scanner/targets/{id}/input POST /api/ethernetip/scanner/targets/{id}/output	Read the Target's current O→T assembly bytes (hex), or write the T→O assembly bytes.
POST /api/ethernetip/scanner/discover?seconds={n}	Broadcast a UDP ListIdentity discovery for 0.5-10 s; each device found is matched against the EDS catalog to pre-fill a "add as Target" form when possible.
GET /api/ethernetip/eds-catalog POST /api/ethernetip/eds-catalog/reload POST /api/ethernetip/eds-catalog/directory POST /api/ethernetip/eds-catalog/browse-folder POST /api/ethernetip/eds/parse	Catalog folder status; force a re-scan; change which folder is watched; open a folder picker (interactive desktop sessions only); or parse a single uploaded .eds file's assemblies directly (no catalog folder involved).
GET /api/ethernetip/scanner/targets/{id}/tags PUT /api/ethernetip/scanner/targets/{id}/tags POST /api/ethernetip/scanner/targets/{id}/tags DELETE /api/ethernetip/scanner/targets/{id}/tags/{tagName} PUT /api/ethernetip/scanner/targets/{id}/tags/{tagName} GET /api/ethernetip/scanner/targets/{id}/tags/values POST /api/ethernetip/scanner/targets/{id}/tags/write	List / replace / add-or-update a Target's tags; remove or rename-and-update one tag; read current values; write one or more Output-direction tags (§5.3).

A.4 EtherNet/IP Adapter — assemblies

Every route in this section lives under the /api/ethernetip/adapter/ prefix. Unlike the Scanner, the Adapter has no endpoint that lists connected Scanners — there is exactly one Input/Output/Config assembly set, so its own status (running/stopped, last activity) is read from /api/ethernetip/adapter/status instead.

Endpoint	Purpose
GET /api/ethernetip/status GET /api/ethernetip/adapter/status	Role identity/status message, or Adapter-specific status (whether a Scanner is currently connected, last activity).
GET /api/ethernetip/adapter/config PUT /api/ethernetip/adapter/config	Read, or set, the local Input/Output/Config assembly instances and sizes this engine exposes to a Scanner (§8.2).
POST /api/ethernetip/adapter/stop POST /api/ethernetip/adapter/start	Manually stop or start the TCP/UDP listeners that accept a Scanner's connection.

Endpoint	Purpose
GET /api/ethernetip/adapter/output POST /api/ethernetip/adapter/input	Read the Output assembly bytes most recently received from a Scanner, or write the Input assembly bytes (bench-test without a real Scanner attached).
GET /api/ethernetip/adapter/tags PUT /api/ethernetip/adapter/tags POST /api/ethernetip/adapter/tags DELETE /api/ethernetip/adapter/tags/{tagName} PUT /api/ethernetip/adapter/tags/{tagName} GET /api/ethernetip/adapter/tags/values POST /api/ethernetip/adapter/tags/write	List / replace / add-or-update this Adapter's (single) set of tags; remove or rename-and-update one tag; read current values; write one or more Output-direction (T→O) tags (§5.3).

A.5 PROFINET Controller — devices

Endpoint	Purpose
GET /api/devices	List all IO-Devices with summary status.
GET /api/devices/{name}	Full detail: slots, subslots, statistics.
POST /api/stop POST /api/start	Manually stop or start the PROFINET protocol layer for every IO-Device at once — the web interface asks for confirmation before stopping (§9.6).
POST /api/ip	Apply the Controller's own station name and IP/subnet/gateway settings; restarts the PROFINET layer to take effect.
GET /api/devices/{name}/io POST /api/devices/{name}/io	Current raw IO bytes per submodule (hex); write to an output submodule.
POST /api/devices/{name}/enabled	Enable or disable a device.
DELETE /api/devices/{name}	Remove a device from the configuration.
POST /api/devices/xdm	Import a .xdm device proxy file (multipart/form-data) — adds every device it contains in one step (§9.3).
POST /api/devices/discover	DCP network scan — lists every PROFINET station found, each with its automatic GSDML catalog match if any (§9.10).
GET /api/gsdml-catalog GET /api/gsdml-catalog/devices GET /api/gsdml-catalog/devices/{vendorId}/{deviceId}/{dapId}/modules	Catalog folder status (file/device counts, last scan, errors); list every DAP in the catalog; list a DAP's plug-in modules (§9.10).
POST /api/gsdml-catalog/reload POST /api/gsdml-catalog/directory POST /api/gsdml-catalog/browse-folder	Rescan the catalog folder now; change which folder is scanned; open a folder picker (interactive desktop sessions only) (§9.10).
POST /api/devices/gsdml	Build and add a new device directly from a GSDML catalog entry (DAP + slot/module assignments) — no .xdm involved (§9.10).

A.6 PROFINET Device — own configuration

Endpoint	Purpose
GET /api/device GET /api/slots	This Device's own identity (station name, Vendor/Device ID) and current slot/subslot layout.
GET /api/modules POST /api/modules	Read or set this Device's own slot/subslot module layout — must match what is described to the IO-Controller (via GSDML) and what is physically simulated here (§10.3).
GET /api/io POST /api/io	Read the Device's current input bytes as seen by the Controller, or write them (simulating a sensor); Output bytes arrive from the Controller and are read-only here. The GET response also carries the current AR phase and the connected Controller's station name, if any — there is no separate /api/ar endpoint.
GET /api/diagnostics POST /api/diagnostics/test-pattern POST/DELETE /api/diagnostics/force DELETE /api/diagnostics/forces POST /api/diagnostics/provider-status POST /api/diagnostics/reset-statistics	Read the Diagnostics card's state; drive a repeating test pattern on a slot; force (or clear) a single byte range or a tag's simulated value; clear every active force at once; simulate a module fault (IOPS = Bad) or clear it; reset the frame/error counters (§10).

Appendix B

Configuration Files

Every engine keeps its configuration under its own Config\ folder next to the service executable, as plain, human-readable JSON. Files are normally edited from the web interface; editing them by hand while the service is stopped also works, but a running service must be restarted to pick up manual changes.

B.1 Config\enginesettings.json

A small host-level file that holds exactly one thing: which of the five engines is active. Written when you switch engines from the top bar or the engine picker (§4); the service reads it once at startup and restarts itself into the newly chosen engine within a few seconds of a switch (§4). It does NOT hold NIC selection, station name, IP/subnet/gateway or any other per-engine setting — those live in each engine's own settings file, listed in §B.2 below.

B.2 Per-engine settings and device/Target roster

Each engine keeps its own identity/network settings and, where it manages more than one remote station, its device/slave/Target roster, under a filename specific to that engine — not a shared generic name. For engines with several stations, the roster carries each one's address and I/O shape (PROFINET Controller devices are normally populated by importing a .xdb file, §9.3, and carry station name, IP and full slot/subslot layout; EtherCAT Master slaves carry their ESI-matched PDO map, §6.2; EtherNet/IP Scanner Targets carry IP, assembly instances/sizes and RPI, §7.2). PROFINET Device and EtherNet/IP Adapter represent a single station, so they have no separate roster file — their one settings file covers both identity and I/O shape.

Engine	Settings file	Roster (multi-station only)
PROFINET Controller	Config\controllersettings.json	Config\devices.json
PROFINET Device	Config\deviceconfig.json	— (single station)
EtherNet/IP Scanner	Config\ethernetip-scanner-targets.json (also holds the Target roster)	same file
EtherNet/IP Adapter	Config\ethernetip-adapter-config.json	— (single station)
EtherCAT Master	Config\ethercat-master-config.json (also holds the slave roster)	same file

Edited from the top bar and each engine's own settings/devices cards; the service must be restarted to apply a change made by hand while it is stopped.

B.3 Per-engine tags file

Tag definitions (§5.3) also live in a filename specific to the active engine — a dictionary shaped {"scope": [tags], ...}, where "scope" is a device/slave/Target name for the multi-station engines or a single fixed key for PROFINET Device and EtherNet/IP Adapter. Edited from the web interface's Tags card or the .NET client's tag methods; removing a device/Target from a multi-station engine's roster automatically removes its tags from here too.

Engine	Tags file
PROFINET Controller	Config\tags.json
PROFINET Device	Config\tags.json
EtherNet/IP Scanner	Config\ethernetip-scanner-tags.json
EtherNet/IP Adapter	Config\ethernetip-adapter-tags.json
EtherCAT Master	Config\ethercat-master-tags.json

PROFINET Controller and PROFINET Device happen to use the identical filename (Config\tags.json) for an unrelated reason — only one engine ever runs at a time, so the two files never coexist.

B.4 Other files

Config\lisans.json holds the activated license (removed by uninstall.bat -RemoveConfig, or by removing the license from the web interface). It is the same file format and location convention on every engine, since one license covers all five (§12.3). Note that the filename is lisans.json, not license.json. The logs\ folder holds rolling daily log files with the same content as the Service Log card's live feed. Engines with a device catalog also keep it in its own subfolder — Config\profinet-controller-gsdml\ (GSDML), Config\ethercat-master-esi\ (ESI), Config\ethernetip-scanner-eds\ (EDS) — reported by that engine's /api/paths (§A).

Appendix C

Glossary

C.1 PROFINET

Term	Definition
AR	Application Relation — the logical connection between a Controller and an IO-Device.
IOCR	IO Communication Relation — the cyclic data channel set up as part of an AR.
DCP	Discovery and Configuration Protocol — used to find a device by station name and, if needed, assign it a name/IP.
RPC	PROFINET's real connection-management protocol, over UDP/IP (port 34964) — used for Connect, Write/PrmEnd, and Release.
DAP	Device Access Point — the DeviceAccessPointItem in a device's GSDML, representing the device itself. It is always Slot 0, is fixed (never removed or swapped), and its GSDML-declared plug-in modules populate the device's other slots (§9.10).
IOPS / IOCS	IO Provider Status / IO Consumer Status — one status byte attached to each submodule's cyclic data in each direction: IOPS says whether the data the producer just sent is good or bad, IOCS says whether the consumer is actually using it. The Device diagnostics card's "simulate a module fault" forces a submodule's IOPS to Bad (§10).
PrmEnd	The RPC call that ends parameterization and tells the device to start applying its configuration.
GSDML	General Station Description Markup Language — the XML file a device's manufacturer publishes describing its capabilities, modules, and parameters.
XDB	Device proxy file — TIA Portal's compiled export of a Controller PC station's project configuration; the Controller reads it directly instead of a device's raw GSDML (§9.5).
IO-Controller	The Master station — this product, running the PROFINET Controller engine.
IO-Device	A remote station with modular I/O — either real field hardware, or this product running the PROFINET Device engine.
Watchdog	The time a Controller waits for a cyclic frame before declaring an AR lost; also called the update-time / timeout multiplier.

C.2 EtherCAT

Term	Definition
ESI	EtherCAT Slave Information — the XML file a slave's manufacturer publishes describing its PDOs, parameters and supported modes; matched automatically against each slave's SII signature (§6.2).
SII	Slave Information Interface — a small EEPROM every slave carries, reporting its Vendor ID, Product Code and Revision over the ring itself.
PDO	Process Data Object — the cyclic input/output bit/byte layout a slave exchanges every cycle; read from the matched ESI entry.
SDO	Service Data Object — an acyclic mailbox read/write to a slave's CoE object dictionary, used for startup parameters and diagnostics.

Term	Definition
ESM	EtherCAT State Machine — the INIT → PREOP → SAFEOP → OP sequence every slave walks through before cyclic data starts flowing.
SM	Sync Manager — the slave-side buffer mechanism that hands off cyclic and mailbox data between the Master's frame and the slave's application.
FMMU	Fieldbus Memory Management Unit — maps a slave's local physical memory into the Master's single logical process image.
CoE	CANopen over EtherCAT — the object-dictionary convention EtherCAT reuses from CANopen for SDO parameters and diagnostics.

C.3 EtherNet/IP

Term	Definition
CIP	Common Industrial Protocol — the object-oriented messaging model EtherNet/IP carries over TCP (explicit messages) and UDP (cyclic I/O).
Forward Open	The CIP service that establishes a Class 1 (cyclic) connection between a Scanner and a Target, negotiating the RPI and assembly sizes.
Forward_Close	The CIP service that tears down a Class 1 connection. This engine sends it whenever a Target's cyclic exchange loop ends — including when a Target is deleted from the Scanner's roster, right before it is removed (§A.3).
Class 1 / Class 3	CIP connection types by purpose: Class 1 is the cyclic, connected UDP traffic a Forward Open sets up for I/O assemblies; Class 3 is connected, request/response TCP messaging (explicit messages) used for one-off reads/writes such as parsing an EDS or reading device identity.
RPI	Requested Packet Interval — the cyclic update rate a Scanner asks a Target to run at.
Originator / Scanner	The station that opens the connection and drives the exchange — this product's EtherNet/IP Scanner engine, typically a PLC in a real system.
Target / Adapter	The station that accepts a Forward Open and exposes assemblies — this product's EtherNet/IP Adapter engine, typically a field device in a real system.
Assembly	A named group of bytes CIP exchanges as one unit — Input, Output or Config, each with its own instance number and size.
ListIdentity	The CIP/UDP broadcast discovery command (port 44818) — the EtherNet/IP equivalent of PROFINET's DCP Identify. The Scanner's discovery feature broadcasts it and reads back every replying device's Vendor ID, Product Code, Device Type and Revision, then matches that identity against the EDS catalog (§A.3).
EDS	Electronic Data Sheet — the file format a Target's manufacturer publishes describing its assemblies and parameters (the EtherNet/IP equivalent of GSDML/ESI).

C.4 Shared terms

Term	Definition
Engine	One of the five protocol implementations this product ships — EtherCAT Master, EtherNet/IP Scanner, EtherNet/IP Adapter, PROFINET Controller, PROFINET Device. Only one runs at a time.
Tag	A name given to a byte/bit range inside a device's raw IO data, so it can be read or written by name from the web interface or the .NET client instead of by raw offset (§5.3).
Cycle time	The interval at which an engine drives its cyclic exchange — configurable per engine, typically single-digit milliseconds.
Jitter	How far a cycle's actual timing drifts from the configured cycle time/RPI/update time; kept in the sub-millisecond range by the real-time tuning described in Chapter 13.
Machine ID	The identifier shown on the License card and used to activate a key on this PC. It is derived from two-of-three independent hardware/OS sources (SMBIOS system UUID, Windows installation GUID, system-drive volume serial), each salted and hashed so the raw values never leave the machine; a license stays valid as long as at least two of the three still match, so routine maintenance (disk swap, NIC change, OS reinstall) does not silently break it (§12.3).
Trial period	The 30-day evaluation window before a license key is required. Its start date is recorded redundantly in three locations (the install folder, %ProgramData%, and the registry), each entry HMAC-sealed to this PC's Machine ID so it cannot be hand-edited to extend it; the oldest recorded start and the most advanced "last seen" time across all three win on read, and a backward system-clock jump is treated as an expired trial (§12.3).