

ZEUS AUTOMATION

Zeus PLC IDE

Operation Manual

Configuration, programming, commissioning and maintenance of the Zeus PLC
IPC-CM400200 with the native Zeus runtime

Software version 0.1.0.13

Edition 2.10 · 2 October 2026

Web optizeus.org

About this manual

This manual describes how to configure, program, commission and maintain a Zeus PLC IPC-CM400200 controller running the **native Zeus runtime**, using the **Zeus PLC IDE**. It covers the IDE from installation to online debugging, the Structured Text, Ladder and SFC languages, the IO systems (Modbus TCP, EtherCAT, Profinet, IO-Link), the integration services (OPC UA, EtherNet/IP, Modbus TCP slave, operator screens), and the complete reference of the built-in functions, standard function blocks and function-block library.

Who should read it

Automation and controls engineers who configure and program the controller, and commissioning and maintenance staff who work with a running machine. It assumes you know PLC fundamentals — scan cycles, IO, IEC 61131-3 programming — but not Zeus.

How the manual is organised

Part	Chapters	Read it when
I Getting started	1–3	You are new to Zeus. Chapter 3 is a complete first project.
II The IDE	4–6	You need to find your way around the IDE, projects and controllers.
III Hardware	7–12	You configure IO devices, addresses and IO-Link.
IV Programming	13–19	You write programs. Chapter 14 is the language reference.
V Running and debugging	20–26	You download, monitor, force, trace and debug.
VI Integration and maintenance	27–37	You connect SCADA/HMI, configure alarms, secure and maintain the system.
Appendices	A–H	You need a reference: functions, blocks, library, limits, keys, glossary, examples, configuration.

Conventions

Convention	Meaning
Bold	A menu item, button, tree node or field exactly as the IDE labels it
Monospace	Code, variable names, file names, addresses, keys typed
Project → Save As	A menu path
Ctrl+F7	A keyboard shortcut
%IX0.0	A direct address (Chapter 11)

Notes in the text:

Note: Information that helps you understand or use a feature.

Tip: A faster or better way to do something.

Important: Information you need to avoid a problem — lost work, a failed download, an unexpected result.

WARNING: A risk to people or to the machine. Read it before acting.

Safety

WARNING: A controller drives a machine. Downloading a program, writing a value, forcing a point, halting at a breakpoint or changing the run state can move actuators immediately. Before any of these on a machine that can move, make sure the machine is in a safe state and that the people near it know what you are about to do.

WARNING: The controller and its software are not a safety system. Emergency stops, guard interlocks, safe torque off and other safety functions must be implemented with hardware and components rated for the purpose, independent of the PLC program.

WARNING: A force holds a point regardless of the program, including the program's interlocks. Release every force before leaving a machine (the **FORCE** badge shows when any are active).

Software version

This edition describes Zeus PLC IDE and runtime **version 0.1.0.13**. Screens and behaviour of other versions may differ.

Related documents

Document	Subject
Zeus-PLC-01	Architecture
Zeus-PLC-03	Timing model and real-time posture
Zeus-PLC-04	Configuration schema
Zeus-PLC-05	Online (monitor) protocol
Zeus-PLC-07 / 08	EtherCAT and EtherNet/IP bring-up
Zeus-PLC-09 / 10	Security and security requirements
Zeus-PLC-24	Controller redundancy (in development)
Zeus-PLC-35	IO-Link
Zeus PLC IDE brochure	Product overview

Trademarks and third-party software

EtherCAT® is a registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany. PROFINET® is a registered trademark of PROFIBUS & PROFINET International (PI); EtherNet/IP™ of ODVA, Inc.; Modbus® of Schneider Electric; OPC UA® of the OPC Foundation; Raspberry Pi® of Raspberry Pi Ltd. None of these companies is affiliated with Zeus Automation or endorses this product; the names are used only to say what the product exchanges data with.

The controller runtime is proprietary software of Zeus Automation / Biosoft Engineering Ltd. It includes the open-source SOEM 1.4.0 (EtherCAT master; GPL-2.0 with a linking exception) and open62541 1.4.6 (OPC UA; MPL-2.0), both unmodified, and uses the operating system's OpenSSL (Apache-2.0). Their complete source code ships with the product: in the IDE package under `licences\`, and on every controller in `/opt/zeus-plc/licences/ ( zeus-plc-third-party-source-<version>.tar.gz )`.

Revision history

Edition	Date	Change
1.0	18 September 2026	First issue.
1.1	19 September 2026	Batch layer (ISA-88), alarms at source, electronic signatures.
2.0	27 September 2026	Complete rewrite and expansion: quick-start tutorial, full Structured Text reference, data types (DUT), Ladder and SFC chapters, task configuration editor, online variable tables with write and force, IO-Link, EtherNet/IP tag server, generated references for all built-in functions, standard function blocks and the library, limits, and eight worked example projects.
2.1	29 September 2026	Version 0.1.0.4: operator screens v2 (dynamic properties and shapes, faceplates, input elements with keypad and actions, alarm table, panel log-in, text lists and languages, access per element, recorded trends, recipes, local HDMI display); per-device start-up wait for critical IO; 256 POU's per download.
2.2	29 September 2026	Version 0.1.0.5: built-in symbol library of 1,081 plant drawings in the operator-screen editor (Symbol library... dialog, colour with state); each screen tab keeps its own screen and selection; Deploy runtime refuses a controller binary older than its source.
2.3	30 September 2026	Version 0.1.0.6: screen editor right-click menu (cut, copy, paste, duplicate, undo/redo, drawing order, alignment), spin and impeller-pulse motion for impellers, fans and mixers, colour palette on every colour box, variable browser (searchable tree of globals, programs and members, offline from the project), Download screens in the editor; one OFFLINE badge in the top bar; controller master image guide.
2.4	30 September 2026	Version 0.1.0.7: Project → Create new project...; Replace this project (after an upload) restores data types, global variables and the library to their own folders and rebuilds the running program exactly; the force / write table opens at once; system tests SEC-13 and SEC-20 check the intended behaviour.

Edition	Date	Change
2.5	30 September 2026	Version 0.1.0.8: ladder editor multi-select, clipboard, undo/redo, find/replace, math/move/LIM boxes, nested branches, rung as text, descriptions, fold, print and online edit; View as ladder for ST programs, function blocks and methods (live, per instance) and Convert to ladder POU; inline online values in the ST editor; import of device descriptions (GSDML / ESI / EDS); two or more Profinet devices on one controller; Download all sends the application.
2.6	2 October 2026	Version 0.1.0.9: settings (OPC UA, screens server, Modbus TCP slave, EtherNet/IP tags) applied without stopping the controller; Download offers to send changed settings with the program; Disable / Enable for POU; Find and Replace in a POU and in the whole project; build errors shown in the code (underline and message) and every controller line mapped to its own file; online inline values that stay while you read and click; folders show RUNNING; Build All about twice as fast; fixes for Hebrew text in downloads, simulator restarts, and editor line numbers on long files.
2.7	2 October 2026	Version 0.1.0.10: trademark and third-party software notices; licensing and demo time (new Chapter 37); update notification and Check for updates (2.8.1); timers and counters show IN / Q / ET / PT live in the ST editor (22.6); the online value column no longer drifts into long lines.
2.8	2 October 2026	Version 0.1.0.11: Getting started tutorial (Project menu, offered on first start) and the worked examples of Appendix G shipped as nine ready projects (Open an example project... , 2.11, Appendix G).
2.9	2 October 2026	Version 0.1.0.12: the simulator drives a 3D process simulation such as Factory I/O over Modbus TCP on the same PC, with no time limit (26.6a, 37.1); example F1-FactoryIO-Conveyor.
2.10	3 October 2026	Version 0.1.0.13: uploading into a new project restores each program into the folder the controller's task names, so the project is IN SYNC at once; a copy an earlier upload put in the wrong folder is removed.

Contents

1. Product overview	18
1.1 What the system is	18
1.2 The controller: Zeus PLC IPC-CM400200	19
1.3 The two runtime options	19
1.4 What the native runtime provides	20
1.5 How a program gets to the controller	21
1.6 Protocols	22
1.7 Capacities at a glance	22
1.8 How Zeus PLC fits with OptiZeus SCADA	23
1.9 In development	24
1.10 Documentation set	24
2. Installation and starting	25
2.1 System requirements	25
2.2 Package contents	25
2.3 Installing	26
2.4 Starting and stopping the IDE	27
2.5 Desktop and Start-menu shortcuts	28
2.6 Ports used	29
2.7 Where the IDE writes files	29
2.8 Updating to a new version	30
2.9 Uninstalling	31
2.10 The IDE window at first start	31
2.11 First run of the example project	32
3. Quick start: your first project	34
3.1 What you will build	34
3.2 Before you begin	34
3.3 Step 1 — Create the project	35
3.4 Step 2 — Set the engineer password	35
3.5 Step 3 — Add the IO	35
3.6 Step 4 — Create the program	36
3.7 Step 5 — Build	38
3.8 Step 6 — Start the simulator and log in	38
3.9 Step 7 — Download	39
3.10 Step 8 — Go online and watch	39
3.11 Step 9 — Write a prepared value	40
3.12 Step 10 — Force inputs and run a fill	40
3.13 Step 11 — Make an online change	41
3.14 Step 12 — Add a Ladder program	42
3.15 Step 13 — Run both programs	43
3.16 Step 14 — Finish	44
3.17 Where to go next	45
4. The IDE window	47
4.1 Layout	47
4.2 The menu bar	47
4.3 Project menu	48
4.4 Debug menu	49

4.5 Online menu	49
4.6 Status badges and connection indicator	50
4.7 The tool strip	52
4.8 The device tree	52
4.9 Tabs and editors	56
4.10 The Messages panel	57
4.11 Dialogs	57
4.12 Keyboard summary	58
5. Projects, archives and libraries	59
5.1 What a project is	59
5.2 What is on disk	59
5.3 Creating, opening and switching projects	60
5.4 Saving	61
5.5 Save project as, and archives	62
5.6 Uploading from a controller	63
5.7 Saved controller data	63
5.8 Backups	64
5.9 The library	64
5.10 Folders in the POU tree	68
5.11 Renaming, excluding and deleting POUs	69
5.12 Working with version control	69
6. Controllers, connection and users	70
6.1 Targets	70
6.2 The Controller target dialog	70
6.3 Connecting and disconnecting	72
6.4 Network paths and securing the connection	72
6.5 Logging in	74
6.6 Roles: what each may do	75
6.7 Users & Roles	76
6.8 Initial passwords that must be changed	77
6.9 Checks when you go online	77
6.10 Controller Settings	78
6.11 Installing or updating the runtime (Deploy runtime...)	79
6.12 Checklist: first connection to a new controller	80
7. IO devices and scanning	82
7.1 Concepts	82
7.2 The IO Devices branch	82
7.3 Adding a device	83
7.4 The device page	85
7.5 Scan for RIO...	88
7.6 The Network Interfaces branch	90
7.7 Download hardware and download program	91
7.8 How the process image is built from the devices	92
7.9 Device states and diagnostics	93
7.10 What outputs and inputs do on STOP and FAULT	94
7.11 Checklist before a hardware download	96
8. Modbus TCP devices	97
8.1 Overview	97
8.2 Adding a Modbus TCP device	97
8.3 Blocks	98
8.4 Addresses produced	100

8.5 Byte order and word order	101
8.6 Polling and timing	103
8.7 Connection handling and failures	104
8.8 Diagnostics	105
9. EtherCAT	108
9.1 Status and scope	108
9.2 Preparing the controller	108
9.3 ESI files	109
9.4 Scanning a segment	110
9.5 Laying out a rack from an ESI, offline	112
9.6 The rack of a modular coupler	112
9.7 Start-up SDOs	115
9.8 Cycle time and distributed clocks	116
9.9 Bring-up procedure	117
9.10 Diagnostics and failure codes	118
9.11 Limits	120
10. Profinet	122
10.1 Status and scope	122
10.2 Concepts	122
10.3 Preparing	123
10.4 Discovering devices	124
10.5 Naming and addressing a device (commissioning)	125
10.6 Adding the device and its rack	125
10.7 Cyclic exchange and timing	128
10.8 Bring-up procedure	129
10.9 Troubleshooting	131
11. Addresses, aliases and signal ranges	132
11.1 The process image	132
11.2 Direct addresses	132
11.3 Channel aliases	134
11.4 Signal ranges (analogue scaling)	136
11.5 Inhibit and critical	138
11.6 Marker memory	138
11.7 Clock memory bits	138
11.8 Byte order	139
11.9 Summary of address limits	140
12. IO-Link	141
12.1 How IO-Link reaches a program	141
12.2 The rack	141
12.3 The IO-Link ports page	142
12.4 The zeus-iolink library	143
12.5 Process data	144
12.6 Parameters (ISDU)	144
12.7 Events	147
12.8 Bench checklist	149
12.9 Limits and what is not done	150
13. POU's and editors	152
13.1 POU kinds	152
13.2 The POU tree	152
13.3 Creating a POU	154

13.4 Folders	156
13.5 The POU context menu	156
13.6 Renaming, deleting, converting and disabling	157
13.7 The Structured Text editor	158
13.8 Methods	160
13.9 Variables of a POU	161
13.10 Editing aids	162
13.11 Build and Check	165
13.12 Keyboard reference for editing	166
14. Structured Text reference	168
14.1 How a program is compiled	168
14.2 Lexical elements	168
14.3 Literals	170
14.4 Elementary data types	172
14.5 Derived data types	174
14.6 Program organisation units	175
14.7 Variable declarations	177
14.8 Direct addresses	179
14.9 Expressions and operators	180
14.10 Assignment and type conversion	182
14.11 Statements	184
14.12 Calling function blocks	187
14.13 Functions	189
14.14 Methods	191
14.15 VAR_IN_OUT (by reference)	192
14.16 Arrays	193
14.17 Structures	194
14.18 Enumerations	195
14.19 Bit access	196
14.20 Strings	197
14.21 Built-in functions	198
14.22 Process-image byte access (PI_* functions)	199
14.23 Run-time behaviour and errors	200
14.24 Compiler diagnostics	201
14.25 Limits	201
14.26 Differences from other IEC 61131-3 implementations	202
15. Data types (DUT)	205
15.1 What a DUT is	205
15.2 The Data Types (DUT) node	205
15.3 Structures	207
15.4 How DUTs are linked	210
15.5 Enumerations	211
15.6 Worked example: a motor record, an array of motors and one FB per motor	
15.7 Aliases	216
15.8 Pasting DUT text from other IEC 61131-3 environments	216
15.9 Renaming and deleting a DUT	217
15.10 Viewing structures online	218
15.11 Retain	218
15.12 Limits	219
15.13 Summary of what is not supported	219
16. Ladder	220

16.1 How a ladder POU works	220
16.2 Creating a ladder POU	220
16.3 The editor window	221
16.4 Rungs and lines	222
16.5 Execution order	223
16.6 Parts	223
16.7 Selecting, copying and editing	227
16.8 Declarations	229
16.9 Worked example: a conveyor	230
16.10 Raw ST rungs and editing a rung as ST	232
16.11 Online: power flow and live values	233
16.12 A second example: counting, comparing, data blocks and shared coils	234
16.13 A ladder function block	236
16.14 Tips and limitations	236
17. Sequential Function Charts	238
17.1 How SFC is implemented	238
17.2 Creating a chart	238
17.3 Syntax	239
17.4 Action qualifiers	240
17.5 Step flags	241
17.6 Branches	241
17.7 How a chart runs in each scan	242
17.8 The Chart view	242
17.9 Worked example: a mixing sequence	244
17.10 Watching and debugging a chart	247
17.11 Limits	247
17.12 Summary of differences from IEC 61131-3 graphical SFC	248
18. Task configuration	249
18.1 The execution model in one page	249
18.2 Where the task settings are	249
18.3 MainTask settings	250
18.4 The program call list	251
18.5 Tasks and task groups	252
18.6 Programs at different rates	253
18.7 Cores and service threads	256
18.8 Scan-time monitoring	257
18.9 Overruns and the overrun policy	257
18.10 Fixed limits (Stack Usage page)	258
18.11 Real-time posture	258
18.12 Checklist	259
19. Global variables and libraries	260
19.1 How shared declarations reach every POU	260
19.2 Global Variables (<code>globals.st</code>)	260
19.3 Further global variable lists (<code>gv1/</code>)	
19.4 RETAIN belongs in <code>globals.st</code>	262
19.5 The function-block library	263
19.6 Writing your own library	266
19.7 Methods	269
19.8 Summary	270
20. Build, download and online change	272
20.1 Overview	272

20.2 Check, Build and Build All	272
20.3 What is linked into one unit	274
20.4 The download commands	275
20.5 Which program is downloaded	278
20.6 Online change	279
20.7 Upload from controller	281
20.8 The boot application	282
20.9 Is the controller running this project?	282
20.10 Procedure: first download to a new controller	284
20.11 Procedure: online change on a running machine	285
20.12 Messages you may see	285
21. Run states and faults	286
21.1 The scan	286
21.2 The run states	286
21.3 Commands	288
21.4 Faults	290
21.5 Output behaviour on stop and fault	293
21.6 Starting up: the boot application	294
21.7 Panel lamps	294
21.8 Examples	295
21.9 Procedure: recovering from a FAULT	296
22. Online monitoring, write and force	297
22.1 Where live values come from	297
22.2 Going online: Monitor	297
22.3 The online variable tables	298
22.4 Prepared values	300
22.5 Write, force and release	300
22.6 Live values in the Structured Text editor	302
22.7 Live power flow in Ladder	303
22.8 The Force / write table	304
22.9 Who may write and force	305
22.10 What is recorded	305
22.11 Worked procedure: commissioning a conveyor	306
22.12 Troubleshooting	307
23. Trace and data snapshots	308
23.1 Trace or monitor?	308
23.2 How the trace works	308
23.3 The Trace / scope page	309
23.4 Recording a trace	310
23.5 Exporting a trace	311
23.6 Example: tuning a control loop	311
23.7 Data snapshots	312
24. Breakpoints and debugging	315
24.1 What a breakpoint does to the machine	315
24.2 Debug menu	316
24.3 Setting breakpoints	316
24.4 When the program stops	317
24.5 Stepping	317
24.6 The call stack and the breakpoint list	317
24.7 Ending a debugging session	318
24.8 Example	318

24.9 Alternatives to breakpoints	319
25. Diagnostics and messages	320
25.1 Where to look	320
25.2 The Messages panel	320
25.3 The controller log and the audit trail	322
25.4 IO diagnostics	324
25.5 Scan statistics	325
25.6 Panel lamps	326
25.7 Where used (cross-reference)	326
25.8 Versions and compatibility	327
25.9 Checklist: something is wrong	327
26. The simulator	328
26.1 What the simulator is	328
26.2 The Simulator page	328
26.3 How the project is transformed	330
26.4 Simulating inputs and outputs	331
26.5 Using the simulator	332
26.6 Connecting a SCADA system to the simulator	332
26.6a Connecting a 3D process simulation (Factory I/O)	333
26.7 Simulating the process	334
26.8 Limitations compared with a controller	335
26.9 zeus-sim.exe from the command line	336
26.10 Troubleshooting	336
27. Alarms	339
27.1 How alarms work in Zeus	339
27.2 The state model (ISA-18.2)	340
27.3 Priorities	341
27.4 Defining alarms	341
27.5 The alarm view (Online → Alarms...)	344
27.6 Operator actions	345
27.7 Who may do what	346
27.8 The alarm journal	347
27.9 Analogue alarms with ALARM_AN	348
27.10 Screen alarm lists (HMI → Alarms)	350
27.11 Alarms on the simulator	350
27.12 Checklist for an alarm configuration	350
28. Batch control, e-signatures and validation	351
28.1 Why there is a batch layer in the controller	351
28.2 The library blocks	351
28.3 The phase: S88_PHASE	352
28.4 Units, arbitration and mode	356
28.5 Driving phases from a batch engine	358
28.6 Batch context in controller records	359
28.7 Electronic signatures	360
28.8 Validation documents	361
28.9 Library versions and locking	362
28.10 Summary: a batch-ready project	364
29. OPC UA server	365
29.1 Overview	365
29.2 Enabling the server	365

29.3 The address space	367
29.4 Publishing variables	369
29.5 Security and users	372
29.6 Connecting a client	373
29.7 Ports on a PC running the simulator	374
29.8 When the address space is built	374
29.9 Troubleshooting	375
30. EtherNet/IP tag server	376
30.1 Overview	376
30.2 Enabling the tag server	376
30.3 Tags: names, types and browsing	378
30.4 Services	379
30.5 Writing tags	380
30.6 The EDS file	380
30.7 Connecting HMI and SCADA clients	381
30.8 Port 44818 on a PC running the simulator	382
30.9 Troubleshooting	383
31. Modbus TCP slave	384
31.1 Overview	384
31.2 Enabling the slave	384
31.3 The address map	386
31.4 Reads: consistency and timing	388
31.5 Writes	388
31.6 Unit id and gateways	389
31.7 Performance and scan impact	390
31.8 Availability on the simulator	390
31.9 Troubleshooting	390
32. Operator screens (HMI)	391
32.1 Concepts	391
32.2 Adding operator screens to a project	391
32.3 The screen editor	392
32.4 Widgets	393
32.5 Binding rules	413
32.6 The screens server	413
32.7 Downloading screens	416
32.8 Opening screens	416
32.9 Availability on the simulator	416
32.10 Troubleshooting	416
32.11 Local display on the controller	418
33. Retain, power-fail and maintenance	421
33.1 What survives a restart	421
33.2 RETAIN variables	421
33.3 The retain store	422
33.4 SD-card wear	424
33.5 Power-fail save	425
33.6 CONFIG mode	426
33.7 Backup and restore	427
33.8 Time synchronisation	428
33.9 Restart, reboot and versions	429
33.10 Updating the controller runtime	430
33.11 Maintenance checklist	430

34. Security	432
34.1 What is being protected	432
34.2 Users and roles	432
34.3 Lockout	434
34.4 Machine tokens	434
34.5 Protecting the connection	435
34.6 Program signing	436
34.7 The audit trail	436
34.8 Services and their exposure	437
34.9 Host firewall	437
34.10 Process hardening	438
34.11 Known limits	438
34.12 Hardening checklist for a production panel	439
35. Troubleshooting	440
35.1 Where to look first	440
35.2 Connection and login	440
35.3 Build and download	441
35.4 Run state, faults and performance	442
35.5 IO and fieldbuses	443
35.6 OPC UA	445
35.7 EtherNet/IP tag server	446
35.8 Modbus TCP slave	447
35.9 Operator screens	447
35.10 Alarms and batch	447
35.11 Retained data, power and time	448
35.12 Simulator	449
35.13 When to call support	449
36. Practices for validated systems	450
36.1 Responsibilities	450
36.2 People and access	450
36.3 Configuration management	450
36.4 Downloads and online changes	451
36.5 Forces, writes and debugging	451
36.6 Records	451
36.7 Electronic signatures	452
36.8 Alarms	452
36.9 Integration servers	452
36.10 Retained data and power	452
36.11 Backups and recovery	453
36.12 Measurements and qualification evidence	453
36.13 Change control and periodic review	453
36.14 Quick checklist before a controller enters GxP service	453
37. Licensing and demo time	455
37.1 What runs where	455
37.2 When the demo time runs out	455
37.3 Checking the license	456
37.4 Getting and installing a license	456
37.5 New and existing Zeus controllers	457
Appendix A. Built-in functions	459
A.1 Arithmetic and mathematics	460
A.2 Selection and limits	460

A.3 Bit shift and rotate	461
A.4 Type conversion	461
A.5 Time	461
A.6 Process image and raw data	462
A.7 Examples	462
Appendix B. Standard function blocks	463
B.1 TON	463
B.2 TOF	463
B.3 TP	463
B.4 CTU	464
B.5 CTD	464
B.6 CTUD	464
B.7 R_TRIG	465
B.8 F_TRIG	465
B.9 RS	465
B.10 SR	465
B.11 Example	466
Appendix C. Library reference	467
C.1 10_signal.st — zeus-signal 1.0.0	467
C.2 20_control.st — zeus-control 1.0.0	472
C.3 22_autotune.st — zeus-autotune 1.0.0	477
C.4 30_equipment.st — zeus-equipment 1.0.0	478
C.5 40_hvac.st — zeus-hvac 1.0.0	484
C.6 50_s88.st — zeus-s88 1.0.0	486
C.7 70_iolink.st — zeus-iolink 1.0.0	489
Appendix D. Limits and capacities	494
D.1 Program and language	494
D.2 Process image and I/O	494
D.3 Communication and services	495
D.4 Diagnostics and alarms	495
D.5 What is not a limit	495
Appendix E. Keyboard shortcuts	496
E.1 Anywhere in the IDE	496
E.2 Code editor (Structured Text)	496
E.3 Online variable table	497
E.4 Debugging (while halted at a breakpoint)	497
Appendix F. Glossary	498
Appendix G. Worked example projects	500
G.0 How to use these examples	500
G.1 Tank fill with level control, safety timeouts and alarms	502
G.2 Conveyor with start/stop, jam detection and fault reset	509
G.3 Temperature control with PID and auto-tune	514
G.4 Duty/standby pump set with alternation and run-hour balancing	520
G.5 Valve and motor management with a device table	524
G.6 Batch mixing sequence in SFC	533
G.7 A multi-task project	539
G.8 IO-Link sensor: parameters at start-up and event handling	546
Appendix H. Controller configuration file	554
H.1 Sections	554

H.2 `io[].scale[]` — signal ranges
H.3 Driver parameters (`io[].params`) — main keys 555

ZEUS PLC IDE · OPERATION MANUAL

Part I — Getting started

1. Product overview

This chapter describes what the Zeus PLC system is, which parts it is made of and how they fit together. It compares the two runtime options available on the Zeus PLC IPC-CM400200 controller, lists what the native runtime provides, and states plainly what is still in development. Read it once before you commission a first controller; the rest of the manual assumes the vocabulary introduced here.

WARNING: The Zeus PLC controller is not a safety controller. It carries no SIL or PL rating and must not implement a safety function. Emergency stop, guard interlocking and any other function that protects people belong in a certified safety relay or safety PLC, wired independently of this controller. This applies to both runtime options described below.

1.1 What the system is

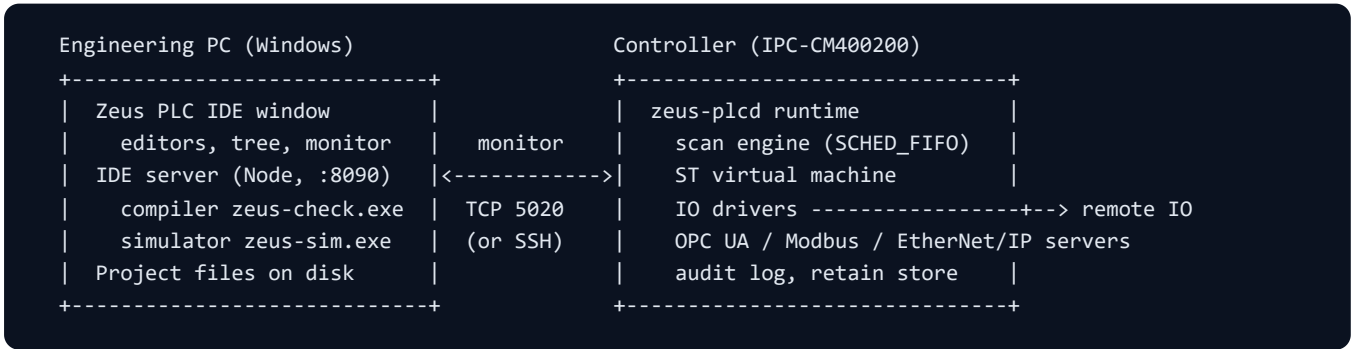
Zeus PLC is a soft-PLC system made of three parts:

Part	What it is	Where it runs
Controller	The Zeus PLC IPC-CM400200 industrial PC (Raspberry Pi Compute Module 4 on an industrial carrier), or its CM5-based successor IPC-CM500200	In the cabinet
Runtime	zeus-plcd , the real-time program that runs the scan: reads inputs, executes the IEC 61131-3 application, writes outputs, and serves the protocols listed in section 1.6	On the controller, under a PREEMPT_RT Linux kernel
Zeus PLC IDE	The engineering tool: editors, device tree, compiler, simulator, download, online monitoring, trace and diagnostics	On a Windows engineering PC, in its own application window

The IDE and the runtime talk over the **monitor protocol**: newline-delimited JSON on TCP port 5020 of the controller. On a hardened controller that port is bound to loopback and the IDE reaches it through an SSH tunnel it opens itself (Chapter 6).

A **project** is a directory of plain text files on the engineering PC: programs, data types, global variables, the function-block library, the IO configuration and operator screens. It can be put under version control, compared with ordinary tools and archived into a single file (Chapter 5). One project can be downloaded to several controllers and to the local simulator.

The controller does not depend on the IDE. It stores the last downloaded program as its **boot program**, starts it at power-up, and keeps running whether or not an IDE is connected. Disconnecting the IDE never stops a machine.



1.2 The controller: Zeus PLC IPC-CM400200

	IPC-CM400200	IPC-CM500200
Compute module	Raspberry Pi CM4	Raspberry Pi CM5
SoC	BCM2711, 4 cores	BCM2712, 4 cores
GPIO controller	<code>pinctrl-bcm2711</code>	<code>pinctrl-rp1</code>
Ethernet	On the SoC	Through the RP1 I/O controller
Minimum recommended task interval	500 µs	250 µs
Target kind in the IDE	IPC-CM400200	IPC-CM500200

One runtime binary runs on both models. At start-up the runtime reads the device tree, identifies the board and reports it to the IDE; the IDE shows it in the target badge (for example `CM4 · REAL IO`) and refuses to treat a controller as the model you selected if it reports a different one (Chapter 6).

The CM4 has no battery-backed real-time clock of its own; plan NTP time synchronisation for any controller whose timestamps matter (alarms, audit trail, electronic signatures). The DF40HC3.0 carrier used in current units adds a UPS with a supercapacitor and a power-fail signal, which the runtime uses to save retained data and shut down cleanly (Chapter 33).

1.3 The two runtime options

The IPC-CM400200 can be delivered with either of two control runtimes. The hardware is the same; the choice is about licensing, tooling and features.

- **Codesys V3.5** — the Codesys Control runtime with the Codesys Development System as the engineering tool.
- **Native Zeus runtime + Zeus PLC IDE** — the runtime and IDE described in this manual.

Both are IEC 61131-3 environments; a controls engineer who knows one will recognise most concepts in the other. The table below is a fair summary to help you choose. It is not a feature-by-feature certification; confirm any capability that decides your project against the current documentation of the option you pick.

Aspect	Codesys V3.5	Native Zeus runtime + Zeus PLC IDE
Runtime licence	A Codesys runtime licence is needed per controller; add-ons (for example motion or visualisation) are licensed separately	No per-unit runtime licence
Engineering tool	Codesys Development System (Windows desktop application)	Zeus PLC IDE: self-contained Windows package, no installer, runs in its own app window (Chapter 2)
IEC 61131-3 languages	ST, LD, FBD, SFC, IL (deprecated), plus CFC	ST, LD (graphical, compiles to ST), SFC. No FBD, CFC or IL
Object orientation	Function blocks with methods, properties, interfaces, inheritance	Function blocks with METHODS; no interfaces or inheritance
Task model	Several tasks with their own intervals and priorities	One cyclic task with a call list of up to 16 programs, each able to run every Nth scan (Chapter 18)

Aspect	Codesys V3.5	Native Zeus runtime + Zeus PLC IDE
Fieldbuses	Broad set of configurators and stacks	Modbus TCP (client), Profinet IO controller, EtherCAT master, EtherNet/IP scanner — see section 1.6 for the maturity of each
OPC UA server	Yes	Yes
HMI	Integrated visualisation (licensed per target)	Operator screens drawn in the IDE and served by the controller's own web server (Chapter 32); OptiZeus SCADA for plant-level HMI
Motion control	SoftMotion (licensed add-on)	Not yet — Zeus Motion is in design (section 1.9)
Redundancy	Available as an add-on	In development (section 1.9)
GxP / validated-system features	Provided by the application or add-ons	Built in: per-action audit trail with user and address, 21 CFR Part 11 electronic signatures, ISA-18.2 alarm states raised at source, ISA-88 phase blocks, library version lock with SHA-256, program signing (Chapters 27, 28, 34, 36)
Access control	User management in the runtime	Named users with viewer / operator / engineer roles, enforced by the controller for every command (Chapter 6)
Maturity and ecosystem	Long-established, large library and user community	Young product; smaller library (Appendix C); some drivers not yet field-validated
Source ownership	Vendor product	The whole stack is owned by Zeus Automation, so a site problem can be fixed in the runtime itself
Safety functions	Not on this hardware	Not on this hardware

How to choose. Choose Codesys V3.5 when the project needs FBD or CFC, several independent tasks, motion control today, or a third-party library that exists only for that environment. Choose the native Zeus runtime when per-unit licensing matters across a fleet, when the plant is a regulated (GxP) environment that benefits from the built-in audit, signature and library-lock features, or when the controller is part of an OptiZeus SCADA system.

Note: A controller runs one runtime at a time. Switching an installed controller from one option to the other is a re-imaging job, not a setting, and the application must be re-engineered in the other tool — projects are not exchanged between the two IDEs. The Zeus PLC IDE can import Structured Text from PLCopen XML and text exports (**Project** → **Import ST...**, Chapter 13), which covers much of the ST code but not hardware configuration or visualisations.

The rest of this manual describes only the native Zeus runtime and the Zeus PLC IDE.

1.4 What the native runtime provides

Area	What the runtime does
Scan	One cyclic task on a real-time (SCHED_FIFO) thread with a fixed interval, overrun policy (skip, catch up, or fault) and a watchdog. A call list of up to 16 PROGRAMs runs in order every scan; each program may run every Nth scan with an offset (Chapter 18).
Languages	Structured Text, Ladder Diagram and Sequential Function Chart, compiled on the controller by the same compiler the IDE uses. FUNCTION, FUNCTION_BLOCK (with METHODS) and PROGRAM POU; user data types (STRUCT, ENUM, alias) (Chapters 13–17).

Area	What the runtime does
Execution safety	Every scan has an instruction budget, so a runaway loop faults that cycle instead of hanging the core. Arrays are bounds-checked. Integer division by zero is a counted <i>minor fault</i> : the scan continues (Chapter 21).
Online change	A program download keeps the values of variables whose name and type did not change, including the elapsed time of running timers; the IDE reports what was carried over, reset and dropped (Chapter 20).
Process image	<code>%I</code> , <code>%Q</code> and <code>%M</code> areas. <code>%I</code> / <code>%Q</code> are sized from the configured IO devices; <code>%M</code> is 4096 bytes by default. Analogue channels can be converted to engineering units in the IO layer (Chapter 11).
IO fault handling	A rack with one dead device keeps running the others (<code>DEGRADED</code>); a device can be marked <i>critical</i> so its loss faults the controller; per-device output behaviour on STOP and on FAULT (clear or hold last value) (Chapter 7).
Retained data	<code>%M</code> and every <code>RETAIN</code> variable survive a power cycle in an atomic, CRC-checked store with a layout signature; power-fail save on carriers with a UPS (Chapter 33).
Alarms	ISA-18.2 alarm states (unacknowledged, acknowledged, return-to-normal, shelved, suppressed, out of service), evaluated on the scan thread, with a journal; up to 10,000 alarms (Chapter 27).
Batch and records	ISA-88 phase, unit, resource and mode blocks; batch record context; 21 CFR Part 11 electronic signatures with re-authentication (Chapter 28).
Audit trail	Every action that changes the machine (download, write, force, run state, login, signature) is logged with the user name and the network address it came from.
Diagnostics	Scan time and jitter statistics, per-driver state and uptime, controller log readable from the IDE, trace recorder of 8 channels × 20,000 samples at scan rate, up to 32 breakpoints with single-step (Chapters 23–25).
Security	Named accounts with roles, salted PBKDF2 password hashes, per-address lockout after failed logins, optional SSH tunnel or TLS for the monitor protocol, Ed25519 program signing with rollback protection (Chapter 34).

1.5 How a program gets to the controller

1. You edit POU's in the IDE. The IDE saves them as `.st` files in the project directory.
2. **Build** (F7) compiles the POU on the engineering PC with `zeus-check.exe` — the same compiler source the controller runs — together with the global variables, the data types and the library, and lists every error in the **Messages** panel.
3. **Download program** (F8) links the whole application (library, globals, data types and every program in the task's call list) into one unit, signs it with the site key, and sends it to the controller.
4. The controller compiles it again, binds its `AT` addresses to the process image, and only then replaces the running program between two scans. A program with errors is refused and the running program is not disturbed.
5. The controller stores the source it is running. **Verify, Compare...** and the header's sync badge ask the controller what it runs, rather than trusting a record of what was sent (Chapter 20).

Hardware configuration (IO devices, process image, servers, accounts) travels separately with **Download hardware**, which restarts the runtime and therefore stops the control loop briefly.

1.6 Protocols

1.6.1 Remote IO (the controller as master)

Protocol	Role	Validated hardware	Status
Modbus TCP	Client (master), cyclic	ODOT CN-8031 coupler with C-series cards; any Modbus TCP server (drives, meters, third-party IO)	Production
Profinet IO	IO controller: DCP discovery, name and IP assignment, GSDML rack layout, cyclic RT exchange	ODOT CN-8032 / CN-8032-L	Working against the CN-8032 at full cycle rate
EtherCAT	Master (SOEM-based), rack laid out from the vendor ESI file	ODOT CN-8033 modular coupler	Bench-proven against the CN-8033; walk the bring-up procedure with outputs disconnected before first use on a machine (Chapter 9)
EtherNet/IP	Scanner (originator), Class 1 cyclic IO	—	Not yet validated against a real adapter; treat as a bench starting point
IO-Link	Through an EtherCAT IO-Link master (ODOT IP-EC-8IA): port configuration from the ESI start-up commands, ISDU and event blocks in the library	ODOT IP-EC-8IA	Not yet validated on real hardware (Chapter 12)

Up to 16 IO devices can be configured per controller; EtherCAT is limited to one master with one modular coupler per segment and a 4096-byte process image (Appendix D).

1.6.2 Servers (the controller as server)

Server	Default port	Purpose	Chapter
Monitor protocol	TCP 5020	The IDE connection: download, watch, write, force, trace, debug	6
OPC UA server	TCP 4840	SCADA and historians; program variables and function-block instances as nodes	29
EtherNet/IP tag server	TCP/UDP 44818	Read (and optionally write) program variables by name from EtherNet/IP (CIP) clients such as HMIs and SCADA, with no register mapping	30
Modbus TCP slave	TCP 5502	Expose process-image ranges to a Modbus master; off by default and read-only by default	31
Operator screen server	TCP 8080	Serves the HMI screens drawn in the IDE to a browser or panel	32

Every server is off or bound to loopback until you configure it. None of them is required for the IDE to work.

1.7 Capacities at a glance

The most important limits of the current release are below. The complete, authoritative list is in **Appendix D**.

Limit	Value
POUs per download unit (library, every FB, method and program)	256
Programs in the task call list	16
Members per STRUCT or function block	128
Parameters per FUNCTION / FB call	64
Call depth	32
Enumeration constants (whole project)	256
STRING length	254 characters
Array dimensions	3
IO devices	16
%M area (default, configurable)	4096 bytes
Alarms	10,000
Trace	8 channels × 20,000 samples
Breakpoints	32
Controller accounts	16 users

1.8 How Zeus PLC fits with OptiZeus SCADA

OptiZeus is the SCADA system from the same supplier. A Zeus controller appears in OptiZeus as an ordinary data source:

Link	How	Typical use
OPC UA	OptiZeus connects to the controller's OPC UA server (port 4840) and browses program variables and function-block instances by name	Plant-wide SCADA, historian, alarms and reports
EtherNet/IP tags	OptiZeus (or any CIP client) reads tags by name from the tag server (port 44818)	Where the SCADA or panel prefers CIP
Modbus TCP	Through the controller's Modbus TCP slave	Legacy masters

Responsibilities are split deliberately. The controller owns control, interlocks, alarm detection with scan-accurate timestamps, retained data and the audit trail of engineering actions. OptiZeus owns operator-level visualisation across many controllers, history, reporting and plant alarm management. The operator screens that the controller serves itself (Chapter 32) are for a local panel on the machine; they do not replace the SCADA.

The simulator in the IDE (Chapter 26) lets you develop the SCADA side against a running program before the hardware exists. Note which simulator build you have:

- The simulator in the self-contained Windows package (`bin\zeus-sim.exe`) runs the real scan engine, program, alarms and monitor protocol, but **no OPC UA server, no Modbus slave, no EtherNet/IP tag server and no fieldbus**. The IDE says so in the start-up report.
- On an engineering PC that also has a Linux build of the runtime (under WSL), the IDE starts that build instead, and it serves OPC UA and EtherNet/IP exactly like a controller — that is the set-up to use when connecting

OptiZeus to a simulated controller.

1.9 In development

The following are announced or partly built. Do not design a project that depends on them without confirming their status with the supplier.

Feature	State
Zeus Motion	Design stage. Planned first stage: CiA 402 servo drives over EtherCAT, PLCopen motion blocks (<code>MC_Power</code> , <code>MC_Home</code> , <code>MC_MoveAbsolute</code> ...) and an axis commissioning page in the IDE. Not in this release.
Two-controller redundancy	Hot-standby pair over a dedicated sync link. Transport, arbitration (split-brain protection), state synchronisation and configuration are built and unit-tested, and Controller Settings already shows the redundancy fields; integration with the scan engine, EtherCAT takeover and the IDE status view still need bench validation with two controllers. Do not rely on failover in this release.
IO-Link	Library blocks, ESI start-up commands and the IDE IO-Link ports page are in this release, but they have not yet been validated against real IO-Link hardware.
EtherNet/IP scanner	Driver and scan dialog present; not yet validated against a real adapter.
Several independent tasks	Only one cyclic task exists today; programs share it and can run at slower multiples of its interval.

1.10 Documentation set

Document	Content
This manual	Operation and programming of the Zeus PLC IDE and the native runtime
? Help in the IDE	The built-in quick reference, and a separate Ladder editor help
<code>docs/04-config-schema</code>	The controller configuration file (summarised in Appendix H)
<code>docs/05-monitor-protocol</code>	The IDE–controller protocol, for integrators
<code>docs/07</code> , <code>docs/08</code> , <code>docs/35</code>	EtherCAT, EtherNet/IP and IO-Link bring-up procedures
<code>docs/09</code> – <code>docs/21</code>	Security design, hardening and provisioning
<code>docs/27</code> – <code>docs/31</code>	Validation plan, user requirements, functional specification, IQ/OQ protocols, traceability matrix

2. Installation and starting

This chapter covers installing the Zeus PLC IDE on a Windows engineering PC, starting and stopping it, the network ports it uses, where it writes its logs and state, how to update and remove it, and a first run with the example project. You need it once per engineering PC, and again when a new version is delivered.

The controller itself is delivered with its runtime installed. Installing or replacing the runtime on a controller (**Deploy runtime...**) is described in Chapter 6.

2.1 System requirements

Item	Requirement
Operating system	Windows 10 or Windows 11, 64-bit
Browser	A Chromium-based browser for the application window. Microsoft Edge ships with Windows, so this is normally already satisfied; Google Chrome and Brave are also found automatically. Firefox cannot provide the app window (see 2.4.4).
Disk	About 90 MB for the unpacked package, plus your projects
Memory	No special requirement beyond the browser
Rights	An ordinary user account. Nothing is installed system-wide and no administrator rights are needed.
Network	Ethernet access to the controllers you will program. No internet access is needed at any point.
SSH client	The OpenSSH client that ships with Windows 10/11 (<code>ssh.exe</code> , <code>scp.exe</code>) — needed for Download hardware , Deploy runtime... , the SSH tunnel and the controller maintenance commands. Not needed for the simulator.

Note: The package contains no runtime for the controller. `zeus-plcd` needs a PREEMPT_RT Linux kernel and real IO; it runs on the controller, not on Windows. What the package does contain is the same **compiler** and a **simulator** built from the same source, so a program that compiles and behaves in the IDE compiles and behaves on the controller, apart from the real-time timing a PC cannot provide.

2.2 Package contents

The IDE is delivered as one ZIP file named `ZeusPLC-IDE-<version>-win-x64.zip` (about 35 MB). It unpacks into a folder of the same name:

ZeusPLC-IDE-0.1.0.13-win-x64\	
Zeus PLC IDE.cmd	the launcher - double-click this
README.txt	short getting-started text
VERSION	the release number of this package (compared with the update announcement)
node\node.exe	Node.js runtime for the IDE server (unmodified, MIT licence)
bin\zeus-check.exe	the IEC 61131-3 Structured Text compiler
bin\zeus-sim.exe	the simulator - runs a project with no hardware
ide\	the IDE itself (server in ide\src, user interface in ide\public)
stdlib\	the standard function-block library (*.st)
projects\example\	a small program that compiles as delivered
licences\	NOTICE.md and the Node.js licence

Part	Purpose
Zeus PLC IDE.cmd	Starts, stops and reports the IDE. Always uses the bundled node\node.exe , never a Node.js installed elsewhere on the PC, so every PC gets the same result.
node\node.exe	The only runtime the IDE server needs. The IDE has no third-party npm packages.
bin\zeus-check.exe	Used by Build , Build All and by every download for a pre-check. Static executable, no DLLs.
bin\zeus-sim.exe	Started by the Simulator node (Chapter 26).
stdlib\	Copied into every new project's lib\ folder, and used by any project that has no library of its own (Chapter 5).
projects\example\	A starting program (section 2.9).

Nothing is downloaded at run time. The package runs on a PC with no internet connection, which is where most engineering PCs for a plant network are. The one thing the IDE fetches when the PC has internet is a small announcement of the latest release (section 2.8.1); without internet it simply says nothing.

2.3 Installing

1. Copy the ZIP file to the engineering PC.
2. Right-click the ZIP file and choose **Extract All....** Extract it to a folder you can write to, for example C:\Zeus\ or D:\Tools\ . The result is a folder such as C:\Zeus\ZeusPLC-IDE-0.1.0.13-win-x64\ .
3. Do not run the IDE from inside the ZIP (from Explorer's compressed-folder view). It must be able to write to its own folder.

That is the whole installation. There is no installer, no registry entry and no service.

Important: Extract to a folder your user can write to. The IDE writes its log, its list of controllers, its recent-projects list and the site program-signing key into the package folder (section 2.7). A package extracted under C:\Program Files\ by an administrator will not be writable by an ordinary user.

Tip: Keep your projects **outside** the package folder, for example in C:\ZeusProjects\ or in a version-controlled folder. A project inside the package folder is deleted with it, and is easy to lose when you update (section 2.8).

2.4 Starting and stopping the IDE

2.4.1 Starting

Double-click **Zeus PLC IDE.cmd** in the package folder.

The launcher runs hidden — no console window appears. It starts the IDE server in the background and opens the IDE window. After a few seconds the **Zeus PLC IDE** window appears: an application window with its own taskbar entry and icon, no address bar and no browser tabs.

If the IDE is already running, double-clicking the launcher again opens another window onto the same running IDE; it does not start a second server.

2.4.2 Closing the window and stopping the IDE

The window and the server are separate:

You do	Result
Close the IDE window	The window closes. The IDE server keeps running in the background, with the controller connection and any SSH tunnel it holds. Double-click the launcher to open a window again.
Run "Zeus PLC IDE.cmd" stop	Closes every IDE window of this IDE, stops the server and everything it started (for example an SSH tunnel), and prints what it stopped.

Note: Neither closing the window nor stopping the IDE affects a controller. Its scan keeps running, its outputs keep being driven, and any forced values stay forced. Clear forces (**Debug** → **Clear forces**) before you leave a controller, not after.

2.4.3 Launcher commands

The launcher accepts a command as its first argument. Open a Command Prompt in the package folder (in Explorer, type `cmd` in the address bar of the folder and press Enter) and run:

Command	What it does
"Zeus PLC IDE.cmd" or "Zeus PLC IDE.cmd" open	Open a window; start the server first if it is not running
"Zeus PLC IDE.cmd" start	Start the server only, with no window
"Zeus PLC IDE.cmd" stop	Stop every running IDE server found, and close its windows
"Zeus PLC IDE.cmd" status	Report whether the IDE is running, its address, process id and log file
"Zeus PLC IDE.cmd" restart	Stop, then start again, and open a window

Add `/wait` to any command to keep the console open afterwards so you can read the result, for example `"Zeus PLC IDE.cmd" status /wait`. Without it, a command that fails also pauses so its message can be read.

Example output:

```
C:\Zeus\ZeusPLC-IDE-0.1.0.13-win-x64> "Zeus PLC IDE.cmd" status
running      http://127.0.0.1:8090/
pid          11234
log          ide\ide-server.log (48213 B)

C:\Zeus\ZeusPLC-IDE-0.1.0.13-win-x64> "Zeus PLC IDE.cmd" stop
stopped 8090 (pid 11234), closed 1 window
```

How the launcher finds a running IDE. It looks for it by port, not by a process-id file: it probes 8080 and 8090–8109 and asks whatever answers whether it is the Zeus PLC IDE. `stop` and `status` therefore work on an IDE however it was started, and `stop` stops every instance it finds and names each one.

2.4.4 When no app window appears

The app window needs a Chromium-based browser (Edge, Chrome or Brave). If none is installed, the IDE server still starts, but no window opens — and because the launcher runs hidden, nothing visible happens. In that case:

1. Run `"Zeus PLC IDE.cmd" status /wait` to confirm the IDE is running and read its address.
2. Open that address (normally `http://127.0.0.1:8090/`) in any browser.

The IDE works the same in an ordinary browser tab. The app window is only a convenience: no address bar, and no tab that can be closed by accident.

If the status says `not running`, read `ide\ide-server.log` (section 2.7); the last lines say why the server did not start.

2.5 Desktop and Start-menu shortcuts

The package does not create shortcuts. To start the IDE from the desktop, create a shortcut to the launcher:

1. In the package folder, right-click **Zeus PLC IDE.cmd** and choose **Show more options** → **Send to** → **Desktop (create shortcut)** (Windows 11) or **Send to** → **Desktop (create shortcut)** (Windows 10).
2. Optional: right-click the new shortcut, choose **Properties** → **Change Icon...**, and select `ide\public\zeus-ide.ico` in the package folder.
3. For a stop shortcut, make a second shortcut to the launcher and append `stop /wait` to its **Target** field, after the closing quote.

Note: The IDE source also contains a per-user shortcut installer, `ide\src\install-shortcut.ps1`, which creates **Zeus PLC IDE** and **Zeus PLC IDE - Stop** on the desktop and a **Zeus PLC** folder in the Start menu, and removes them again with `-Remove`. In the 0.1.0.7 package it looks for the launcher inside the `ide\` folder, where the package does not put it, and stops with `launcher not found`. Use the manual procedure above until a package fixes this.

2.6 Ports used

2.6.1 On the engineering PC

Port	Used by	Notes
TCP 8090 (8090–8109)	The IDE server	Bound to <code>127.0.0.1</code> only, so the IDE cannot be reached from other PCs. If 8090 is taken, the next free port up to 8109 is used; <code>status</code> shows which.
TCP 5020	The local simulator's monitor port	Bound to <code>127.0.0.1</code> . Configurable on the Simulator page (Monitor port).
Random local port	SSH tunnel to a controller	Chosen by Windows when a tunnelled target connects; bound to <code>127.0.0.1</code> .

The IDE server has no login of its own. It relies on being reachable only from the PC it runs on; the controller enforces who may do what (Chapter 6).

WARNING: Do not make the IDE server listen on a network address (the `ZPR_IDE_BIND` setting). Anyone who could reach it could download to, force and stop any controller the IDE is connected to. The server prints a warning if you do.

2.6.2 On the controller

Port	Service	Direction from the PC
TCP 22	SSH — hardware download, runtime deployment, tunnel, clock, restart/reboot/shutdown	PC → controller
TCP 5020	Monitor protocol — the IDE connection	PC → controller (direct, or inside the SSH tunnel)
TCP 4840	OPC UA server (if enabled)	SCADA → controller
TCP/UDP 44818	EtherNet/IP tag server (if enabled)	HMI/SCADA → controller
TCP 5502	Modbus TCP slave (if enabled)	Modbus master → controller
TCP 8080	Operator screen server (if enabled)	Panel/browser → controller
UDP 9910	Redundancy sync link (if enabled)	Controller ↔ controller

The fieldbus ports the controller uses as a master (Modbus TCP 502 to couplers, Profinet, EtherCAT, EtherNet/IP UDP 2222) are described in the fieldbus chapters (Chapters 8–10).

If the engineering PC has a firewall that restricts outgoing connections, allow TCP 22 and TCP 5020 to the controllers.

2.7 Where the IDE writes files

Everything the IDE writes outside your projects is inside the package folder, except the browser profile of the app window.

File or folder	Content
ide\ide-server.log	The IDE server log. Each start appends a line <code>=== started <time> on <port> ===</code> , followed by the server's start-up report: address, active target, open project and compiler path. Read this first when the IDE does not start.
.zeus-ide-targets.json	The controllers defined in Target... and which one is active, including monitor tokens (Chapter 6).
.zeus-ide-recent.json	The recently opened projects; the first entry is opened at the next start.
.zeus-ide-signing.key	The site program-signing key (private). Generated at the first start; restricted to your user account. Every download is signed with it and a controller can be set to refuse unsigned programs (Chapter 34).
.zeus-ide-signing.key.counter	The signing version counter used for rollback protection.
runtime\sim\	The simulator's working files: <code>sim-active.json</code> (the configuration it runs), <code>sim.out</code> (its console output), <code>runtime.log</code> and <code>retain.bin</code> .
%TEMP%\zeus-ide-profile-<port>\	The browser profile of the app window (cache and window size). Safe to delete when the IDE is stopped.

Messages from the IDE window itself — build results, download reports, and any internal error of the user interface — appear in the **Messages** panel at the bottom of the window (Chapter 4). The controller's own log is read with **Online** → **Controller** → **Controller log...** (Chapter 25).

Important: Treat `.zeus-ide-signing.key` like a password. Anyone with a copy can sign programs that your controllers accept. Include it in the engineering PC's protected backup, and never in a project archive or a ticket.

2.8 Updating to a new version

A new version is a new ZIP. The versions are independent folders, so an update never overwrites the version you are using until you decide to remove it.

1. Stop the running IDE: `"Zeus PLC IDE.cmd" stop`.
2. Extract the new ZIP next to the old folder (for example `C:\Zeus\ZeusPLC-IDE-0.1.0.13-win-x64\`).
3. Copy these files from the old package folder into the new one:
 - `.zeus-ide-signing.key` and `.zeus-ide-signing.key.counter` — **required** if your controllers verify program signatures. A new package folder without them generates a new key, and a controller provisioned with the old public key will warn about, or refuse, programs signed with the new one. Copying the counter keeps rollback protection working.
 - `.zeus-ide-targets.json` — your controller list and their tokens.
 - `.zeus-ide-recent.json` — optional, your recent projects.
4. If you kept projects inside the old package folder, move them out now (see the tip in 2.3).
5. Start the new version with its own **Zeus PLC IDE.cmd**.
6. Check **Online** → **Controller** → **Versions...** after connecting to a controller: it shows the IDE and runtime versions side by side. The IDE and the runtime are versioned independently; what decides whether they can

work together is the monitor protocol version, which the IDE checks when it connects and reports in the **Messages** panel if it is out of range.

7. When you are satisfied, delete the old package folder.

Updating the IDE does not change anything on a controller. Updating a controller's runtime is a separate, deliberate step (**Deploy runtime...**, Chapter 6), because it stops the control loop.

2.8.1 Update notification

The IDE tells you when a new release is out. 15 seconds after it starts, and every 12 hours while it runs, the IDE server reads `https://optizeus.org/zeus-ide/latest.json` - a small file with the latest version number, its download link and a few lines on what is new. The request carries nothing about the PC, the user or the projects.

When that version is newer than this package (the `VERSION` file), a blue **↑ Update 0.1.x.y** button appears in the top bar, after **? Help**. Click it to see what is new, with three choices:

Button	What it does
Download	Opens the new package's download in the browser. Then follow the steps above.
Later	Closes the window; the button stays in the top bar.
Skip this version	Hides the button for this version on this PC. A later version shows it again.

The **☞ Check for updates** button in the top bar (in the same place; the blue button replaces it while an update is waiting), or **Online** → **Controller** → **Check for updates...**, asks at once, even for a skipped version, and says "You have the latest version" when there is nothing new, or that optizeus.org could not be reached.

Without internet - a plant network - the check fails silently and nothing appears. To turn it off altogether, or to point it at a copy of the file on your own server, set the environment variable `ZPR_UPDATE_URL` before starting the IDE: `off`, or the address of the file.

Note: A new IDE version may bring a newer standard library in `stdlib\`. Existing projects keep their own `lib\` copy and do not change; the **Library versions** dialog shows which new library files are available and adds them on request (Chapter 5).

2.9 Uninstalling

1. Stop the IDE: `"Zeus PLC IDE.cmd" stop`.
2. Stop the simulator first if it is running (**Simulator** node → **Stop**), or end `zeus-sim.exe` in Task Manager.
3. Save anything you need from the package folder: projects kept there, and the signing key if other IDE installations or controllers depend on it.
4. Delete the package folder.
5. Optionally delete `%TEMP%\zeus-ide-profile-*` and any shortcuts you created.

Nothing else was installed.

2.10 The IDE window at first start

On the very first start the IDE has no recent project, so it opens the package's `projects\` folder as a project and prepares it: it creates `controller.json` with a default configuration, empty `pou\` and `dut\` folders and a copy of

the standard library in `lib\`. The window shows the device tree on the left and a welcome panel with **Add a device**, **New program** and a summary of the project (folder, number of devices and POU's, process-image sizes, and whether the compiler is available).

The welcome panel must say **Compiler: available**. If it says `NOT BUILT – Build is disabled`, the package is incomplete (`bin\zeus-check.exe` is missing); extract the ZIP again.

The example project is one level down, in `projects\example\`. Open it as described in 2.11, or create your own project as described in Chapter 3.

2.11 First run of the example project

Tip: The quickest start is the guided tour: **Project** → **Getting started tutorial...** (offered once when the IDE first starts). A panel in the corner takes you through nine steps - open an example, Build All, start the simulator, download, go online, force an input, release it, and view the program as ladder - and ticks each one off as you do it. The worked examples of Appendix G are ready to open from **Project** → **Open an example project....**

The example is a single program, `Main`, that starts a fill pump on a start button and stops it at a level setpoint or after a two-minute timeout:

```
PROGRAM Main
VAR
    start_pb  AT %IX0.0 : BOOL;
    level     AT %IW2   : INT;
    pump      AT %QX0.0 : BOOL;
    setpoint  : INT := 20000;
    running   : BOOL;
    fill_timer : TON;
    start_edge : R_TRIG;
END_VAR

start_edge(CLK := start_pb);
IF start_edge.Q THEN running := TRUE; END_IF;

(* A fill that never completes must not run the pump for ever *)
fill_timer(IN := running AND (level < setpoint), PT := T#120s);
IF fill_timer.Q THEN running := FALSE; END_IF;

pump := running AND (level < setpoint);
END_PROGRAM
```

To open and build it:

1. Click **Project** ▾ → **Open / switch project....**
2. In the **Open, create or copy** box, type the full path of the example, for example `C:\Zeus\ZeusPLC-IDE-0.1.0.13-win-x64\projects\example`, and click **Open**. *Result:* the IDE switches to the example. **Open** fills in what the folder lacks, so it now also has a `controller.json` and a `lib\` folder.
3. In the device tree, expand **Application** → **POUs** and double-click **Main**. *Result:* the program opens in a tab, with the declarations in the upper pane and the code in the lower pane.
4. Press **F7** (**Debug** ▾ → **Build**). *Result:* the **Messages** panel shows `0 error(s)` and a note with the symbol and code size.

To run it in the simulator, the project first needs IO that covers the addresses the program uses (`%IX0.0` , `%IW2` and `%QX0.0`). A new project has no IO devices, so its input and output areas are empty, and the controller refuses a program whose `AT` addresses lie outside them with a message such as `'start_pb'` is at `%I0`, which needs `%I` to be at least 1 byte, but it is 0 - add the IO device this address belongs to, or drop the `AT` clause .

5. Right-click **IO Devices** and add an **ODOT CN-8031** coupler with any host address (the simulator never contacts it) and these cards in this slot order: **CT-1116** (16 digital inputs, `%IX0.0` – `%IX1.7`), **CT-3104** (4 analogue inputs, `%IW2` – `%IW8`), **CT-2108** (8 digital outputs, `%QX0.0` – `%QX0.7`). Chapter 3 shows this dialog step by step.
6. Open the **Simulator** node and click **Start / Restart**. Leave **Generate moving input signals** ticked for this example: the digital inputs then toggle and the analogue inputs sweep, so `start_pb` presses itself and `level` rises and falls.
7. Log in when asked and set a new password for the `admin` account; then press **F8** to download `Main` . Chapter 3 explains the login and the password change in detail.
8. Click **Online ▾** → **Monitor** and watch `pump` follow `start_pb` and `level` .

Chapter 3 walks through the same cycle with a new project, from an empty folder to forcing inputs and adding a Ladder program.

3. Quick start: your first project

This chapter is a complete hands-on tutorial. Starting from an empty folder you will create a project, add IO, write a small Structured Text program, build it, run it in the simulator, watch it online, write a value, force inputs, make an online change and add a Ladder program. It takes about an hour and needs no hardware. Do it once before you touch a real controller: every step works the same way against the IPC-CM400200.

Each step says what to do, then what you should see. If what you see differs, stop and read the **Messages** panel at the bottom of the window before going on.

3.1 What you will build

A tank-fill control with a timeout, plus an agitator start/stop drawn in Ladder:

- A start push-button starts the fill pump, provided the stop circuit is healthy, the tank is not already full and no alarm is latched.
- The fill ends normally when the high-level switch operates; the completed fills are counted.
- The stop circuit (a normally closed contact, so a broken wire reads as *stop*) stops the fill.
- A fill that takes longer than `MaxFillTime` stops the pump and latches an alarm lamp until the alarm reset button is pressed.
- A separate Ladder program runs an agitator with a classic start/stop seal-in.

IO list (ODOT CN-8031 Modbus TCP coupler with one 8-channel input card and one 8-channel output card):

Address	Name	Signal
<code>%IX0.0</code>	<code>StartPB</code>	Fill start push-button, normally open
<code>%IX0.1</code>	<code>StopOK</code>	Fill stop circuit, normally closed — TRUE when healthy
<code>%IX0.2</code>	<code>LevelHigh</code>	High-level switch — TRUE when the tank is full
<code>%IX0.3</code>	<code>AlarmReset</code>	Alarm reset push-button
<code>%IX0.4</code>	<code>AgitStart</code>	Agitator start push-button, normally open
<code>%IX0.5</code>	<code>AgitStopOK</code>	Agitator stop circuit, normally closed — TRUE when healthy
<code>%QX0.0</code>	<code>Pump</code>	Fill pump contactor
<code>%QX0.1</code>	<code>AlarmLamp</code>	Fill-timeout alarm lamp
<code>%QX0.2</code>	<code>AgitRun</code>	Agitator contactor

3.2 Before you begin

- The IDE is installed and starts (Chapter 2).
- You have a folder for projects that is **outside** the IDE package folder, for example `C:\ZeusProjects\`.
- Nothing else is using TCP port 5020 on this PC (the simulator listens there).

3.3 Step 1 — Create the project

1. Click **Project** ▾ → **Open / switch project....** *Result:* the **Project** dialog opens, showing the current project and a **Recent** list.
2. In the **Open, create or copy** box, type `C:\ZeusProjects\tankfill`.
3. Click **Create**. *Result:* the dialog closes, a toast says `Created C:\ZeusProjects\tankfill`, and the **Messages** panel notes `Project switched to C:\ZeusProjects\tankfill`. The device tree shows a fresh project: the controller node with **Application**, **IO Devices**, **Network Interfaces**, **Modbus TCP Slave**, **OPC UA Server**, **Users & Roles**, **Controller Settings** and **Simulator**.
4. Open **Controller Settings**, change **Controller name** to `TankFill`, and click **Apply**. *Result:* the name in the menu bar, next to the **Z** mark, changes to `TankFill`.

Create refuses a folder that already contains files, so it can never scatter a project through an unrelated folder. The new project already contains `controller.json`, empty `pou\` and `dut\` folders and its own copy of the standard function-block library in `lib\` (Chapter 5).

3.4 Step 2 — Set the engineer password

Every new project contains one controller account, `admin`, with the **engineer** role and the initial password `admin`. Because that password is printed in this manual, the IDE will not send it to a controller: you must replace it before the first download.

1. In the device tree, click **Users & Roles**. *Result:* the **Users & Roles** dialog lists `admin` with the role `engineer` and an **initial password** badge.
2. Under **Add or replace**, enter:
 - **User:** `admin`
 - **Password** and **Confirm:** a password of at least 8 characters. `admin`, `password`, `12345678`, `zeus` and the user name itself are refused.
 - **Role:** `engineer – download and everything above`
3. Click **Save user**. *Result:* a toast says `admin saved – download hardware to apply`, and the **initial password** badge is gone.
4. Click **Close**.

Remember this password: you will log in to the simulator with it in step 6. The password is stored in `controller.json` only as a salted hash; nobody can read it back, only replace it (Chapter 6).

3.5 Step 3 — Add the IO

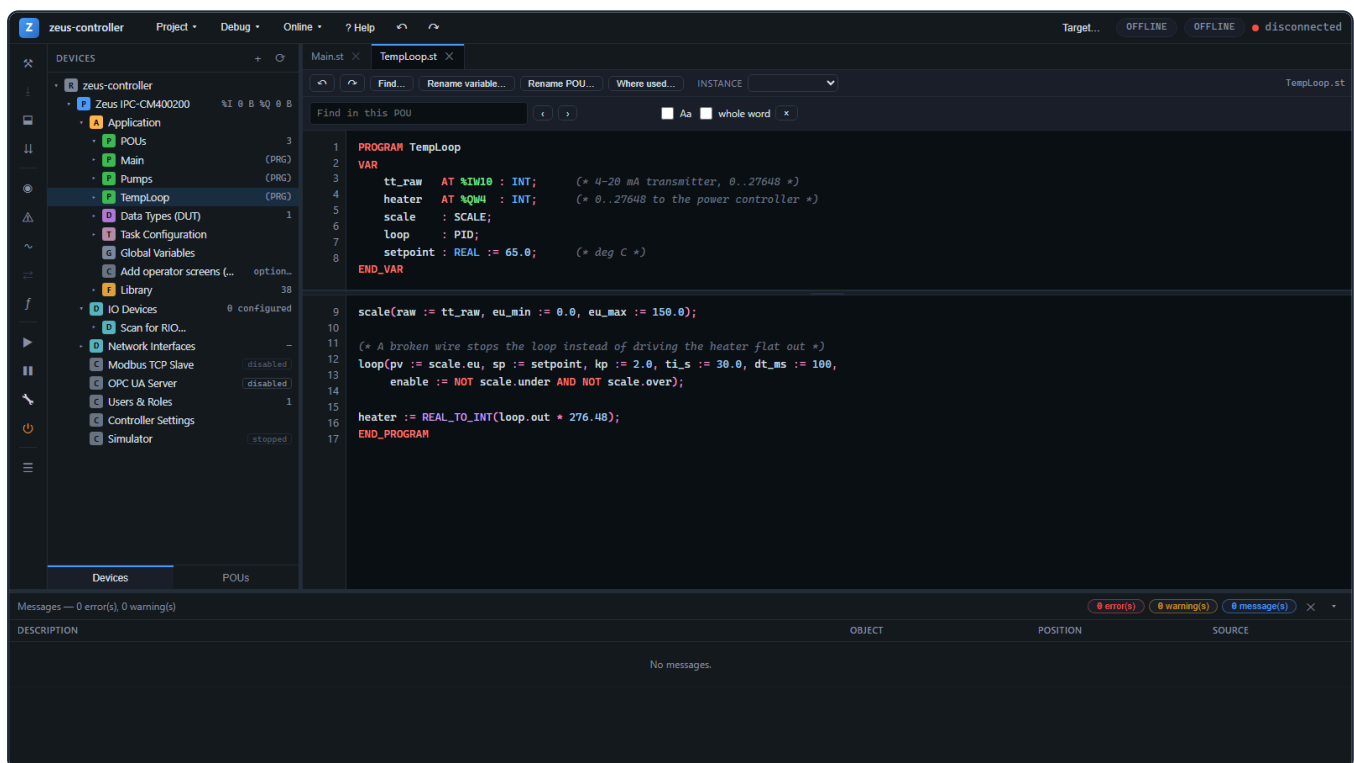
1. Right-click **IO Devices** in the device tree. *Result:* the **Add Device** dialog opens.
2. Under **1 • Device type**, click **CN-8031** (Modbus TCP network adapter, C-series bus).
3. Under **2 • Connection**, enter:
 - **Instance name:** `rack1`
 - **Host / IP:** `192.168.1.10` (any address will do for this tutorial — the simulator never contacts the coupler; on a real machine this is the coupler's IP address)
 - **Port:** `502`, **Unit id:** `1`
 - **Exchange mode:** `async – driver thread` (recommended)

- Under **3 · Modules (slot order)**, select **CT-1108 — 8x digital input, 24 VDC, sink/source** and click **Add to rack**. Then select **CT-2108 — 8x digital output, 24 VDC 0.5 A** and click **Add to rack**. *Result:* the slot list shows slot 1 CT-1108 at %IX0 and slot 2 CT-2108 at %QX0, and the note under it reads **Process image: %I 1 bytes, %Q 1 bytes**.
- Click **Add device**. *Result:* a toast says rack1 added, the device page for rack1 opens in a tab, and the tree shows **IO Devices → rack1** with its two cards.

Addresses are assigned in slot order. Inserting a card in front of another later moves every address after it; Chapter 11 explains how aliases protect a program from that.

3.6 Step 4 — Create the program

- Click the **POUs** tab at the bottom of the tree pane, then the **+** button at the top of the pane (its tooltip reads **New POU (program, function block or function)**). You can also right-click **Application** and choose **New POU....** *Result:* the **New POU** dialog opens.
- Enter **Name:** FillTank, choose **Type:** Program (PRG) – ST, leave **Folder** at (top level), and click **Create**. *Result:* FillTank appears under **Application → POUs** and opens in a tab. The editor has two panes: the declarations (PROGRAM FillTank and an empty VAR ... END_VAR) above, and the body (with END_PROGRAM) below.



The Structured Text editor: declarations above, body below (example program TempLoop)

- Click in the upper pane and complete the **VAR** block so that it reads:


```

PROGRAM FillTank
VAR
    StartPB      AT %IX0.0 : BOOL;  (* start push-button, normally open      *)
    StopOK       AT %IX0.1 : BOOL;  (* stop circuit healthy (NC contact) = TRUE *)
    LevelHigh    AT %IX0.2 : BOOL;  (* high-level switch, TRUE = tank full    *)
    AlarmReset   AT %IX0.3 : BOOL;  (* alarm reset push-button               *)
    Pump         AT %QX0.0 : BOOL;  (* fill pump contactor                   *)
    AlarmLamp    AT %QX0.1 : BOOL;  (* fill-timeout alarm lamp               *)

    MaxFillTime : TIME := T#30s;  (* longest acceptable fill                *)
    Filling      : BOOL;          (* a fill is in progress                  *)
    FillAlarm    : BOOL;          (* latched: the last fill timed out       *)
    FillsDone    : DINT;          (* completed fills since the last cold start *)

    StartEdge    : R_TRIG;
    FillTimer    : TON;
END_VAR

```

4. Click in the lower pane and type the statements above `END_PROGRAM`:

```

(* Start on the press, not while the button is held. *)
StartEdge(CLK := StartPB);
IF StartEdge.Q AND StopOK AND NOT LevelHigh AND NOT FillAlarm THEN
    Filling := TRUE;
END_IF;

(* Normal end of a fill: the tank is full. *)
IF Filling AND LevelHigh THEN
    Filling := FALSE;
    FillsDone := FillsDone + 1;
END_IF;

(* Stop pressed, or the stop circuit broken. *)
IF NOT StopOK THEN
    Filling := FALSE;
END_IF;

(* A fill that never completes must not run the pump for ever. *)
FillTimer(IN := Filling, PT := MaxFillTime);
IF FillTimer.Q THEN
    Filling := FALSE;
    FillAlarm := TRUE;
END_IF;

IF AlarmReset AND NOT Filling THEN
    FillAlarm := FALSE;
END_IF;

Pump := Filling;
AlarmLamp := FillAlarm;
END_PROGRAM

```

Result: the tab title and the POU in the tree show an unsaved-changes mark. While you type, the editor offers completions for declared names; accept one with **Enter** or **Tab**.

5. Hover the mouse over `FillTimer` in the body. **Result:** a tooltip shows its kind, type (`TON`), where it is declared and, later when online, its live value.

6. Press **Ctrl+S** (**Project** ▾ → **Save project**). *Result:* the unsaved marks disappear. The file is

`C:\ZeusProjects\TankFill\pou\FillTank.st`.

Tip: Instead of typing `AT %IX0.0`, you can click in the declaration pane and press **Ctrl+Space** to open **Browse IO**, pick a channel from the rack, and let the IDE write the declaration (Chapter 11). In the body pane, **Ctrl+Space** opens the completion list instead.

3.7 Step 5 — Build

1. With the `FillTank` tab active, press **F7** (**Debug** ▾ → **Build**, or  at the top of the tool strip on the far left). *Result:* the **Messages** panel title reads `Messages – 0 error(s), 0 warning(s), ...` and a note reports the program's symbols and code size.

Try the error path once, so you know what it looks like:

2. In the body, delete the semicolon after `Filling := TRUE` and press **F7** again. *Result:* the **Messages** panel shows an error with the POU in the **Object** column and `Line ..., Column ...` in **Position**. The line is marked in the editor.
3. Click the error row. *Result:* the editor jumps to the line and column of the error.
4. Put the semicolon back, press **F7** (no errors) and **Ctrl+S**.

The compiler reports every independent mistake in one build, not just the first. **Build** compiles the active POU together with the global variables, the data types and the library, so a missing library block or a global name typed wrongly is found here and not at download.

3.8 Step 6 — Start the simulator and log in

The simulator runs this project's configuration on your PC with the real scan engine and compiler. Each IO device is replaced by a simulated one with exactly the same `%I` / `%Q` sizes, so every address in your program is valid there (Chapter 26).

1. Click the **Devices** tab under the tree, then the **Simulator** node (near the bottom of the controller's branch). *Result:* the **Simulator** page opens with the status `STOPPED`.
2. Clear the **Generate moving input signals** check box. With it cleared, all simulated inputs hold 0 until you force them, which is what this tutorial needs.
3. Click **Start / Restart**. *Result:* after a few seconds the status reads `RUNNING 127.0.0.1:5020`, a toast says `Simulator running`, and the **Messages** panel lists the start-up steps, each with ✓: `runtime binary` (with a note that the Windows simulator has no OPC UA server, Modbus slave, EtherNet/IP tag server or fieldbus), `configuration`, `device – rack1: modbus_tcp simulated, %I 1 B / %Q 1 B ...`, `previous simulator`, and `simulator running`. The **Simulated devices** list shows `rack1`.
4. The IDE connects to the simulator by itself. Because the project has an account, the simulator requires a login, and the **Log in to the controller** dialog opens. Enter **User** `admin` and the password you set in step 2, and click **Log in** (or press **Enter**). *Result:* in the header, the target badge reads `SIMULATOR`, the session badge reads `admin · engineer`, the connection text reads `127.0.0.1:5020` next to a green dot, and a toast says `Logged in as admin (engineer)`.

Note: If you closed the login dialog, the run-state badge reads `NO SESSION`. Click it, or use **Online ▾ → Log in...** A login belongs to one connection: after the simulator or a controller restarts, you may be asked to log in again (Chapter 6).

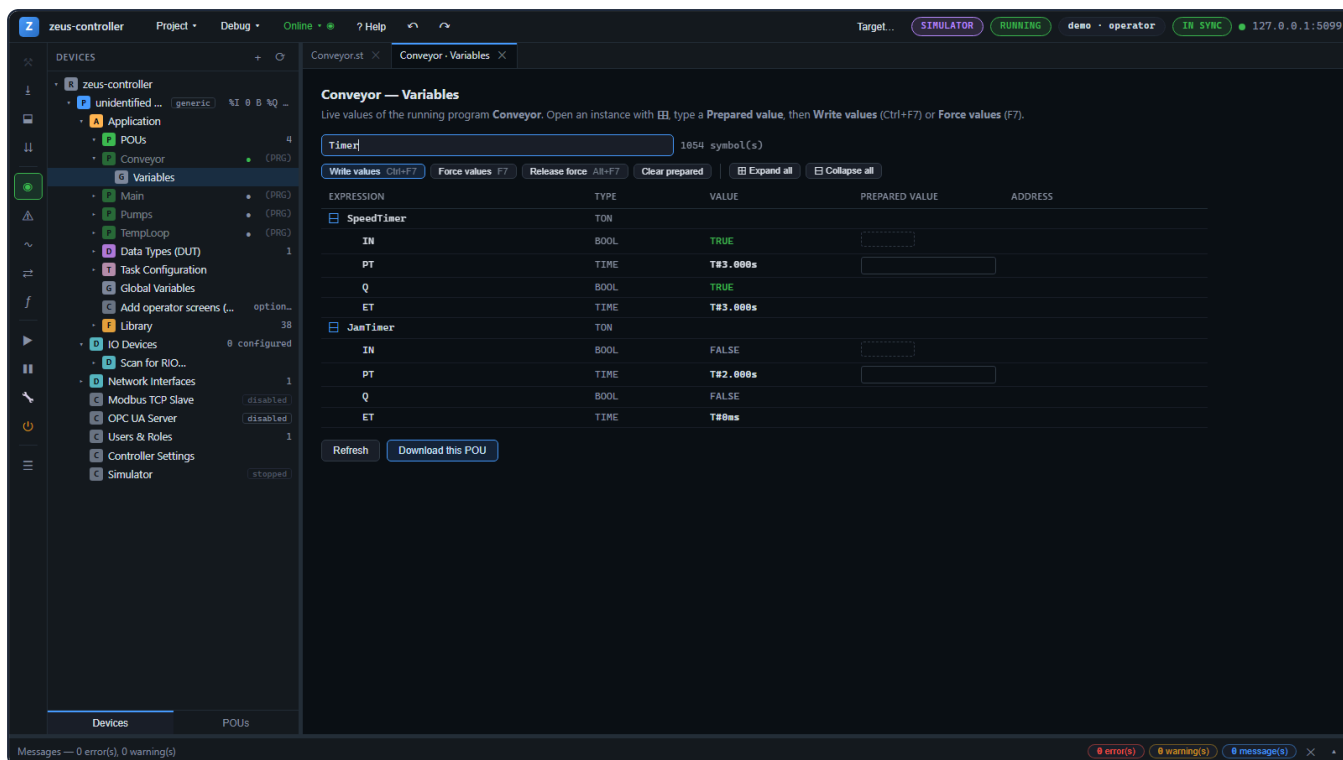
3.9 Step 7 — Download

1. Click the `FillTank` tab and press **F8 (Online ▾ → Download & Upload → Download program)**.
2. Confirm the dialog `Download "FillTank.st" to the controller? This REPLACES the running program...` with **OK**. *Result:*
 - The **Messages** panel reports `Downloaded FillTank: ... symbol(s), ... B code, ... B data. Controller is RUNNING. and Cold start - every variable is at its initial value.`
 - The run-state badge reads `RUNNING`.
 - The sync badge reads `IN SYNC`.
 - In the POU tree, `FillTank` carries a green dot (identical to the controller) and a `RUNNING` badge.

The controller compiled the program itself and bound every `AT` address to the process image before it replaced the running program. Had there been an error, it would have refused the program and kept running the old one.

3.10 Step 8 — Go online and watch

1. Click **Online ▾ → Monitor**, or  in the tool strip. *Result:* a toast says `Program and hardware match the controller`, the **Online ▾** menu title shows , and live values appear beside the declarations in the `FillTank` tab, refreshing several times a second.
2. In the tree, expand **Application → POUs → FillTank** and double-click **Variables**. *Result:* the **FillTank · Variables** tab opens: the online variable table with the columns **Expression**, **Type**, **Value**, **Prepared value** and **Address**.
3. Click  next to `FillTimer` to open the instance. *Result:* its members `IN`, `PT`, `Q` and `ET` appear indented under it. `PT` reads 30 s.



The online variable table with function-block instances expanded (example from another project)

Type part of a name, a type or an address into **Filter — name, type or address** to narrow the list, and click a column header to sort by it. Chapter 22 describes the table in full.

3.11 Step 9 — Write a prepared value

Shorten the timeout so that you do not have to wait 30 seconds for the alarm.

1. In the `MaxFillTime` row, click the **Prepared value** cell and type `T#10s`. *Result:* the table counts 1 prepared.
2. Click **Write values** (or press **Ctrl+F7**). *Result:* the **Value** of `MaxFillTime` changes to 10 s, the prepared cell empties, and the **Messages** panel notes `Wrote 1 value(s) on the running controller`.

A written value is sent once. The program may overwrite it on its next scan; `MaxFillTime` is only read by this program, so it keeps the new value. You can also right-click a row and choose **Write all prepared values (N)** or **Write only this one**.

WARNING: On a real machine a write takes effect on the next scan, without any further confirmation. Make sure the machine is clear before you write, force or download.

3.12 Step 10 — Force inputs and run a fill

Simulated inputs all read 0 at the moment, so the stop circuit reads as *broken* (`StopOK` is FALSE) and a start would be ignored. Force the inputs the way the field would drive them.

1. Right-click the `StopOK` row and choose **Force TRUE**. *Result:* `StopOK` reads TRUE and is marked as forced, and the header shows the amber badge `FORCE 1 point`.
2. Right-click the `StartPB` row and choose **Force TRUE**. *Result:* `Filling` and `Pump` turn TRUE, and `FillTimer.ET` starts counting. `FORCE` now counts 2 points.

3. Right-click the `StartPB` row and choose **Release force**. *Result:* `StartPB` returns to its real value, FALSE. The fill continues: the program latched `Filling` on the rising edge.
4. Wait 10 seconds. *Result:* `FillTimer.Q` turns TRUE, `Filling` and `Pump` turn FALSE, and `FillAlarm` and `AlarmLamp` turn TRUE.
5. Reset the alarm: right-click `AlarmReset` → **Force TRUE**, then right-click `AlarmReset` → **Release force**. *Result:* `FillAlarm` and `AlarmLamp` return to FALSE.
6. Run a fill that completes: force `StartPB` TRUE and release it again as in steps 2–3, then, within 10 seconds, right-click `LevelHigh` → **Force TRUE**. *Result:* `Filling` and `Pump` turn FALSE and `FillsDone` reads 1.
7. Right-click `LevelHigh` → **Release force**. Leave `StopOK` forced for the next steps.

Forcing is only offered for BOOL variables at a % address; a force holds the point at your value on every scan until it is released. An input can be forced but not written, because the IO scan would overwrite a write on the next cycle. Prepared values can also be forced in bulk with **Force values (F7)**, and **Release force (Alt+F7)** releases every forced row that the table is showing.

Important: The **FORCE** badge reflects the controller's own count, including forces set by another engineer or left from an earlier session. Never leave a controller with forces active. Clicking the badge offers to clear every force at once.

3.13 Step 11 — Make an online change

Add a counter of started fills, and download it without losing the data the program holds.

1. In the `FillTank` tab, add this line to the declarations, after `FillsDone` :

```
FillsStarted : DINT;          (* fills started - added by online change *)
```

2. In the body, add a line inside the first `IF` , so that it reads:

```
IF StartEdge.Q AND StopOK AND NOT LevelHigh AND NOT FillAlarm THEN
    Filling := TRUE;
    FillsStarted := FillsStarted + 1;
END_IF;
```

Result: as soon as you type, the sync badge changes to `PROGRAM CHANGED` and the dot next to `FillTank` in the tree turns orange. The live values on screen are now those of the program the controller is *still running* — the old one.

3. Press **F7**, then **F8**, and confirm. *Result:* the **Messages** panel reports `Online change - variable state carried over: ...` instead of a cold start. `FillsDone` still reads 1, `MaxFillTime` is still 10 s (the value you wrote, not the 30 s in the declaration), and `FillsStarted` starts at 0. The sync badge returns to `IN SYNC`.

Values are carried over by **name and type**: a variable that kept both keeps its value, a new or retyped variable starts from its initial value, and a removed variable is reported as dropped. Read the report after every download: it is the only place that says what the machine kept (Chapter 20).

3.14 Step 12 — Add a Ladder program

1. Create a second program: **+** in the **POUs** pane (or right-click **Application** → **New POU...**), **Name** `Agitator`, **Type** `Program – Ladder`, **Create**. *Result:* the Ladder editor opens with an empty rung. Its toolbar holds **+** **Rung**, **Declarations...**, **Add missing declarations**, **Show ST** and **? Help**, and below it the **Parts** palette: **NO**, **NC**, **Rising**, **Falling**, **Compare**, **Parallel**, **Block** and **Coil**.
2. Click **Declarations...** and replace the example declarations with:

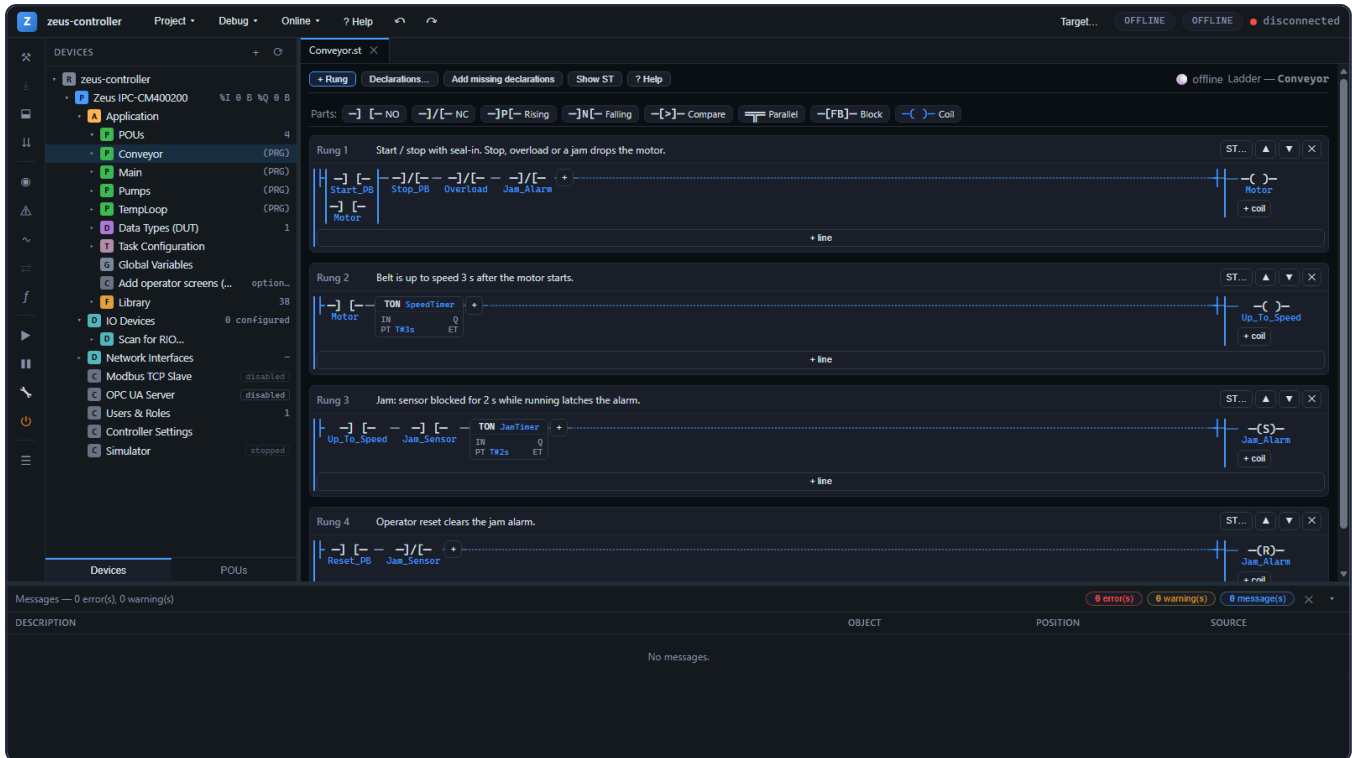
```
AgitStart  AT %IX0.4 : BOOL;  (* agitator start, NO *)
AgitStopOK AT %IX0.5 : BOOL;  (* agitator stop circuit healthy, NC *)
AgitRun    AT %QX0.2 : BOOL;  (* agitator contactor *)
```

and click **Save** in the **Declarations (VAR ... END_VAR)** dialog.

3. Type `Agitator start/stop with seal-in` into the rung's **comment...** box.
4. Click the **Parallel** part. *Result:* a parallel group is added to the rung and the **Parallel branch (OR)** dialog opens with **Path 1** and **Path 2**, each holding one normally open contact. Type `AgitStart` into the signal box of Path 1 and `AgitRun` (the seal-in contact) into the signal box of Path 2, and click **OK**.
5. Click the **NO** part. *Result:* the **Add element** dialog opens with **Kind** `Contact – normally open`. Pick or type the **Variable** `AgitStopOK` and click **OK**. The contact is placed in series after the parallel group. A normally open contact on the stop-circuit input is correct here: the circuit is wired normally closed, so the input is TRUE while nobody presses stop.
6. Click **+** **coil** at the right of the line, choose **Kind** `Coil`, set **Variable** to `AgitRun`, and click **OK**.
7. Click **Show ST**. *Result:* the generated Structured Text is shown under the diagram:

```
(* Rung 1: Agitator start/stop with seal-in *)
AgitRun := (AgitStart OR AgitRun) AND AgitStopOK;
```

8. Press **Ctrl+S**, then **F7**. *Result:* no errors.



A Ladder program in the editor (conveyor example)

The file on disk, `pou\Agitator.st`, is an ordinary ST program: the diagram is kept in a `(*@LADDER ... @*)` comment at the top and the generated code is the body. For reference, this is the program the compiler sees:

```
PROGRAM Agitator
VAR
    AgitStart AT %IX0.4 : BOOL; (* agitator start, NO *)
    AgitStopOK AT %IX0.5 : BOOL; (* agitator stop circuit healthy, NC *)
    AgitRun AT %QX0.2 : BOOL; (* agitator contactor *)
END_VAR

(* Rung 1: Agitator start/stop with seal-in *)
AgitRun := (AgitStart OR AgitRun) AND AgitStopOK;

END_PROGRAM
```

Important: Change a Ladder POU's declarations with **Declarations...**, not by editing the text. The Ladder view rewrites both the body and the declarations from its diagram model each time it saves (Chapter 16).

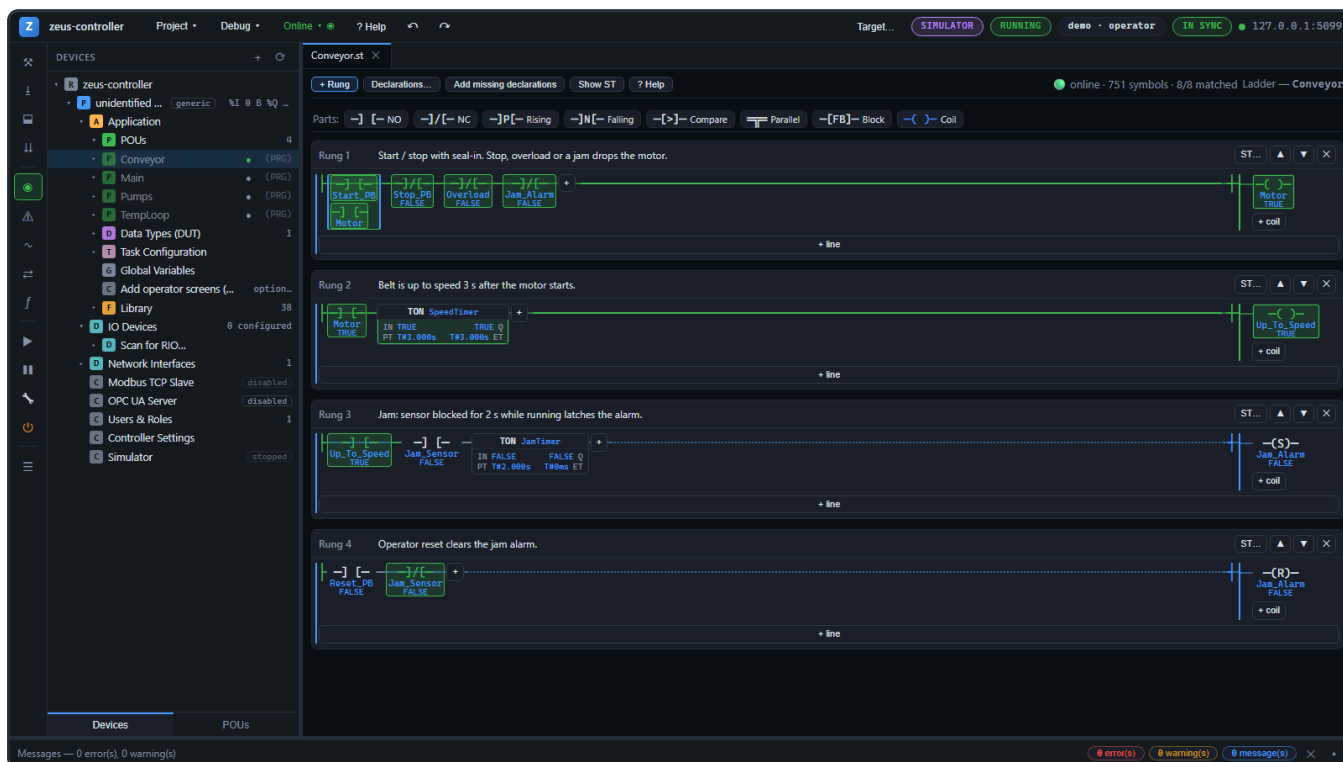
3.15 Step 13 — Run both programs

So far the controller runs one program. To run both, put them in the task's call list; a download then sends every program in the list as one application.

1. In the **Devices** tab, expand **Application** → **Task Configuration** and double-click **MainTask**. *Result:* the **MainTask** page opens. Under **POU calls** the table reads `No calls`. The downloaded program runs on its own.
2. In the drop-down next to **Add call**, select `FillTank.st` and click **Add call**; then select `Agitator.st` and click **Add call**. *Result:* the table lists `FillTank.st ( runs 1st )` and `Agitator.st ( runs #2 )`, and a toast says `Call`

list updated. The list is saved at once.

- Click the **Agitator** tab and press **F8**, then confirm. *Result:* the download report names the application; both **FillTank** and **Agitator** carry the **RUNNING** badge in the tree, and the sync badge reads **IN SYNC**. Because **FillTank** kept its name and variables, its data was carried over again.
- Make sure **Online** ▾ → **Monitor** is still on, and look at the **Agitator** tab. *Result:* the rung is animated: energised wires and conducting contacts are drawn green.
- Force **AgitStopOK** TRUE, then force **AgitStart** TRUE and release it (in **Agitator** → **Variables**, or by right-clicking the contact in the diagram and choosing **Set value...**). *Result:* **AgitRun** turns TRUE and stays TRUE through the seal-in contact after **AgitStart** is released. Release **AgitStopOK** and **AgitRun** drops.



The same Ladder program online, with live power flow

3.16 Step 14 — Finish

- Click the **FORCE** badge in the header and confirm, to release every remaining force. *Result:* the badge disappears.
- Click **Online** ▾ → **Monitor** again to leave online mode.
- Press **Ctrl+S**. While a controller is connected, **Save project** also writes the controller's current variable values to **data\saved.data.csv** (Chapter 5).
- Optional: **Project** ▾ → **Create archive...** packs the whole project into one **.tar.gz** file with a SHA-256 per file (Chapter 5).
- On the **Simulator** page, click **Stop**.

You have now used the whole engineering cycle — project, IO, program, build, download, monitor, write, force, online change, Ladder and the task call list. The same steps apply to a real controller; the differences are choosing it as the target and logging in to it (Chapter 6), and sending the hardware configuration with **Download hardware** (Chapter 20).

3.17 Where to go next

To ...	Read
Learn every part of the window	Chapter 4
Organise, copy and archive projects	Chapter 5
Connect to a real controller and manage accounts	Chapter 6
Configure real IO racks	Chapters 7–12
Learn the Structured Text dialect in full	Chapter 14
Understand download, online change and run states	Chapters 20–21
Use trace, breakpoints and diagnostics	Chapters 23–25

ZEUS PLC IDE · OPERATION MANUAL

Part II — The IDE

4. The IDE window

This chapter describes every area of the Zeus PLC IDE window: the menu bar and its menus, the status indicators, the tool strip, the device tree and what each node offers, the editor tabs and the **Messages** panel. Use it as a map when a later chapter names a menu item or a badge. The full keyboard reference is in Appendix E; the most used keys are summarised at the end of this chapter.

4.1 Layout



Area	Purpose
Menu bar	Project name, the Project , Debug and Online menus, ? Help , Undo/Redo, and on the right Target... and the status badges
Tool strip (far left)	One-click buttons for the most frequent actions. Each calls exactly the same command, with the same confirmations, as its menu entry.
Tree pane	The device tree: the project, the controller, the application, the IO and the services. Its Devices / POUs tabs switch the pane title and the + button.
Editor area	One tab per open object: POU editors, variable tables, device pages, settings pages
Messages panel	Build errors, download reports, runtime messages and notes, with click-to-jump

Drag the vertical splitter between the tree and the editor, and the horizontal splitter above the **Messages** panel, to resize them.

With no tab open, the editor area shows a welcome panel with **Add a device** and **New program** and a summary of the project: its folder, the number of devices and POUs, the process-image sizes (%I , %Q , %M), and whether the compiler is available.

4.2 The menu bar

From left to right:

Element	What it does
Z mark and project name	The name is the controller name from Controller Settings . Click it to open the Project dialog (same as Project ▾ → Open / switch project...).
Project ▾ , Debug ▾ , Online ▾	The menus (4.3–4.5). The menus are the authoritative list of commands; everything on the tool strip is also in a menu.
? Help	Opens the built-in help in a new window. The Ladder editor has its own ? Help .
 Check for updates	Asks optizeus.org whether a newer IDE release is out and answers in a window. While an update is waiting, the blue ↑ Update 0.1.x.y button is shown in its place (section 2.8.1).
 / 	Undo / Redo in the editor you are working in (the pane you last typed in). Same as Ctrl+Z / Ctrl+Y .
Target...	Opens the Controller target dialog: which controller the IDE talks to and downloads to (Chapter 6).
Status badges and connection	Section 4.6

Entries that do not apply at the moment are greyed out rather than hidden, for example **Log in...** while no controller is connected, or **Save As... (POU)** while the active tab is not a POU. **Controller ▶** and **Download & Upload ▶** in the **Online** menu are groups: click them to expand their entries in place.

4.3 Project menu

Item	Key	What it does	Chapter
Getting started tutorial...		A guided tour in a corner panel: nine steps from an example project to a program running in the simulator, online values, a force and the ladder view; each step ticks itself off	2
Open an example project...		Opens one of the worked examples of Appendix G; each runs in the simulator as it is	G
Open / switch project...		Opens the Project dialog: current project, Recent list, and Open , Create and Save as... for a typed folder path	5
Save project	Ctrl+S	Saves every unsaved program and the global variables. The device tree and rack need no saving: they are written as you edit them. While a controller is connected it also saves the controller's variable values to <code>data\saved.data.csv</code> .	5
Save As... (POU)	Ctrl+Shift+S	Saves the active POU under a new name in <code>pou\</code> . Enabled only on a POU tab.	5
Save project as...		Copies the whole project to a new folder and opens the copy	5
Create archive...		Packs the project into one <code>.tar.gz</code> with a SHA-256 per file	5
Restore archive...		Unpacks an archive into a new, empty folder and opens it	5
Import ST...		Imports programs from PLCopen XML or a text export (<code>.exp</code> / <code>.export</code>)	13

4.4 Debug menu

Item	Key	What it does	Chapter
Build	F7	Compiles the active POU with the globals, data types and library	20
Build All	Ctrl+F7	Compiles every POU that is not excluded from Build All; excluded POU's are listed as skipped	20
Clean		Clears the build output in the Messages panel. The compiler keeps no incremental cache, so there is nothing else to clean.	20
Clean all		Clears the build output and runs Build All	20
Step over	F10	At a breakpoint: run one statement and stop again	24
Step into	F11	At a breakpoint: run into the function block being called	24
Step out	Shift+F11	At a breakpoint: run until the current block returns	24
Continue	F5	Leave the breakpoint and run normally. F5 acts only while the program is halted; otherwise it keeps its usual meaning (reload the window).	24
Breakpoints...		Every breakpoint set on the controller	24
Verify		Asks the controller whether it runs exactly the program open here	20
Reset fault		Clears a controller fault; the program resumes on the next cycle. Address the cause first.	21
Force / write table...		A table of points to force (bits) or write (words) on the running controller	22
Clear forces		Releases every forced point after a confirmation	22

4.5 Online menu

Item	Key	What it does	Chapter
Controller ▸ Controller log...		Reads the controller's log without an SSH session	25
Controller ▸ Set clock / time...		Reads and sets the controller's date and time; sync from this PC or set by hand; writes the hardware RTC. Engineer only.	33
Controller ▸ Versions...		The IDE and runtime versions	33
Controller ▸ License...		Licensed or demo, the hardware id, install a license file	37
Controller ▸ Check for updates...		Asks optizeus.org whether a newer IDE release is out	2
Controller ▸ Run		Starts the scan	21
Controller ▸ Pause		Freezes the program; IO keeps running	21
Controller ▸ Config mode		Halts the controller for hardware work: outputs are no longer driven, inputs are still read. Leaving lands in PAUSED.	21

Item	Key	What it does	Chapter
Controller ▶ Restart runtime...		Restarts the runtime: briefly stops the control loop, reconnects drivers, rebuilds the OPC UA address space, reloads the boot program. Asks for confirmation.	33
Controller ▶ Reboot controller (OS)...		Reboots the whole controller (30–60 s down). Real controller only, engineer only.	33
Controller ▶ Shut down controller...		Saves retained data and powers the controller off; it stays off until power is restored. Real controller only, engineer only.	33
Download & Upload ▶ Download program	F8	Compiles and sends the application; the control loop keeps running (online change)	20
Download & Upload ▶ Download hardware		Sends the hardware configuration; stops the control loop while the runtime restarts	20
Download & Upload ▶ Download all		Hardware, then program; stops the control loop	20
Download & Upload ▶ Download screens		Sends the operator screens to the controller	32
Download & Upload ▶ Upload		Reads the running program (and the controller's data) back off the controller	5
Download & Upload ▶ Compare...		Compares program, hardware and scan period on the controller with this project	20
Monitor		Turns online mode on and off: live values beside declarations, animated Ladder and SFC	22
Alarms...		The controller's alarm list and journal	27
Sign...		Records an electronic signature in the controller's audit trail	28
Signatures...		The signatures the controller has taken since it started	28
Trace / scope...		Records up to 8 variables at scan rate and plots them	23
Save data snapshot...		Saves every variable's current value to a CSV file	23
Restore data snapshot...		Writes values from a CSV snapshot back to the controller	23
Log in...		Logs in to the connected controller as a named user	6
Log out		Drops the session. The controller keeps running.	6
Users & Roles...		Controller accounts and their roles	6

While online mode is on, the **Online ▼** menu title shows , so the state is visible with the menu closed.

4.6 Status badges and connection indicator

The right-hand end of the menu bar reports the state of the connection and of the controller. Hover over a badge for its full explanation.

Indicator	Shows	Meaning
Target badge	SIMULATOR	Connected to the local simulator
	CM4 • REAL IO / CM5 • REAL IO	A real controller whose IO devices use real drivers
	CM4 • SIM IO	A real controller whose devices all use the simulated driver — real hardware, fake IO. The case people misread; check it before commissioning.
	CM4 • MIXED IO	Some devices real, some simulated
	CM4 • NO IO	A real controller with no IO devices configured
Run-state badge	OFFLINE	Not connected. The tooltip lists board, kernel (PREEMPT_RT or not) and each driver's state.
	RUNNING, PAUSED, CONFIG, STOPPED, FAULT	What the controller's scan is doing (Chapter 21)
	OFFLINE	No connection
	NO SESSION	Connected, but this connection is not logged in, so the controller refuses every request. Click it to log in.
Session badge	admin • engineer	Who this IDE is logged in as, and the role. Hidden when there is no session. Everything you do is recorded against this name.
DEMO badge	DEMO 1:42:10 (amber), DEMO ENDED (red)	The runtime has no Zeus license and uses real IO: the time until the program stops, or that it has. Hidden on a licensed controller and with simulated IO only. Click for License... (Chapter 37).
FORCE badge	FORCE 3 points (amber)	Points are forced on the controller — including forces set from another IDE or left from an earlier session. Hidden when nothing is forced. Click to release every force.
Sync badge	IN SYNC	The controller runs exactly the program (the whole call list) and hardware configuration of this project
	PROGRAM CHANGED	The program open here — including unsaved edits — differs from the one running. Live values on screen belong to the running program. Normal while editing; download to make them agree.
	HARDWARE CHANGED	The controller runs a different hardware configuration; download the hardware
	PROGRAM + HARDWARE CHANGED	Both
	⚠ NOT CHECKED — LOG IN	The controller has accounts and this IDE is not logged in, so nobody has checked. This is not a match. Click to log in.
Connection dot and text	⚠ NOT CHECKED	The check could not be completed; the tooltip says why. Click to try a full comparison.
	green dot, 192.168.1.50:5020	The IDE server is connected to the controller at that address (for a tunnelled target, the controller's own address is shown)

Indicator	Shows	Meaning
	connecting... / disconnected – <reason>	The IDE server has no connection to the controller; it retries by itself unless you disconnected deliberately
	no link to the IDE server	The window has lost its connection to the IDE server itself. Nothing on the page is being updated. It reconnects by itself; if not, the IDE server has stopped (Chapter 2).

The sync badge is re-checked shortly after every edit, on switching tabs, after every download, and every 30 seconds as a backstop for changes made from another IDE. It is hidden only when no controller is connected. Clicking **IN SYNC** or **... CHANGED** opens the **IDE compared with the controller** dialog with the item-by-item detail.

4.7 The tool strip

The vertical strip at the far left holds the actions used most often, in four groups:

Button	Action	Same as
	Build the active POU	Debug → Build (F7)
	Download program	Online → Download & Upload → Download program (F8)
	Download hardware — stops the control loop	Online → Download & Upload → Download hardware
	Download hardware and program	Online → Download & Upload → Download all
	Monitor on/off (highlighted while online)	Online → Monitor
	Alarms	Online → Alarms...
	Trace / scope	Online → Trace / scope...
	Compare with the controller	Online → Download & Upload → Compare...
	Function block library	Library node in the tree
	Run	Online → Controller → Run
	Pause	Online → Controller → Pause
	Config mode (highlighted while in CONFIG)	Online → Controller → Config mode
	Clear every force	Debug → Clear forces
	Controller log	Online → Controller → Controller log...

A strip button is greyed out whenever its menu entry is, because it simply presses the menu entry.

4.8 The device tree

4.8.1 Structure

The tree is a picture of the project and the controller it is for:


```

TankFill                                project (controller name)
├ IPC-CM400200                          cm4  %I 1 B  %Q 1 B
│ └ Application
│   └ POU's                             3
│     └ motors                           folder
│       └ PumpCtl                        (FB)
│         └ Variables
│           └ Start                      method
│             └ FillTank                  (PRG) • RUNNING
│               └ Variables
│                 └ Batch1                (PRG)
│                   └ Variables
│                     └ Chart              SFC view
│ └ Data Types (DUT)                     2
│ └ Task Configuration
│   └ MainTask                           10 ms
│ └ Global Variables                     (globals.st + further lists)
│ └ Add operator screens (HMI)...         (or HMI, once added)
│   └ Library                            12 • default
├ IO Devices                             1 configured
│ └ rack1                                async • eth0  OPERATIONAL
│   └ 1 CT-1108                          digital input  %IX0.0 ×8
│     └ 2 CT-2108                          digital output %QX0.0 ×8
│   └ Scan for RIO...
├ Network Interfaces                     2
├ Modbus TCP Slave                       disabled
├ OPC UA Server                          disabled
├ Users & Roles                           1
├ Controller Settings
└ Simulator                             stopped

```

Click ► / ▼ to expand or collapse a branch. Click a node to select it and open it; double-click also opens it. Right-click a node for its menu (4.8.3). The ↻ button at the top of the pane reloads the project from disk.

The **Devices** and **POUs** tabs at the bottom of the pane switch the pane title and what the + button creates: **Add device** in **Devices**, **New POU** in **POUs**. **POUs** also expands and scrolls to the **POUs** branch.

4.8.2 Node types

Node	Opens	Notes
Project (top)	—	Labelled with the controller name
Controller	—	Labelled with the board model the controller reports (or Zeus IPC-CM400200 when offline); badge with the board id; the %I / %Q sizes of the project
Application	—	Right-click holds every "add" command for the application
POUs	—	The programs, function blocks and functions, in folders. The count is the number of top-level entries. Drop a dragged POU here to move it out of any folder.
Folder	—	A real folder under pou\ . Its dot summarises its POUs (4.8.4). Drop a dragged POU on it to move the POU in.

Node	Opens	Notes
POU	The editor (ST, Ladder or SFC)	Meta text (PRG) , (FB) , (FC) , or (excl uded) ; status dot; RUNNING badge for programs the controller is running; programs not in the running set are dimmed
Variables (under a POU)	The online variable table of that POU (Chapter 22)	
Chart (under an SFC POU)	The animated chart view (Chapter 17)	Only for a POU whose body is a chart
Method (under an FB)	The method's own editor tab	Methods are hidden from the FB's tab and edited only in their own tabs (Chapter 13)
Data Types (DUT)	New DUT dialog when empty	Children: one node per data type, with its kind (struct , enum , alias) (Chapter 15)
Task Configuration	The Task Configuration editor: task groups, cores, monitor, variable and stack usage, events, properties (Chapter 18)	
MainTask	The task page: priority, interval, CPU affinity, overrun policy, watchdog, POU calls	Meta: the task interval
Global Variables	globals.st	With further lists, expands to globals.st and one node per list in gv1\ (Chapter 19)
Add operator screens (HMI)... / HMI	Adds the HMI section / its branch: Screens, Alarms, Events, Server	Chapter 32
Library	The Function block library dialog	Children: one node per library block with its input and output counts; meta default when the project uses the package's library rather than its own (Chapter 5)
IO Devices	—	Every configured device, whether or not it has ever connected (Chapter 7)
Device	The device page	Badge: the driver state (OPERATIONAL , DEGRADED , FAULT , STARTING ...), or disabled , inhibited , not downloaded . Meta: exchange mode, network port, uptime. A device that is configured but not exchanging blinks while online.
Card / block	The device page	Slot, card type, what it is (for example digital input) and its addresses; live while exchanging
Scan for RIO...	—	Children: Scan for remote IO... (Modbus TCP), Scan for EtherNet/IP... , Scan EtherCAT segment... , Scan Profinet network... , Add EtherCAT rack from ESI... (Chapters 8–10)
Network Interfaces	—	The controller's network ports, as the connected controller reports them; not reported – controller not connected offline

Node	Opens	Notes
Network port	The port page (address, link, speed)	Badge <code>link</code> , <code>no cable</code> or <code>down</code> . Children: Remote IO reached through this port, and Listening here — the servers that answer on it, marked <code>all ports</code> when bound to <code>0.0.0.0</code>
Modbus TCP Slave	Its settings page	Badge <code>disabled</code> , <code>read-only</code> or <code>writable</code> (Chapter 31)
OPC UA Server	Its settings page	Chapter 29
Users & Roles	The Users & Roles dialog	Meta: number of accounts, or <code>none</code> (Chapter 6)
Controller Settings	The controller settings page	Chapter 6
Simulator	The Simulator page	Badge <code>running</code> / <code>stopped</code> (Chapter 26)

The IO-Link ports of an EtherCAT IO-Link master are not tree nodes; they are reached with the **IO-Link ports (N)...** button on the device page (Chapter 12).

4.8.3 Right-click menus

Node	Menu entries
Application	New POU... , Import ST file... , Upload from controller... (needs a connection), Global Variables , New global variable list... , New DUT (data type)... , Add operator screens (HMI)... (or New screen... and Remove operator screens (HMI) once added), Users & Roles...
POUs and each folder	New POU here... , New sub-folder... , Import ST file...
POU	Open , Open Variables , Open Chart (SFC only), Add method... (FB only), Change to FUNCTION... (FB) or Change to FUNCTION_BLOCK... (FC), Save (Ctrl+S), Save As... (Ctrl+Shift+S), New POU here... , Import ST file... , Build (F7), Download to controller (F8), Upload from controller... , Rename... , Disable / Enable , Delete
Method	Open , Rename method... , Delete method...
Data Types (DUT)	New DUT...
Data type	Open , Rename... , Delete
Further global variable list	Open , Rename... , Delete
HMI	New screen... , Download screens to controller , Remove HMI from this project
IO Devices	Opens the Add Device dialog directly
Device	Asks Remove device "<name>" and all its modules? — OK removes it

Important: Right-clicking a device does not open a menu; it asks at once whether to remove the device. Answer **Cancel** unless you mean it. Removing a device shifts the addresses of every device after it; to take a device out of service without moving addresses, **inhibit** it on its device page instead (Chapter 7).

Change to FUNCTION... is refused when the block depends on per-instance memory (for example it declares a `TON`), because a FUNCTION has none; the message names what to change.

4.8.4 POU status dots

While a controller is connected, each POU carries a dot comparing it with what the controller runs:

Dot	Meaning
Green	Identical to the controller
Orange	Differs from the controller, or has unsaved edits — download to update it
Grey	Not on the controller — download to send it
Split green/orange (folder)	Some POUs in the folder match, some do not

Offline there is nothing to compare with, and the dot shows only unsaved edits (amber).

4.8.5 Moving POUs between folders

Drag a POU onto a folder to move it there, or onto **POUs** to move it to the top level. A POU's name is its path under `pou\`, so a moved POU is renamed — for example `FillTank.st` becomes `tanks/FillTank.st` — and that new name is what the task call list, the controller and the cross-reference use (Chapter 5).

4.9 Tabs and editors

Every object opens in a tab. The tab shows the object's title and, when it has unsaved changes, a changed mark; **X** closes it. Closing a tab with unsaved changes asks first, and closing the IDE window with unsaved tabs makes the browser ask whether to leave the page.

Tab kind	Opened from	Chapter
ST POU	POU node	13, 14
Ladder POU	POU node of a Ladder program	16
Method	Method node	13
Chart (SFC)	Chart node	17
<POU> · Variables	Variables node	22
DUT, Global Variables, further GVLs	Their nodes	15, 19
Device, network port	Their nodes	7–12
MainTask, Task Configuration	Their nodes	18
Controller Settings, OPC UA Server, Modbus TCP Slave, Simulator, HMI pages	Their nodes	6, 26, 29–32
Trace / scope, Alarms	Online menu	23, 27

The ST editor has two panes — declarations above, body below — over one file on disk, and a row of buttons above them: , , **Find...**, **Rename variable...**, **Rename POU...**, **Where used...**, and for a function block with

several instances an **Instance** picker that selects which instance's values are shown online. Right-click in the code for **Where used...**, **Find in this POU**, **Rename variable...**, **Cut**, **Copy** and **Paste**. Hovering over a variable shows its kind, type, initial value, comment, where it is declared and, online, its live value.

4.10 The Messages panel

The panel at the bottom collects everything the IDE and the controller report.

Part	Use
Title	Messages – N error(s), N warning(s), N message(s)
error(s) / warning(s) / message(s) chips	Show or hide each severity. Each chip shows its count.
✕	Clear the list
▾	Collapse or expand the panel
Description	The message, with ✕ (error), ⚠ (warning) or i (note)
Object	The file (POU, <code>globals.st</code> , a GVL, a library file) and, for a method, the method name
Position	Line L, Column C
Source	Where the message came from: build, build all, download, runtime, simulator, online, save, project, target, compare, users, ide ...

Severity	Used for
Error	A compile error, a refused download, a failed command, a runtime error message
Warning	Something that works but needs attention: a minor fault on the controller (for example integer division by zero — the scan kept running), a hardware configuration that differs, a force set, a default password, online with differences
Note (message)	Progress and results: build summaries, download reports, online-change results, connections

Click a message to jump to it. A message with a line opens the file at that line and column — including a line in `globals.st`, a global variable list or a method. A message without a line but naming a variable opens the POU at that variable's declaration.

Build and download replace the list with their own results; runtime messages, connection events and command results are added to it. An internal error of the user interface is reported here with the source `ide` rather than disappearing silently.

Short confirmations (Downloaded – controller RUNNING, Logged in as ...) also appear briefly as a **toast** at the bottom of the window; anything that matters is also written to the **Messages** panel.

4.11 Dialogs

Dialogs open over the window. **Esc** or ✕ closes a dialog without applying it. Actions that change a running machine — download, hardware download, run-state changes, clearing forces, restart, reboot, shutdown — always ask for confirmation in words that say what happens to the machine.

4.12 Keyboard summary

The most used keys. Appendix E has the complete list.

Key	Action
Ctrl+S	Save project
Ctrl+Shift+S	Save As... (active POU)
F7	Build the active POU
Ctrl+F7	Build All
F8	Download program
Ctrl+Z / Ctrl+Y (or Ctrl+Shift+Z)	Undo / Redo in the current editor, multi-step
Ctrl+F	Find in this POU; in a variable table, focus the filter
F3 / Shift+F3	Next / previous match
F2	Rename the variable at the caret
Shift+F12	Where used
Ctrl+Space	In the body pane of the code editor: the completion list only. Elsewhere: Browse IO, to insert a declaration
Tab	Indent (four spaces) in the code editor; accepts a completion when the list is open
Alt+drag	Column (block) selection in the code editor
F10 / F11 / Shift+F11	Step over / into / out at a breakpoint
F5	Continue from a breakpoint (only while halted)
Esc	Close the open dialog, or the find bar
In a variable table: Ctrl+F7 / F7 / Alt+F7	Write prepared values / Force prepared values / Release forces on the rows shown

Note: F7 and Ctrl+F7 mean **Force** and **Write** only while a variable table is visible in the active tab, and then they do not also build; everywhere else they are **Build** and **Build All**.

5. Projects, archives and libraries

This chapter explains what a Zeus PLC project is on disk, how to create, open, copy and archive one, how to bring a program back from a controller, where the IDE keeps backups, and how the project's function-block library is versioned and locked. You need it whenever you start, hand over, restore or validate a project.

5.1 What a project is

A project is a **directory**, not a file. It holds everything a controller runs and everything the IDE needs to show it: the IO configuration, the programs, the data types, the global variables, the library, the operator screens and saved controller data. There is no separate project file to keep in step with the folders: the filesystem is the structure.

Because every part is a plain text file (`.st` for code, `.json` for configuration and screens), a project can be kept in a version-control system, compared with ordinary diff tools, and reviewed line by line — which is what a validated system needs.

A project is engineered once and may run on several controllers: the simulator on your desk, the controller on the line, a second identical line. The project does not know which controller it is on; the **target** does (Chapter 6).

5.2 What is on disk

```
C:\ZeusProjects\TankFill\
controller.json      hardware and settings: IO devices, task, servers, accounts, alarms
globals.st           the default global variable list (VAR_GLOBAL ... END_VAR)
pou\
  FillTank.st        one .st file per POU; sub-folders are POU folders
  Agitator.st         a Ladder POU is an .st file too
  tanks\Buffer.st
dut\
  Pump_T.st          one .st file per user data type (STRUCT, ENUM, alias)
gvl\
  Utilities.st        further global variable lists, one .st file each
lib\
  10_signal.st ... 70_iolink.st the function-block library compiled into every POU
hmi\
data\
uploads\
backups\
library.lock.json    the library version this project is locked to (after a lock)
```

Item	Written by	Content	Chapter
<code>controller.json</code>	The device tree, settings pages, Users & Roles , the task pages	The controller configuration: <code>io</code> (devices and racks), <code>task</code> (interval, priority, call list), <code>process_image</code> , <code>log</code> , <code>monitor</code> (port, accounts as salted hashes), <code>opcua</code> , <code>modbus_slave</code> , <code>enip_tags</code> , <code>web</code> , <code>hmi</code> , <code>alarms</code> , <code>redundancy</code> , and <code>pou.disabled</code> (POUs excluded from Build All)	Appendix H
<code>globals.st</code>	Global Variables tab	<code>VAR_GLOBAL</code> blocks; also where IO aliases are written. Prepend to every POU at build and download.	19

Item	Written by	Content	Chapter
<code>pou*.st</code>	POU editors	One PROGRAM, FUNCTION_BLOCK (with its METHODS) or FUNCTION per file. The file name without <code>.st</code> is the POU's name; its path under <code>pou\</code> is its full name.	13
<code>dut*.st</code>	DUT editor	One data type per file, linked before the globals	15
<code>gvl*.st</code>	GVL editor	Further global lists, prepended after <code>globals.st</code> in name order	19
<code>lib*.st</code>	Library (edited as files)	Function blocks compiled with every POU, in file-name order	5.9, Appendix C
<code>hmi*.json</code>	Screen editor	Operator screens	32
<code>data*.csv</code>	Save project, Upload, Save data snapshot...	Controller variable values	5.7, 23
<code>uploads*.st</code>	Upload	The linked program the controller was running	5.6
<code>backups\<time>-<reason>\</code>	The IDE, before an operation that overwrites files	A copy of the project's source and configuration	5.8
<code>library.lock.json</code>	Library versions → Lock this library	The locked version and SHA-256 of every library file, who locked it, when and why	5.9

The build combines these files into one compilation unit in a fixed order: the data types, `globals.st`, the generated IO aliases, the further global lists, the library (`lib\` or the default library), the project's own function blocks and functions, and finally the program. A download does the same for every program in the task's call list, in call-list order, and sends the result as one unit.

Note: The device tree and the rack are written to `controller.json` as you edit them. **Save project** saves programs and global variables; there is nothing to save for the hardware.

5.3 Creating, opening and switching projects

To create a new project: **Project** ▾ → **Create new project....** Type a **Name** (letters, digits, `_`, `-`, `.`); **Folder** starts as the folder that holds the current project, and **Will be created** shows the full path. **Create and open** makes the folder, fills it with an empty controller configuration and the standard library, and switches the IDE to it. Unsaved editors in the current project are saved first. A folder that already has files in it is refused, so an existing project cannot be overwritten from here.

Opening and switching are done in the **Project** dialog: **Project** ▾ → **Open / switch project...**, or click the project name in the menu bar. The same dialog can also create a project from a full path.

Part of the dialog	Use
Current	The folder of the open project (read-only)
Recent	Projects opened before, most recent first. Click a row to open it. A folder that no longer exists is marked <code>missing</code> .

Part of the dialog	Use
Open, create or copy	A box for a full folder path, with Open , Create and Save as...

Paths must be **absolute**, for example `C:\ZeusProjects\TankFill` or `\\fileserver\automation\line2`.

To create a project:

1. Open the **Project** dialog.
2. Type the path of a new folder, or of an existing empty one.
3. Click **Create**.

Create refuses a folder that already contains files, so a project can never be scattered through an unrelated folder. A new project contains a default `controller.json` (task interval 10 ms, %M 4096 bytes, monitor on port 5020 bound to loopback, OPC UA disabled, no IO devices, and one account `admin` with the engineer role and the initial password `admin`, which must be changed before any download — Chapter 6), empty `pou\` and `dut\` folders, and its own copy of the standard library in `lib\`.

To open a project: type its folder path and click **Open**, or click it in **Recent**. **Open** accepts any existing folder and fills in what is missing (`controller.json` , `pou\` , `dut\` , and the library copy when it creates `controller.json`). Use it to open a project checked out from version control or copied from a colleague.

When you switch projects, every open tab closes, the tree reloads, and the **Messages** panel notes `Project switched to ...`. The IDE opens the most recently used project again at its next start.

Important: Switching the project changes the IDE only. The controller keeps running the program and configuration it has. After switching, the sync badge tells you whether the new project matches the connected controller; download to make it so.

5.4 Saving

Command	Key	What it does
Save project	Ctrl+S	Writes every unsaved POU and global variable list. While a controller is connected, also writes the controller's current variable values to <code>data\saved.data.csv</code> (overwritten each time). With nothing unsaved it says so and still saves the data.
Save As... (POU)	Ctrl+Shift+S	Saves the active POU under a new name in <code>pou\</code> . Only on a POU tab: the globals, the task and the screens are single project files that cannot be copied this way.
Save (POU right-click)	Ctrl+S	The same as Save project , from the tree

Saving is not downloading. If you save while online and the saved program no longer matches the controller, the IDE says so explicitly: the live values on screen still belong to the old program.

5.5 Save project as, and archives

5.5.1 Save project as

Project ▾ → **Save project as...** opens the **Project** dialog with a suggested sibling folder (`<name>-copy`) in the path box. Edit the name and click **Save as...**

1. Unsaved editors are saved first, after a confirmation, so the copy contains the change you were making.
2. The project is copied to the new folder: `controller.json` , `globals.st` , `pou\` , `dut\` , `lib\` , `gv1\` , `hmi\` and `data\` .
3. The copy is opened; the original is left as it is.

`backups\` and `uploads\` stay with the original: they are the history of the original project and of the controller it was talking to. The target folder must be new or empty and must not lie inside the project being copied.

Important: In this release **Save project as...** does not copy `library.lock.json` . The copy therefore shows its library as **not locked**. If the project is validated against a locked library, copy `library.lock.json` into the new folder by hand, then check **Library versions** in the copy (5.9).

5.5.2 Create archive

Project ▾ → **Create archive...** packs the project into one file for a handover, a validation file or an off-site copy.

1. If editors are unsaved, the IDE offers to save them. **OK** saves and archives; **Cancel** archives what is on disk without them.
2. The archive is offered as a download named `<project>-<date>T<time>.zeus.tar.gz` , for example `TankFill-2026-09-27T14-05-12.zeus.tar.gz` , and saved to your Downloads folder.
3. The **Messages** panel notes how many files were archived and the size.

In the archive	Not in the archive
<code>zeus-archive.json</code> (the manifest, first in the file), <code>controller.json</code> , <code>globals.st</code> , <code>pou\</code> , <code>dut\</code> , <code>lib\</code> , <code>gv1\</code> , <code>hmi\</code> , <code>data\</code> , <code>uploads\</code>	<code>backups\</code> (usually most of the size, and history rather than the project); <code>library.lock.json</code> in this release

The manifest records the project name, the source folder, the time, the IDE version and a SHA-256 and size for every file. The archive is a standard gzip-compressed tar file: `tar xzf` or 7-Zip opens it years from now without this IDE.

5.5.3 Restore archive

Project ▾ → **Restore archive...** opens the **Restore project archive** dialog.

1. Under **Archive file**, choose the `.tar.gz` file.
2. Under **Restore into**, type the full path of a new or empty folder.
3. Click **Restore**.

The archive is read and checked **before anything is written**. Every file is compared with the SHA-256 recorded in its manifest; the **Messages** panel then reports either `Every file matched its recorded hash.` or the files that did not match — treat such an archive as damaged. The restored project is opened.

An archive never restores over an existing project: a folder that contains anything is refused. To go back to an archived state of a project you are working on, restore into a new folder and compare.

5.6 Uploading from a controller

Upload reads back what a controller is running. Use it when the project on your PC is missing, out of date, or when you need evidence of what a machine runs.

Online ▾ → **Download & Upload** → **Upload**, or right-click **Application** or a POU → **Upload from controller...**

The controller must be connected; the **viewer** role is enough.

1. A confirmation shows the running program's name, size, when and by whom it was downloaded, and its SHA-256.
2. **OK** saves:
 - the program as `uploads\uploaded_<name>.st` — a **read-only snapshot**, never placed in `pou\`;
 - the controller's current variable values as `data\<time>-upload_<name>.data.csv`.
3. The **Running program on the controller** dialog shows the text and where both files were saved.

The snapshot is the whole linked unit the controller compiled — library, data types, globals, every function block and the program in one file. That is why it is kept out of `pou\`: in `pou\` it would be compiled into every build and duplicate every definition in it.

Replacing the project from the controller. A program downloaded from this IDE carries a file map. If it has one, the dialog offers **Replace this project...**, which splits the unit back into the files it came from:

1. A confirmation lists every file that will be **overwritten**, and whether `controller.json` (hardware and settings) will be replaced too.
2. The whole project is copied to `backups\<time>-before-upload\` first.
3. Every part goes back to its own place: programs and function blocks to `pou\`, data types to `dut\`, global variable lists to `globals.st` and `gv1\`, library files to `lib\`. In `globals.st` the channel addresses the build had bound in are taken out again, so the next build binds them from the hardware configuration as before.
4. Library, data-type and variable-list files that the controller's program did **not** contain are removed (they are in the backup) — each of them is part of every build, so one extra file would stop the project from rebuilding the program the controller runs. Programs and function blocks you added are never removed.
5. The project reloads, and the **Messages** panel names the files, the backup folder, and whether the project now rebuilds exactly the program the controller is running (**IN SYNC**).

From a real controller, its configuration is restored as well, so the hardware comes back with the program. From the **simulator** only the programs are restored: the simulator runs a transformed configuration (every device replaced by the simulated driver), which is not your rack.

A program downloaded before file maps existed, or by another tool, can only be kept as a snapshot; the dialog says so.

5.7 Saved controller data

Source code alone does not bring a machine back: setpoints, recipe parameters and tuning values live in the running controller. The IDE therefore saves them with the project:

When	File
Save project while connected	data\saved.data.csv (overwritten each time)
Upload	data\<time>-upload_<program>.data.csv (kept)
Online → Save data snapshot...	A CSV file you name

Online → Restore data snapshot... lists the files in `data\`, newest first, and writes values back to a controller. Only values the program reads but does not write will hold; see Chapter 23.

5.8 Backups

The IDE copies the project aside before any operation that overwrites or deletes source you may not have anywhere else:

Operation	Backup folder
Replace this project... after an upload	backups\<time>-before-upload\
Renaming or deleting a further global variable list	backups\<time>-rename-gvl-<name>\ , ...-delete-gvl-<name>\
Renaming or deleting a data type	backups\<time>-rename-dut-<name>\ , ...-delete-dut-<name>\

A backup contains `pou\`, `lib\`, `dut\`, `gvl\`, `globals.st`, `controller.json` and `data\`. The **Messages** panel names the folder. To go back, copy the files you need from it into the project while the project's tabs are closed, then click  above the tree.

Note: Deleting a POU (right-click → **Delete**) takes no backup and cannot be undone from the IDE. Keep projects under version control, or archive before large changes.

Backups are never deleted automatically and are not included in archives or project copies. Delete old ones by hand when they are no longer needed.

5.9 The library

5.9.1 How the library works

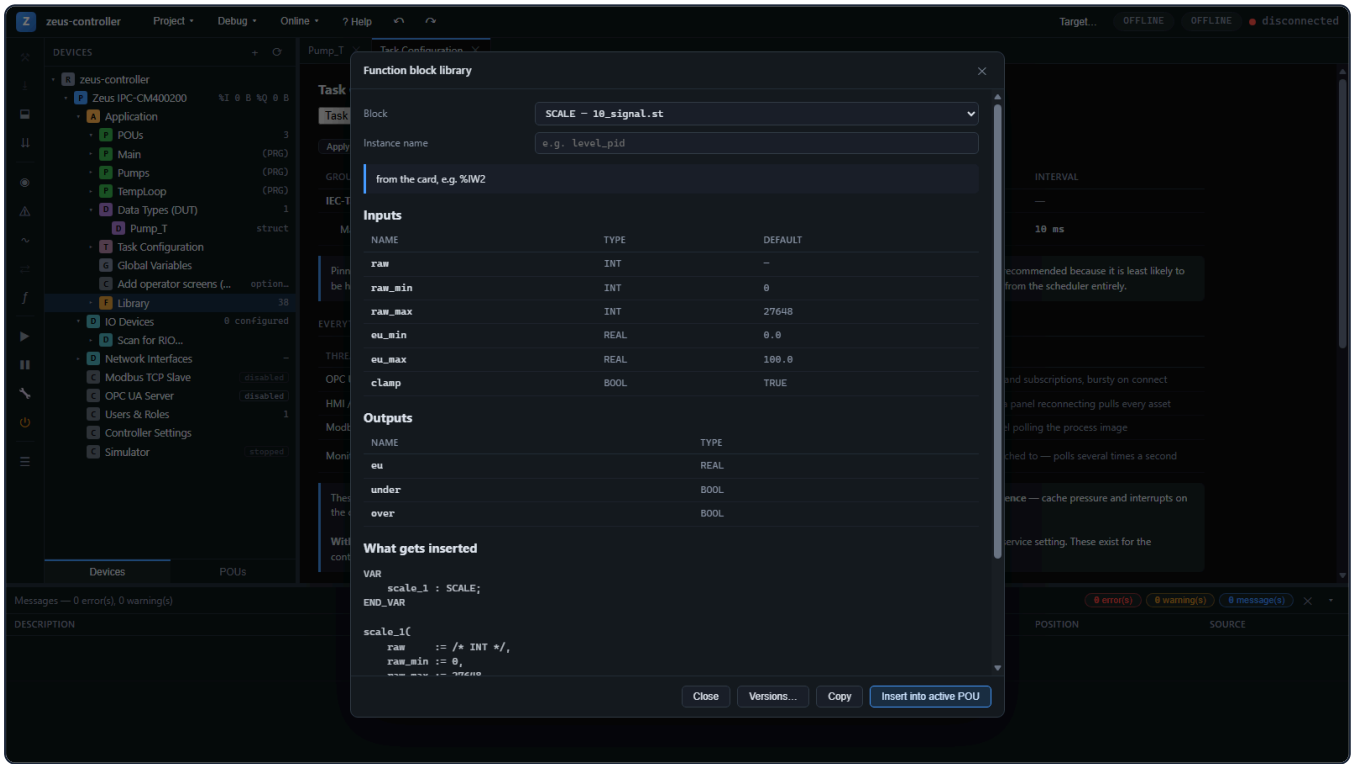
The library is the set of function blocks compiled ahead of every POU, so any POU can declare an instance of `PID`, `SCALE`, `MOTOR_DOL` or an IO-Link block without further declarations. It lives in the project's `lib\` folder as ordinary `.st` files.

Rule	Consequence
Each project has its own copy of the library in <code>lib\</code>	Retuning or fixing a block in one project changes only that project
A project without <code>lib\</code> , or with an empty <code>lib\</code> , uses the default library (<code>stdlib\</code> in the IDE package)	The Library node shows <code>• default</code> . Such a project changes when the IDE's default library changes.

Rule	Consequence
Files are compiled in file-name order	A block that uses another block must be in a file that sorts later. The standard files are numbered (<code>10_signal.st</code> , <code>20_control.st</code> ...) for this reason.
An unused block costs compile time, not program size	Keep the library complete rather than trimming it per project

The standard library's blocks are described in Appendix C. The **Library** node in the tree lists every block with its number of inputs and outputs; clicking a block, or **f** in the tool strip, opens the **Function block library** dialog:

Control	Use
Block	Choose the block. The dialog shows its summary, its Inputs (name, type, default) and Outputs .
Instance name	The name to declare, for example <code>level_pid</code>
Insert into active POU	Adds the instance declaration and a call with every input named to the active POU
Copy	Copies the declaration and call to the clipboard
Versions...	Opens Library versions (5.9.3)



The function block library dialog

5.9.2 Adding your own library file

A library file is ordinary Structured Text containing one or more FUNCTION_BLOCKS (or FUNCTIONS). Put a version header in a comment in its first 40 lines, so the version can be tracked:

```

(* @Library site-utils @version 1.0.0 *)
(* Site utility blocks - owned by the plant engineering team. *)

FUNCTION_BLOCK RunHours
VAR_INPUT
    running : BOOL;      (* the equipment is running *)
    reset   : BOOL;      (* clear the counter *)
END_VAR
VAR_OUTPUT
    hours   : REAL;      (* accumulated running hours *)
END_VAR
VAR
    seconds : DINT;      (* whole running seconds *)
    ms      : DINT;      (* running milliseconds not yet a whole second *)
    last    : TIME;      (* scan clock at the previous call *)
    now     : TIME;
    started : BOOL;      (* FALSE until the first call has set 'last' *)
END_VAR

now := TIME_NOW();
IF started AND running THEN
    ms := ms + TIME_TO_INT(now - last);
    IF ms >= 1000 THEN
        seconds := seconds + ms / 1000;
        ms := ms MOD 1000;
    END_IF;
END_IF;
last := now;
started := TRUE;

IF reset THEN
    seconds := 0;
    ms := 0;
END_IF;
hours := DINT_TO_REAL(seconds) / 3600.0;
END_FUNCTION_BLOCK

```

Save it as `lib\80_site_utils.st` (the `80_` prefix sorts it after the standard files, so it could use their blocks). Any POU can then use it:

```

PROGRAM PumpHours
VAR
    PumpRunning AT %IX0.0 : BOOL;
    HoursReset  : BOOL;
    P101Hours   : RunHours;
    P101Total   : REAL;
END_VAR

P101Hours(running := PumpRunning, reset := HoursReset);
P101Total := P101Hours.hours;
END_PROGRAM

```

Files are read when the project loads; after adding a file with Explorer, click  above the tree so the **Library** node and the completion lists include it. A file without a header is still compiled and still hashed by the version lock; it just shows no version.

Tip: Library blocks keep state per instance. Put blocks that are shared between several projects under version control of their own and copy them into each project deliberately — the version lock (below) then tells you exactly which copy each project runs.

5.9.3 Library versions and the lock

"Which version of the PID block did this batch run with?" is a question a validated system must answer, and a folder of files cannot. The **Library versions** dialog answers it: it compares the library as it is now with the state the project was **locked** to.

Open it with **Versions...** in the **Function block library** dialog.

Part	Content
Lock line	Locked <date> by <user> – <reason> , or a warning that the project has never been locked
Available files	The default library files this project does not have yet, with Add to this project
Drift warning	The library is not what this project was locked to. when any file is CHANGED, added or REMOVED
Table	File, Library (the @library name), Version (the @version now), Locked at (the version when locked), State
Why are you locking this?	The reason recorded with the lock
Lock this library / Lock again	Takes the lock

State	Meaning	What to do
unchanged	Same content (same SHA-256) as when locked	Nothing
CHANGED	The content differs from the lock — whether or not the version number changed. The code that will be downloaded is not the code that was validated.	Restore the file, or review the change and lock again with a reason
added	A file that was not there when the lock was taken	Review and lock again
REMOVED	A locked file is gone. Programs that use its blocks will not compile.	Restore it, or remove its use and lock again
not locked	No lock has been taken yet	Lock once the library is the one you intend to validate

The hash is what makes the version trustworthy: a version number is a claim, and an edited file that kept its number is exactly the case worth catching. Line endings are normalised before hashing, so the same file checked out on Windows and on Linux is not reported as changed.

To lock the library:

1. Open **Library versions**.
2. Check that every file is the one you intend to run.

3. Type the reason in **Why are you locking this?**, for example `validated for the 2026-09 batch campaign, change CC-114`.
4. Click **Lock this library** (or **Lock again**). *Result:* `library.lock.json` records every file's version and SHA-256, the time, the reason, and the logged-in user (or `unknown` when no session exists). A toast reports the number of files locked, and the lock is noted in the **Messages** panel.

Locking is always an explicit act; a lock never updates itself when a file changes.

Adding new standard blocks. A project with its own `lib\` never receives blocks added to the IDE's default library later — nothing should change under a validated project by itself. When a new IDE version brings new library files, **Library versions** lists them as available. Click **Add to this project** to copy them into `lib\`. Existing files are never overwritten, so a block you have edited stays as it is. On a locked project the new files show as `added` until you lock again.

Important: A library change takes effect on the controller only with the next program download, which sends the library with the program. After locking, download, then check that the sync badge reads `IN SYNC`.

5.10 Folders in the POU tree

Folders group POU's, for example by machine area (`filling`, `cip`, `utilities`). A folder is a real sub-folder of `pou\`; nesting is allowed (`motors/pumps`).

To create a folder:

- In the **New POU** dialog: type a name such as `motors` or `motors/pumps` in the box below **Folder** and click **Create folder**, then choose it in **Folder**; or
- right-click **POUs** or an existing folder → **New sub-folder...**

To create a POU in a folder: right-click the folder → **New POU here...**, or choose the folder in the **New POU** dialog. **Import ST file...** on a folder imports into that folder.

To move a POU: drag it onto a folder, or onto **POUs** to move it to the top level. If the target already has a POU of that name, the move is refused.

A POU's name is its path: `FillTank.st` moved into `tanks` becomes `tanks/FillTank.st`. The file is moved on disk, its exclusion from Build All (if any) moves with it, and the **Messages** panel states the new name.

Moving or renaming a **PROGRAM** also renames its entry in the task call list, in place, so it keeps its position and still runs; a program's own rate (**Task Configuration**) follows a rename of its declaration header.

Important: Moving or renaming a POU does not update the POU's that use it. After renaming a **FUNCTION_BLOCK** or **FUNCTION**, update the instances and calls that use its old name (**Where used...** finds them). Moving a block into a folder does not change its type name, so its users need no change.

5.11 Renaming, excluding and deleting POU's

Action	How	What happens
Rename	Right-click → Rename... , or Rename POU... above the editor	The file is renamed and the header (<code>PROGRAM FillTank</code>) is changed to the new name, because that is the name the controller reports. If the header did not match the old file name, only the file is renamed and a warning says so. The task call list entry, the Build All exclusion and the program's rate follow the new name.
Disable	Right-click → Disable	The POU is not built, not downloaded and not compared with the controller; the tree shows it dimmed and struck through, marked (disabled) . Recorded in <code>controller.json</code> (<code>pou.disabled</code>); the <code>.st</code> file is not touched, so version control shows no change. Enable reverses it.
Delete	Right-click → Delete , confirm	The file is deleted and the POU is removed from the task call list and the Build All exclusion list. No backup is taken.
Change kind	Right-click → Change to FUNCTION... / Change to FUNCTION_BLOCK...	Rewrites the header. Refused when a function block depends on per-instance memory (timers, counters, edge detectors).

5.12 Working with version control

A project directory is well suited to Git or any other version-control system:

Commit	Do not commit
<code>controller.json</code> , <code>globals.st</code> , <code>pou\</code> , <code>dut\</code> , <code>gvl\</code> , <code>lib\</code> , <code>hmi\</code> , <code>library.lock.json</code>	<code>backups\</code> (local history)
<code>data\</code> if saved values are part of your record	<code>uploads\</code> unless you keep them as evidence deliberately

`controller.json` contains the accounts' password **hashes** (salted PBKDF2), never passwords. The program-signing key and the controller tokens are not in the project; they belong to the IDE installation (Chapter 2) and must not be committed.

6. Controllers, connection and users

This chapter explains how the IDE reaches a controller: defining targets, connecting and disconnecting, the network paths (direct, SSH tunnel), logging in, what each role may do, managing controller accounts, the checks the IDE makes when you go online, and the **Controller Settings** page. Read it before connecting to a real controller for the first time, and whenever you set up accounts for a plant.

WARNING: The IDE can start, stop, force and reprogram a running machine from a desk, possibly in another room. Before you connect to a controller on a machine, make sure you know which machine it is, and that it is safe to change.

6.1 Targets

A **target** is a controller the IDE can talk to. The IDE holds a list of targets and has exactly one **active** target at a time. Every command that reaches a controller — download, write, force, run, trace — goes to the active target, and only while it is connected.

Targets belong to the IDE installation, not to the project. The same project can be downloaded to the simulator, to the controller on line 1 and to the one on line 2 by switching the target, and switching the project does not change the target. The list is stored in `.zeus-ide-targets.json` in the IDE package folder (Chapter 2).

Target type	Meaning	Board check
Simulator	The local simulator on this PC (<code>127.0.0.1:5020</code>). Always present as <code>Simulator (local)</code> .	None
IPC-CM400200	A Zeus PLC controller with a Raspberry Pi CM4	The controller must report a CM4 board
IPC-CM500200	A Zeus PLC controller with a Raspberry Pi CM5	The controller must report a CM5 board

When you connect to a target of type IPC-CM400200 or IPC-CM500200, the IDE compares the board the controller reports with the type you selected. If they differ, the **Messages** panel shows an error such as `Target "Line 1 filler" is set to IPC-CM500200 (expects CM5) but 192.168.1.50:5020 reports board "cm4..."`. This is not the controller you selected. On a plant floor, that mismatch is often the moment before a program goes to the wrong line — stop and check.

6.2 The Controller target dialog

Click **Target...** in the menu bar. The **Controller target** dialog lists every target:

Column	Content
(dot)	For the active target: green when connected
Name	The name you gave it
Type	Simulator, IPC-CM400200 or IPC-CM500200
Address	<code><SSH user>@<host>:<port></code>

Column	Content
Buttons	Connect (or Reconnect for the active target) / Disconnect (when connected); Deploy runtime... (controllers only); Edit ; Remove (not while connected)

Under **Add or edit**:

Field	Meaning	Default
Id	Short identifier: letters, digits, <code>.</code> , <code>-</code> , <code>_</code> . Saving with an existing id replaces that target.	—
Name	What the IDE calls this controller	the id
Type	Simulator, IPC-CM400200, IPC-CM500200	IPC-CM400200
Host	IP address or host name of the controller	—
SSH user	The Linux account on the controller used for SSH: hardware download, deployment, tunnel, clock, restart, reboot and shutdown	<code>pi</code>
Port	The monitor port	5020
Token	The monitor token, if the controller requires one. Shown as <code>(set)</code> once stored; leave <code>(set)</code> in the box to keep the stored token.	empty

Buttons: **Save target** stores the entry; **Clear (add new)** empties the fields for a new target.

To add a controller:

1. Click **Target...**
2. Click **Clear (add new)**.
3. Enter **Id** (for example `line1`), **Name** (`Line 1 filler`), **Type** (`IPC-CM400200`), **Host** (`192.168.1.50`), **SSH user** (the account on the controller) and **Port** (`5020`). Leave **Token** empty unless you were given one.
4. Click **Save target**. *Result:* the dialog reopens with the new target in the list.
5. Click **Connect** on its row. *Result:* the dialog closes, a toast says `Connecting to 192.168.1.50:5020...`, and the **Messages** panel notes `Connected to Line 1 filler (192.168.1.50:5020)`. The connection text in the header shows the controller's address next to a green dot.

A target that is connected cannot be removed; connect to another first. The last target cannot be removed.

6.2.1 Preparing SSH access

Every operation that changes a controller's files — **Download hardware**, **Deploy runtime...**, the SSH tunnel, **Set clock / time...**, **Restart runtime...**, **Reboot controller (OS)...** and **Shut down controller...** — runs over SSH with **key-based authentication only**. The IDE never waits for a password prompt: a missing key fails immediately with a message rather than hanging.

1. On the engineering PC, create a key once if you have none: open a Command Prompt and run `ssh-keygen` (accept the defaults).
2. Install it on the controller: run `type %USERPROFILE%.ssh\id_ed25519.pub | ssh <user>@<controller> "cat >> ~/.ssh/authorized_keys"` and enter the controller account's password once. (On systems with `ssh-copy-id`, use that.)
3. Check it: `ssh <user>@<controller> hostname` must print the controller's host name without asking for a password.

The first connection to a controller records its host key (`accept-new`); a controller whose key later changes is refused, as SSH should. If a controller is re-imaged, remove its old entry from `%USERPROFILE%\ssh\known_hosts`.

6.3 Connecting and disconnecting

Connect (or **Reconnect**) on a target's row makes it the active target and connects the IDE server to it. The previous connection is closed first, so the IDE is never connected to two controllers at once.

Once connected:

- the connection text shows the controller's `host:port` and the dot turns green;
- the target badge shows the board and the IO kind (Chapter 4);
- if the controller has accounts, the **Log in to the controller** dialog opens (6.5);
- the IDE checks the monitor protocol version and reports a mismatch in the **Messages** panel (`The controller is newer than the IDE - update the IDE.` or the reverse);
- the sync badge starts reporting whether the controller runs this project (6.9).

If the connection drops — a cable pulled, a controller restarting — the IDE reconnects by itself and shows `disconnected - <reason>` meanwhile. Everything shown as live is blanked while disconnected rather than left showing stale values.

To go offline deliberately: **Target...** → **Disconnect** on the connected row, and confirm. The IDE stops polling and does **not** reconnect by itself until you click **Connect** again.

Note: Disconnecting does not affect the controller. Its scan keeps running, its outputs keep being driven and any forces stay in place. Being disconnected is also a safe position: no download or force from this IDE can reach any controller.

Important: **Download sends to whichever target is connected.** Check the target badge and the connection text before every download.

The **Simulator** page's **Start / Restart** always selects and connects the simulator target, so starting the simulator also switches the IDE away from a real controller.

6.4 Network paths and securing the connection

The monitor protocol is line-delimited JSON over TCP. On its own it is **not encrypted**: a token, login passwords and every value read or written cross the network in clear. Choose the path accordingly.

Path	How	When
SSH tunnel (recommended)	The controller's monitor listens on loopback only; the IDE opens an SSH tunnel and connects through it	Any controller on a plant network
TLS	The controller's monitor speaks TLS (<code>monitor.tls</code> on the controller); the target is marked for TLS	Where TLS certificates are provisioned on the controllers
Direct	The monitor listens on the controller's network address	Isolated bench networks only

The runtime protects itself against the worst case: it refuses to start if the monitor is bound to a network address without a token of at least 16 characters (`monitor.bind` is `"..."` (not loopback) but no token is set).

6.4.1 The SSH tunnel

A target marked for tunnelling is reached like this: the IDE starts `ssh` with a local forward from a free port on `127.0.0.1` of the engineering PC to `127.0.0.1:5020` on the controller, watches it, and restarts it if it drops. The header still shows the controller's own address. The tunnel uses the same key-based SSH as the other operations, so it needs no extra credentials; anyone who can open it already has SSH access to the controller. Authentication to the runtime (token or login) and the role checks still apply inside the tunnel.

The tunnel is enabled per target with `"tunnel": true` in `.zeus-ide-targets.json`. The **Controller target** dialog has no field for it in this release:

1. Stop the IDE (`"Zeus PLC IDE.cmd" stop`).
2. Open `.zeus-ide-targets.json` in the IDE package folder with a text editor and add `"tunnel": true` to the target's entry:

```
{
  "active": "line1",
  "list": [
    { "id": "simulator", "name": "Simulator (local)", "kind": "simulator",
      "host": "127.0.0.1", "port": 5020, "token": null },
    { "id": "line1", "name": "Line 1 filler", "kind": "ipc-cm400200",
      "host": "192.168.1.50", "user": "zeus", "port": 5020, "token": null,
      "tunnel": true }
  ]
}
```

3. Start the IDE and connect. *Result:* the **Messages** panel notes `tunnel up: 127.0.0.1:<port> -> 192.168.1.50:5020 over SSH`.

A **Download hardware** to a tunnelled target writes a controller configuration whose monitor is bound to loopback, so after it the controller can only be reached through the tunnel.

If the tunnel cannot be opened, the IDE reports `Could not open the SSH tunnel to <host>: ... Falling back to a direct connection - the monitor protocol is NOT encrypted...` and tries a direct connection. Treat that message as an incident on a plant network.

Important: In this release, clicking **Save target** in the dialog for a target rewrites its entry without the `tunnel` and `tls` fields. After editing a tunnelled target in the dialog, add `"tunnel": true` again as above.

6.4.2 Tokens

A **token** is a shared secret for a program rather than a person. When a controller requires one, the IDE sends it with every request; it is entered in the target's **Token** field. When you use **Download hardware** to a directly-connected controller that has no usable accounts, the IDE generates a token, writes it into the controller configuration and stores it in the target itself. When the project defines accounts including an engineer, the accounts are the gate and no token is written, so that every person has to log in under their own name.

6.5 Logging in

6.5.1 A login belongs to a connection

The controller authenticates **connections**, not browsers or PCs. A login is valid for the connection it was made on, and ends when that connection ends:

- after the controller or the simulator restarts, or after a hardware download (which restarts the runtime), the IDE tries to log in again with the same credentials; if that fails — for example because the download changed the password — it asks you to log in again;
- after **Disconnect** and **Connect**, you log in again;
- **Online ▾** → **Log out** drops the connection and opens a new, anonymous one. The controller keeps running; you simply stop being authenticated to it.

The password is sent to the controller once and not kept by the IDE; what the IDE keeps is the name and role the controller confirmed.

6.5.2 When the IDE asks you to log in

The controller announces on connection whether it requires a login (it does whenever it has at least one account or a token). The IDE then opens **Log in to the controller** once per connection. If you close it, the IDE does not ask again on every poll: the run-state badge shows `NO SESSION` and the sync badge `⚠ NOT CHECKED – LOG IN`, and nothing is read from the controller. Click either badge, or use **Online ▾** → **Log in...**, when you want live values.

A command that needs a login — a download, a write, a force — also opens the dialog, with the reason, and continues when you have logged in.

To log in:

1. **Online ▾** → **Log in...** (or the prompt, or click `NO SESSION`).
2. Enter **User** and **Password**.
3. Click **Log in**, or press **Enter**. *Result:* the session badge shows `<user> · <role>` and a toast says `Logged in as <user> (<role>)`.

A wrong user name and a wrong password give the same answer, `the user name or password is not correct`, and take the same time, so the answer does not reveal which accounts exist.

A controller that has **no accounts** cannot be logged in to; it answers `this controller has no accounts configured, so there is nothing to log in to...`. Such a controller (and no token) accepts every connection as an anonymous engineer. That is appropriate for a bench, not for a plant.

6.5.3 Failed logins and logout

Rule	Value
Failed attempts allowed per source address	5
Lockout of that address after the 5th failure	60 seconds
Connection	Closed by the controller after 5 failures on it
Counting	Failed logins and failed token attempts count together, so alternating does not give more attempts

Rule	Value
Record	Every failure and every lockout is written to the controller's audit log with the address

While an address is locked out, every attempt from it is refused, including correct ones. Wait 60 seconds.

6.6 Roles: what each may do

Every account has one of three roles. The controller checks the role for **every** command, whatever client sends it; a command that is not listed as permitted for a role is refused (default deny).

Action	viewer	operator	engineer
Connect, read status, watch live values, read the variable table and the IO map	✓	✓	✓
Read the controller log, see which points are forced	✓	✓	✓
Verify, Compare, Upload	✓	✓	✓
Read alarms and the alarm journal, read signatures, read a trace	✓	✓	✓
Sign... (with re-entry of the password)	✓	✓	✓
Write values, force and release points, clear forces		✓	✓
Run, Pause, Reset fault , reset statistics		✓	✓
Acknowledge, shelve and unshelve alarms		✓	✓
Download program			✓
Scan fieldbuses (Modbus, EtherNet/IP, EtherCAT, Profinet), set Profinet names and addresses			✓
Config mode (enter and leave)			✓
Breakpoints and single-step			✓
Arm, start, stop and clear a trace			✓
Take an alarm out of service			✓
Download hardware, Download screens, Deploy runtime..., Restart runtime..., Reboot controller (OS)... , Shut down controller... , Set clock / time... , change a MAC address (checked by the IDE; see note)			✓

Seeing what is forced is deliberately a **viewer** right: the person who most needs to know is often the one with the least authority to change it.

Note: The operations in the last row run over SSH, not over the monitor protocol. The IDE refuses them unless the connected controller's session is an **engineer** — and, for a download, unless no account in the project still has its initial password (6.8). On a controller that has no accounts yet, they are allowed, because the first hardware download is what installs the accounts; the SSH key is then the credential.

Every action that changes the machine is written to the controller's audit log with the user name and the address it came from. Refusals are logged too, with the role that would have been needed.

6.7 Users & Roles

Controller accounts are part of the project: they are stored in `controller.json` and therefore reviewable, versioned and archived with it. They reach the controller with **Download hardware**. The same accounts are used by the IDE and by the operator screens served by the controller.

Open the **Users & Roles** dialog from the **Users & Roles** node in the tree, **Online ▾ → Users & Roles...**, or right-click **Application → Users & Roles...**

Part	Content
Table	User, Role (with an initial password badge if the account still has its creation password), May (a summary of the role), Remove
Add or replace	User, Password, Confirm, Role, Save user

To add a user, or change a password or role:

1. Enter **User**: 2 to 31 characters, letters, digits, `.`, `-` or `_`.
2. Enter **Password** and **Confirm**: at least 8 characters. `admin`, `password`, `12345678`, `zeus` and the user name itself are refused.
3. Choose **Role**: `viewer – watch only`, `operator – write, force, run/pause`, or `engineer – download and everything above`.
4. Click **Save user**. *Result*: the table shows the user; a toast says `<user> saved – download hardware to apply`.
5. **Online ▾ → Download & Upload → Download hardware** (logged in as an engineer) to put the change into effect on the controller.

Saving an existing user name **replaces** that account — this is how a password or a role is changed. Passwords are stored as a salted PBKDF2-SHA256 hash (200,000 iterations); an existing password cannot be displayed, only replaced.

To remove a user: click **Remove** on the row and confirm. The IDE refuses to remove the last engineer of a project that has other accounts, because only an engineer can download or change accounts, and the controller would be locked.

Limit	Value
Accounts per controller	16

Important: Accounts take effect on the controller **only after Download hardware**. Until then the controller keeps the accounts it had. A controller you believe is protected and is not is worse than one you know is open — download, then log in with the new account to confirm.

A project with no accounts shows a note that the controller accepts any connection, and a button **Create the default admin account**. It creates `admin / admin` with the engineer role and the must-change mark (6.8), so commissioning has a way in. It is refused once any account exists.

6.8 Initial passwords that must be changed

Every new project contains the account `admin`, role engineer, password `admin`, marked **must change**. That password is printed in this manual, so it protects nothing. The system makes sure it does not survive commissioning:

Where	What happens
Users & Roles	The account shows the initial password badge
Logging in with it	The controller accepts the login, reports <code>must_change</code> , and writes <code>STILL USING ITS INITIAL PASSWORD</code> to its audit log. The IDE immediately opens Change the password for "admin" .
Any download	The IDE refuses to send a configuration that still contains an initial password, and opens the change-password dialog; the download continues once it is changed
Sign...	An account on its initial password cannot sign: a guessable credential is not evidence of who signed

To change an initial password when asked:

1. In **Change the password for "<user>"**, enter the new password in **New password** and **Confirm** (same rules as in 6.7).
2. Click **Set password**. *Result:* the password is changed in the project (the role is kept), and the **Messages** panel warns `Password for "<user>" changed in the project. It does not take effect on the controller until you Download hardware - until then the old one still works.`
3. Download hardware. After the runtime restarts, log in with the new password.

On a controller that has no accounts yet, the very first hardware download may carry the initial password — otherwise the account could never arrive. Log in with it once, change it when asked, and download hardware again. It is simpler to set the password in **Users & Roles** before the first download, as in Chapter 3.

6.9 Checks when you go online

The IDE continuously checks whether the controller runs what is on your screen, because live values that come from a different program than the one you are reading are the most misleading thing an IDE can show.

The sync badge (Chapter 4) is updated shortly after every edit — including unsaved edits — after every download and every 30 seconds. It compares the SHA-256 of the application the IDE would send now (the whole call list, linked) with the one the controller reports, and the hash of the hardware configuration the IDE would send with the one the controller loaded. If the project has accounts and you are not logged in, it reports `⚠ NOT CHECKED – LOG IN`: the controller refuses to answer, and an unanswered question is not a match.

Going online (**Online** ▾ → **Monitor**) runs a full comparison. If everything matches, a toast says `Program and hardware match the controller`. If anything differs, the **IDE compared with the controller** dialog opens with the warning `This project is NOT what the controller is running. Live values below come from the program and rack listed under "On the controller"...`, and a warning is added to the **Messages** panel.

The comparison dialog (also **Online** → **Download & Upload** → **Compare...**, ⇌ in the tool strip, or a click on the sync badge) lists three groups:

Group	Items
Different	Items that do not match, with In this project , On the controller and What that means

Group	Items
Identical	Items that match — shown so that "nothing is wrong" can be told apart from "nothing was checked"
Could not be checked	Items the controller could not answer, with the reason. Treat the controller as unverified until they can be checked.

Item compared	How
program	The controller hashes the source it is running and compares it with the linked application from this project (or the call list, when the task calls several programs)
device <name> (one per IO device)	Present on the controller or not; and the %I / %Q byte sizes, because every address after a size change moves. Devices running on the controller but absent from the project are listed too.
scan period	The task interval in the project against the controller's

Both sides are compared against what the **controller reports**, not against a record of what was last sent: a failed download, a hand-edited configuration or a second engineer's laptop all show up.

The device tree adds a hardware check of its own: every few seconds it compares the hash of the hardware configuration the IDE would send with the one the controller loaded. When they differ, it stops painting devices green — a card that exists only in the project must not look live — marks them `not downloaded`, and warns once in the **Messages** panel: `The controller is running a different hardware configuration from this project... use Download hardware to send this rack.`

Debug ▾ → Verify asks the single question "does the controller run exactly the program in the active POU tab?" and answers with a toast (`Match – the controller is running this source` or `MISMATCH – the controller is running something else`) and in the **Messages** panel. Open the POU first; with several programs in the call list, use the sync badge or **Compare...**, which check the whole application.

6.10 Controller Settings

The **Controller Settings** node opens a page with the controller-wide settings of the project. Changes are stored in `controller.json` when you click **Apply**, and reach the controller with **Download hardware**.

Field	Meaning	Default
Controller name	The name shown in the menu bar, reported by the controller and used in archive names	<code>zeus-controller</code>
%M marker bytes	Size of the %M area	4096
Retain file	Where the controller keeps retained data. A hardware download always sets the controller's standard path (<code>/var/lib/zeus-plc/retain.bin</code>), whatever this field says.	<code>/var/lib/zeus-plc/retain.bin</code>
Log level	<code>trace</code> , <code>debug</code> , <code>info</code> , <code>warn</code> , <code>error</code>	<code>info</code>
Monitor bind	The address the monitor listens on. Loopback (<code>127.0.0.1</code>) is the safe choice with an SSH tunnel. A hardware download sets it for the target (the controller's address for a direct target, loopback for a tunnelled one).	<code>127.0.0.1</code>

Field	Meaning	Default
Monitor port	The monitor port	5020
Allow write / force	When cleared, the controller refuses downloads, writes and forces over the monitor protocol, whatever the role — a read-only controller	ticked

The page also holds these sections, described in their own chapters:

Section	Content	Chapter
Redundancy (2-controller)	Enable, role (<code>auto</code> , <code>primary</code> , <code>secondary</code>), sync port (9910), peer and local addresses for the sync link and an optional backup path, heartbeat, peer timeout, checkpoint and takeover-guard times. In development — see Chapter 1.	33
HMI tag server (EtherNet/IP)	Enable EtherNet/IP tags , Bind address , Port (44818), Allow writes ; and Create EDS file... for the controller model	30
Retained data	How <code>%M</code> and <code>RETAIN</code> variables survive a power cycle, and why a changed retained layout resets them	33

Button	What it does
Apply	Saves the page into <code>controller.json</code> . Checks the redundancy fields (a peer address when enabled; peer timeout greater than heartbeat).
Validate with the runtime	Asks the runtime to load the saved configuration and report, without starting anything; shows <code>✓ configuration is valid</code> or <code>X configuration REJECTED</code> and the runtime's own explanation. Needs a runtime build on the engineering PC: the self-contained Windows package has only the compiler and says so (<code>This install has the ST compiler but not the runtime...</code>).

Important: The monitor settings decide whether you can reach the controller after the next hardware download. Do not clear **Allow write / force** or change the port on a remote controller unless you have another way in (SSH, or a local panel).

6.11 Installing or updating the runtime (Deploy runtime...)

Controllers are delivered with the runtime installed. **Target...** → **Deploy runtime...** installs or replaces the runtime on a controller over SSH; it is not the same as a download.

	Download program / hardware	Deploy runtime...
Replaces	The program, or the configuration	The runtime executable and its service
Effect on the machine	Program: none (online change). Hardware: the runtime restarts.	Stops the control loop ; the runtime is reinstalled and restarted
Role	Engineer	Engineer

The **Deploy runtime — <name>** dialog shows the controller, **SSH user**, **Install service** and **Start after install**, and lists each step with **OK** or **!!**. Deploy can be repeated: it replaces what is there rather than assuming a clean controller. It needs a runtime build for the controller (ARM64) on the engineering PC, which the self-contained

Windows package does not contain; runtime updates are supplied by Zeus Automation with their own instructions.

WARNING: Deploy runtime stops the control loop and restarts the controller's runtime. Never run it on a machine that is producing.

6.12 Checklist: first connection to a new controller

1. The controller is powered, on the network, and you can `ping` it.
2. SSH key access works (6.2.1).
3. The project has at least one engineer account with a password you have set (6.7).
4. **Target...** → add the controller with the correct **Type**, **Host** and **SSH user**; for a plant network, enable the tunnel (6.4.1).
5. **Connect**. Check the target badge (board and IO kind) and that no board-mismatch error appears.
6. **Download hardware** (Chapter 20) — this installs the IO configuration and the accounts, and restarts the runtime.
7. Log in with your engineer account. Confirm the session badge.
8. **Download program**, then go online and confirm `IN SYNC`.

ZEUS PLC IDE · OPERATION MANUAL

Part III — Hardware

7. IO devices and scanning

This chapter explains how the controller reaches its field IO: the **IO Devices** branch of the device tree, adding and removing devices, the scan tools under **Scan for RIO...**, the device page, and how the devices together build the process image. It also covers the **Network Interfaces** branch and the settings that decide what outputs and inputs do when something stops or fails. Read it before you configure the first rack of a new machine. The chapters after it cover each fieldbus in detail: Modbus TCP (Chapter 8), EtherCAT (Chapter 9), Profinet (Chapter 10) and IO-Link (Chapter 12).

7.1 Concepts

An **IO device** is one entry in the controller's configuration that owns one fieldbus connection. It is one of these:

- one Modbus TCP server (a head station, a drive, a meter)
- one EtherCAT master and the segment behind it
- one Profinet IO device (a coupler and its rack)
- one EtherNet/IP adapter
- one simulated device (the `sim` driver)

Each IO device runs one **driver**. The driver exchanges data with the field and copies it into and out of the controller's **process image**: `%I` for inputs and `%Q` for outputs. The program never talks to a fieldbus. It reads `%I` at the start of the scan and writes `%Q`, and the drivers do the rest.

Term	Meaning
Device	One entry under IO Devices ; one driver instance with a unique name
Driver	The code that speaks the protocol: <code>modbus_tcp</code> , <code>ethercat</code> , <code>profinet</code> , <code>ethernet_ip</code> , <code>sim</code>
Block	A contiguous piece of one device's data in <code>%I</code> or <code>%Q</code> . A Modbus block is one request; an EtherCAT or Profinet block is one card's inputs or outputs
Card / module	One physical IO card in a rack. On EtherCAT and Profinet each card becomes one or two blocks
Rack	The coupler (head station) and the cards fitted to it, in slot order
Slice	The part of <code>%I</code> and <code>%Q</code> that belongs to one device

The configuration of all devices is stored in the project's `controller.json` file, under the `io` array (see Appendix H). Everything the IDE shows on a device page is read from and written to that file.

7.2 The IO Devices branch

The **IO Devices** node sits under the controller node in the **Devices** pane. Its meta text reads *N configured*. Under it are:

- one node per configured device, in configuration order
- under each device, one node per card (or per Modbus block)
- the **Scan for RIO...** branch, which holds the scan tools (Section 7.5)

The controller node itself shows the size of the process image that the configuration produces, for example `%I 36 B %Q 24 B`.

7.2.1 What a device node shows

Element	Meaning
Label	The device's instance name
Badge	The live state while connected (<code>OPERATIONAL</code> , <code>DEGRADED</code> , <code>STARTING</code> , <code>FAULT</code> , <code>STOPPED</code>), the driver name when offline, <code>disabled</code> , <code>inhibited</code> , or <code>not downloaded</code>
Meta	The exchange mode (<code>async</code> or <code>sync</code>), the network port, <i>N card(s)</i> under a port view, and <i>up 2h 14m</i> when the controller reports how long the link has been up
Blinking	The device is configured but not exchanging data (any state other than <code>OPERATIONAL</code> or <code>DEGRADED</code>)

The port in the meta text is either observed or configured. A Modbus or EtherNet/IP device reports the port its socket actually used, for example `async · eth0`. An EtherCAT or Profinet device binds a named interface, so the port is shown with `(configured)` until the controller reports it.

Note: A device shows `not downloaded` (amber) when the controller is running a different hardware configuration from the one in the project. The cards shown in the tree are then not necessarily the ones exchanging data. Use **Download hardware** (Section 7.7) to send the project's configuration.

An inhibited or disabled device never blinks. It is doing what it was told.

7.2.2 What a card node shows

Each card node shows its slot, its part number, what kind of card it is, and where its channels are. For example:

```
2 CT-3238 8×AI · 0-20mA/4-20mA %IW2 ×8
4 CT-4234 4×AO · 0-20mA/4-20mA %IB34 ×1 · %QW0 ×4
```

The kind (DI, DO, AI, AO) and the ranges come from the vendor's description file (ESI or GSDML). They are never guessed from the data width. A card node takes the colour of its device, because nothing on the wire reports per-card health.

Opening any card node opens the device page of its device.

7.3 Adding a device

There are two ways to add a device:

- **Add Device** dialog: for Modbus TCP head stations and generic Modbus TCP devices, and for the simulated device. You lay the rack out by hand.
- **Scan for RIO...**: the controller searches the network and you add what answers. This is the only way to add a Profinet device, and the recommended way for EtherCAT.

For EtherCAT without hardware, use **Add EtherCAT rack from ESI...** (Chapter 9).

7.3.1 The Add Device dialog

To add a device with the Add Device dialog:

1. In the **Devices** pane, click **+** (tooltip *Add device*). Alternatively, right-click **IO Devices**, or click **Add a device** on the welcome page.
2. Under **1 • Device type**, click the device type. The types are listed in Table 7-1.
3. Under **2 • Connection**, enter an **Instance name**. Leave it empty to accept the default (`cn8031_1` , `generic_modbus_2` , and so on). For a Modbus device, enter the **Host / IP**, the **Port** (default 502) and the **Unit id** (default 1).
4. Select the **Exchange mode: async — driver thread (recommended)** or **sync — inline in the scan**. See Section 7.4.3.
5. For a head station, under **3 • Modules (slot order)**, select each card and click **Add to rack**, in the order the cards sit on the rail. The dialog shows the addresses each card will get, and the total process image, as you build the rack. Remove a card with **X**.
6. Click **Add device**.

The new device's page opens. Any warning from the address assignment (for example, a block larger than one Modbus request) is written to the **Messages** panel.

Table 7-1 — Device types in the Add Device dialog

Type	Driver	Use
CN-8031	<code>modbus_tcp</code>	ODOT C-series head station over Modbus TCP. The dialog assigns register addresses from the card catalogue
CN-8033	<code>ethercat</code>	Marked <i>phase 2+</i> . Adds a disabled placeholder only. Use Scan EtherCAT segment... or Add EtherCAT rack from ESI... instead
CN-8032 / CN-8032-L	<code>profinet</code>	Marked <i>phase 2+</i> . Adds a disabled placeholder only. Use Scan Profinet network... instead
Modbus TCP device	<code>modbus_tcp</code>	Any Modbus TCP server: drive, meter, third-party IO. The blocks come from the device manual
Simulated IO	<code>sim</code>	Signal generators and <code>%Q</code> → <code>%I</code> loopback. No hardware needed

Important: The CN-8033 and CN-8032 entries in the Add Device dialog are left over from before the EtherCAT and Profinet drivers existed. They add a disabled device with no usable configuration. Always add EtherCAT and Profinet devices through **Scan for RIO...** (or, for EtherCAT only, **Add EtherCAT rack from ESI...**).

7.3.2 The card catalogue (CN-8031)

For a CN-8031 head station the dialog offers these ODOT C-series cards:

Card	Kind	Channels	Modbus object	Process image
CT-1108 / CT-1116 / CT-1132	DI	8 / 16 / 32	discrete inputs	1 / 2 / 4 bytes of <code>%I</code>
CT-2108 / CT-2116 / CT-2132	DO	8 / 16 / 32	coils	1 / 2 / 4 bytes of <code>%Q</code>
CT-1616	DI + DO	8 + 8	discrete inputs + coils	1 byte <code>%I</code> , 1 byte <code>%Q</code>

Card	Kind	Channels	Modbus object	Process image
CT-3104 / CT-3108	AI 0-20/4-20 mA	4 / 8	input registers	8 / 16 bytes of %I
CT-3204	AI 0-10 V / ± 10 V	4	input registers	8 bytes of %I
CT-4102 / CT-4104	AO 0-20/4-20 mA	2 / 4	holding registers	4 / 8 bytes of %Q
CT-4204	AO 0-10 V / ± 10 V	4	holding registers	8 bytes of %Q
CT-3304	RTD PT100/PT1000	4	input registers	8 bytes of %I
CT-3404	Thermocouple	4	input registers	8 bytes of %I
CT-5102	Counter, 32-bit	2	input registers (2 per channel)	8 bytes of %I

The dialog tracks two address spaces independently: the Modbus address inside each object type, and the byte offset in %I and %Q. A DI card and an AI card can both start at Modbus address 0, because they are in different object types, while they sit at different %I offsets.

WARNING: The register addresses the catalogue assigns are the ODOT defaults for a rack populated left to right with no gaps. Before commissioning, compare every block with the head station's own web page or the vendor's internal-bus mapping table. A wrong map does not fail: it produces a controller that runs and reads or drives the wrong terminals.

7.3.3 Removing a device

To remove a device:

1. Open the device page and click **Remove device**, or right-click the device node.
2. Confirm *Remove device "... and all its modules?*

The device is deleted from `controller.json`. Its aliases are deleted with it.

Important: Removing a device shifts the process-image slice of every device after it (Section 7.8). If you only want the device to stop exchanging data while a panel is unfinished, **inhibit** it instead (Section 7.4.4).

7.3.4 The 16-device limit

The controller accepts at most **16 entries** in the `io` array. The limit counts every entry, including disabled and inhibited ones. A configuration with a seventeenth entry is refused at startup with *too many io entries (max 16)*. The IDE does not stop you from adding a seventeenth device, so the error appears when the configuration reaches the controller.

A single device can carry far more than one card: a Modbus device takes up to 64 blocks, and one EtherCAT or Profinet device carries a whole rack. Sixteen devices is rarely the constraint. Other limits that apply: one EtherCAT master per controller (Chapter 9), and one Profinet device per controller in this release (Chapter 10).

7.4 The device page

Double-click a device node (or any of its card nodes) to open its page. The page title is the device name, and the line under it reads *driver · mode mode*, with *disabled* added when the device is disabled.

The page has these sections, from top to bottom:

1. **Live state** (only while connected): the state, the number of blocks or slaves up, error and timeout counters, the driver's detail line, and the connection time (Section 7.9).
2. **Connection**: the driver's parameters and the device flags. The fields depend on the driver.
3. **Apply** and **Remove device**.
4. **Channel aliases**: one row per channel, with its address and an alias box (Chapter 11).
5. **Modules — process image mapping**: the card table, the **Signal range** column, and **Edit rack...** for EtherCAT and Profinet.

Changes in the **Connection** section are written to `controller.json` only when you click **Apply**. Aliases and signal ranges are saved as soon as you change them. Nothing reaches the controller until **Download hardware**.

7.4.1 Connection fields by driver

Driver	Fields
Modbus TCP	Host, Port, Unit id, Poll cycle (ms), Response timeout (ms), Exchange mode, On stop — outputs, On fault — outputs, Enabled, Inhibited, Critical — stop the controller if it fails, Wait for connection at start (s)
EtherCAT	Interface, Cycle (μs), Distributed clocks, Expected slaves, Require bus at startup, %I bytes (bus absent), %Q bytes (bus absent), Exchange mode (async only), On stop — outputs, On fault — outputs, Enabled, Inhibited, Critical — stop the controller if it fails, Wait for connection at start (s)
Profinet	Interface, Station name, IP address, Vendor / Device (read only), Send clock (×31.25μs), Reduction ratio, Watchdog (cycles), Exchange mode (async only), On stop — outputs, On fault — outputs, Missing card, Enabled, Inhibited, Critical — stop the controller if it fails, Wait for connection at start (s)
EtherNet/IP	Load EDS file... plus the connection fields

The driver-specific fields are described in Chapters 8 to 10. The device flags are common to all drivers and are described next.

7.4.2 Enabled

Clearing **Enabled** removes the device from the running configuration. The controller does not create the driver at all, reserves no process image for it, and logs *io[n] '...' is disabled - skipping*.

Important: A disabled device has **no slice** in the process image, so every device after it moves down (Section 7.8). A program with **AT** addresses on a later device then refuses to load, or worse, reads a different device's data. Use **Inhibited** when the device must stay in place.

7.4.3 Exchange mode: async and sync

Mode	How the data moves	Use it for
async	The driver has its own thread. It exchanges data at its own cycle and the scan copies a shadow buffer. The scan never waits on the network	Every fieldbus on a fast scan. Required for EtherCAT and Profinet
sync	The driver exchanges data inside the scan: read at the start, write at the end	A Modbus device on a slow scan (for example 50 ms or more) on a dedicated switch, where one round trip is a small fraction of the period

In sync mode a Modbus device charges its whole round trip to the scan, and in the worst case *blocks × response timeout*. The runtime prints that worst case at startup. On a 10 ms scan, use async.

The async driver thread runs one real-time priority below the scan unless the configuration says otherwise, so the control loop always preempts its own IO.

7.4.4 Inhibited

Inhibited keeps the device configured and addressed, but never starts it:

- no sockets and no bus traffic
- no expectation that it answers, and no **IO** lamp blink for it
- its **%I** slice holds the fail-safe pattern (zeros)
- its slice stays exactly where it is, so no address moves

The device tree shows it with an **inhibited** badge. The controller logs an audit record: *io '...' is INHIBITED: addressed but not started*.

Use it for a panel whose wiring is not finished, or for a skid that is out for repair.

7.4.5 Critical

Critical — stop the controller if it fails turns the loss of this device into a controller **FAULT**. The scan stops executing the program and stays in FAULT until someone resets it (Chapter 21).

The default is off. On most plant a rack with one dead card should keep running the other fifteen, and a controller that stops everything is the more dangerous answer. **DEGRADED** counts as exchanging data. Turn **Critical** on only where running without the device is itself the hazard, for example a dosing skid that has lost its flow meter.

A critical device gets a start-up wait before its absence counts. When the controller starts, the program runs and the controller waits for the device to connect. If the device has not connected when the wait runs out, the controller goes to **FAULT**. The wait covers a Profinet identify and connect, an EtherCAT walk to OPERATIONAL, and a switch learning its ports.

Wait for connection at start (s) sets the wait for this device, 0 to 600 seconds. The field is enabled only while **Critical** is ticked. Leave it empty to use the controller default, `task.io_startup_grace_ms` in `controller.json` (30 s unless changed; the placeholder shows the value in force). Enter `0` for no wait: the device must already be exchanging on the first scan. The IDE saves it as `"startup_timeout_ms"` on the device (Appendix H).

Give a remote rack behind a slow switch a longer wait than the coupler next to the controller, rather than raising the default for every device. The controller log says *io '...': critical - waiting up to N s for it to connect* at start, and *io '...': connected after X s* when the device arrives. Use the second figure to choose the wait.

The rules are:

- Inside the wait nothing about the device faults the controller. A coupler that connects, drops once or twice while it releases an old connection, and then stays up has started normally (the CN-8032 does this after a runtime restart).
- A critical device that is not exchanging when the wait ends faults the controller: `critical-io-never-came-up` if it was never seen (look at cabling, address and station name), `critical-io-lost` if it had connected and dropped again.
- After the wait, a critical device that stops exchanging faults the controller at once, with the event `critical-io-lost`. Look at what changed.
- An inhibited device is never critical.

WARNING: A controller fault stops the program, not the machine. What the outputs do next is decided by **On fault — outputs** (Section 7.10) and by each device's own watchdog. Check both before you rely on **Critical** as a protective measure. Critical is not a safety function.

7.4.6 Missing card (Profinet)

A Profinet coupler that answers with fewer cards than the configuration describes is still exchanging data for the cards that are present. That condition is handled by **Missing card**, not by **Critical**:

Setting	Effect
degrade — report it, keep running (default)	The device shows <code>DEGRADED</code> , the IO lamp blinks, the program keeps running on the cards that are present
fault — report a fault (stops only if Critical)	The device reports <code>FAULT</code> . The controller stops only if Critical is also ticked
ignore — carry on as if complete	The mismatch is logged once when the coupler connects, and nowhere else

The page explains the effect of the chosen combination under the form. A `module_policy` of `fault` without `critical` also produces a warning when the configuration loads, because it reports a fault and keeps the machine running.

7.5 Scan for RIO...

The **Scan for RIO...** branch under **IO Devices** groups the tools that find hardware on the network:

Node	Meta	What it does
Scan for remote IO...	Modbus TCP	Probes an IP range for Modbus TCP servers
Scan for EtherNet/IP...	EtherNet/IP	Sends ListIdentity to an IP range, optionally broadcast
Scan EtherCAT segment...	EtherCAT	Enumerates the slaves on one interface, stopping at <code>SAFE_OP</code>
Scan Profinet network...	Profinet	DCP Identify-All on one interface; also names and addresses devices
Add EtherCAT rack from ESI...	offline	Lays out an EtherCAT rack from a vendor file with no bus present

Every scan runs **on the controller**, not on the engineering PC. The fieldbus subnet is normally reachable only from the controller, and a scan from the laptop would find nothing. The IDE must therefore be connected to a controller, and you must be logged in with the **engineer** role. A Modbus scan is also refused when the controller's monitor is set to read-only (`monitor.allow_write: false`), because a scan is an active probe of the network.

WARNING: A scan puts traffic on a live fieldbus. The EtherCAT scan takes the bus to `PRE_OP` and `SAFE_OP` and back to `INIT`, and is refused while the controller's own master is cycling. Do not scan a segment that is running a machine unless you have confirmed that the other devices on it tolerate it.

7.5.1 Scan for remote IO (Modbus TCP)

To scan for Modbus TCP devices:

1. Open **Scan for RIO...** → **Scan for remote IO...**

2. Check **From** and **To**. They default to `.1` and `.254` of the controller's first network port. The dialog lists the controller's ports so you can pick the subnet the IO is on.
3. Set **Port** (502), **Unit id** (1) and **Timeout (ms)** (300).
4. Click **Scan**.
5. In the result table, tick the devices to add and click **Add selected**.

The scan works in two phases. First every address gets a non-blocking TCP connect, all in one wait, so a /24 costs one timeout rather than 254. Then each address that answered is asked for its Device Identification (function 43/14, optional in the standard) and, for each of the four object spaces, the largest block it will read starting at address 0. At most 1024 addresses are scanned in one run.

The result table shows the address, the identification string, and the counts of **Coils**, **DI**, **HR** and **IR**.

Each selected device is added as `mb_<ip>_<unit>` (for example `mb_192_168_1_10_1`), enabled, async, 10 ms poll, 50 ms response timeout, with **one block per object space** named `DI`, `AI`, `DO` and `AO`. Re-scanning is idempotent: a device already in the project is skipped.

Important: The counts say what the device will read, not which cards are fitted. Modbus has no module inventory: two 8-channel cards and one 16-channel card are the same 16 coils on the wire. Split the blocks into cards yourself before commissioning, and verify the register map (Chapter 8).

7.5.2 Scan for EtherNet/IP

To scan for EtherNet/IP adapters:

1. Open **Scan for RIO...** → **Scan for EtherNet/IP...**
2. Set **From**, **To** and **Timeout (ms)** (500). Tick **Also broadcast** only when you are looking for a device whose address nobody knows.
3. Click **Scan**, then click a row to add that device.

The result shows address, product, vendor, device type, revision and serial number. A clicked device is added **disabled**, named `enip_<ip>`, with its assembly instances set to 0. EtherNet/IP does not report its assembly layout on the wire, so the instances and sizes must come from the device's EDS file: open the device page and click **Load EDS file...**, pick the connection, then enable the device.

Note: The EtherNet/IP IO driver (Class 1 scanner) has not yet been run against hardware. Treat it as unvalidated. This is separate from the EtherNet/IP tag server of Chapter 30, which serves the controller's own tags to HMIs.

7.5.3 Scan EtherCAT segment, Add EtherCAT rack from ESI

These are described in Chapter 9.

7.5.4 Scan Profinet network

This is described in Chapter 10. It also gives devices their station name and IP address, which a Profinet device must have before any controller can connect to it.

7.6 The Network Interfaces branch

The **Network Interfaces** node lists the controller's own Ethernet ports, as the controller reports them. It is empty (*not reported — controller not connected*) while the IDE is offline. Its meta text is the number of ports.

Each port node shows:

Element	Meaning
Label	Interface name, for example <code>eth0</code>
Badge	<code>link</code> , <code>no cable</code> , or <code>down</code>
Meta	IPv4 address, link speed, and <code>USB</code> for a USB adapter

Under each port are two nodes:

- **Remote IO**: the devices reached through this port, with their state. A Modbus or EtherNet/IP device is placed by the port its socket actually used; an EtherCAT or Profinet device by the interface it is configured to bind.
- **Listening here**: the servers that answer on this port, currently the **Modbus TCP Slave** and the **OPC UA Server**, with their bind address and whether they are read-only or anonymous. A service bound to `0.0.0.0` is marked *all ports*.

The two branches answer different questions. **IO Devices** is the configuration: what is configured and what is on each rack. **Network Interfaces** is the runtime view: which port each device actually came up on.

7.6.1 The port page

Open a port node to see its page: **IPv4** and netmask, **MAC**, **State**, **Speed**, **MTU**, **Bus** and **Kernel driver**, a table *IO on this port*, and a table *Listening on this port*.

The page warns when:

- the port is a **USB network adapter**. Do not put a fieldbus on it: USB adds latency that no real-time tuning removes.
- a service is bound to `0.0.0.0`, and so answers on every port, including the plant LAN.

7.6.2 Configuring a port's address

To set a controller port's address:

1. Open the port page and click **Configure addressing...**
2. Select **Addressing: Static** or **DHCP**.
3. For static, enter **Address** and **Prefix**. Leave **Gateway** empty on the fieldbus port. **DNS** is optional.
4. Click **Apply**.

The change is made over SSH, through NetworkManager, not through the runtime. The SSH user comes from the target (Chapter 6). The interface restarts immediately.

The same dialog can set the port's **MAC address (Set MAC, Reset to hardware MAC)**. The MAC is stored on the NetworkManager profile and survives a reboot; the burned-in address is never changed. Use a locally administered address (second hex digit 2, 6, A or E).

WARNING: If the port you change is the one the IDE is connected through, the dialog asks you to tick *I understand this will disconnect me*. A wrong address on that port leaves the controller reachable only from its own console.

Rules for a two-port controller:

- Put the two ports on **different subnets**. On one subnet the kernel chooses a port by route metric, and that choice can change after a reboot or a DHCP lease.
- Give the fieldbus port a static address and **no default gateway**. The default route belongs on the plant side only.
- Nothing in the IO configuration selects a port for Modbus or EtherNet/IP. The kernel's routing table does. EtherCAT and Profinet name their interface explicitly.
- An EtherCAT port must not be managed by NetworkManager (Chapter 9). Do not use **Configure addressing...** on it.

7.7 Download hardware and download program

The hardware configuration and the program are downloaded separately, because they are applied differently.

Command	What it sends	What happens on the controller
Download program (F8)	The compiled application	Online change: the program is replaced between two scans. The control loop does not stop (Chapter 20)
Download hardware	controller.json : IO devices, racks, process image, servers, task settings	The runtime restarts to read it. The control loop stops for a few seconds and the outputs go to their stop behaviour
Download all	Hardware, then the program	Both, in that order

The commands are under **Online ▾ → ⚡ Download & Upload ▸**, and on the left rail.

The configuration is read once, at runtime start. That is why a hardware change needs a restart and cannot be applied online.

To download the hardware configuration:

1. Log in as an engineer.
2. Choose **Online ▾ → ⚡ Download & Upload ▸ → ⚡ Download hardware**.
3. Read the confirmation. It gives the number of enabled devices and says the control loop stops. Confirm.
4. Watch the **Messages** panel. Each step is reported: *probe, runtime present, copy configuration, permissions, restart*.

The IDE keeps your login across the restart and reconnects. It then compares the configuration the controller loaded with the project's, and the devices stop showing `not downloaded`.

Important: A hardware download restarts the runtime onto its **boot program**, the last program downloaded. If you changed IO addresses and the program together, use **Download all**. It sends the hardware, waits for the controller to come back, and then sends the program, so the controller never runs new IO against old logic. If the program step fails, the IDE says so in red: the controller is then running the previous program.

WARNING: During a hardware download on a real controller, the drivers stop and the outputs take their **On stop** behaviour and then the fieldbus devices' own watchdog behaviour. Make sure the machine is in a state where that is safe.

A hardware download to the **simulator** target replaces every device with a simulated one of the same name and the same `%I` / `%Q` sizes. The addresses and the symbol table are identical to the controller's, while no hardware is touched (Chapter 26).

The runtime itself is not changed by a hardware download. Installing a new runtime is **Target...** → **Deploy** (Chapter 6).

7.8 How the process image is built from the devices

When the runtime starts, it walks the `io` array in order. For each **enabled** device it asks the driver how many bytes of `%I` and `%Q` it needs, and gives it the next free slice:

```
io[0] cn8031_1   needs %I 12 B, %Q 4 B   -> %I 0..11,   %Q 0..3
io[1] (disabled)                -> nothing
io[2] pn0        needs %I 36 B, %Q 24 B -> %I 12..47,  %Q 4..27
```

The rules are:

- Slices are assigned in configuration order, from byte 0, with no gaps.
- A disabled device gets no slice. An inhibited device keeps its slice.
- `%I` and `%Q` are sized to what the drivers need, unless `process_image.input_bytes` or `output_bytes` in `controller.json` reserve a larger size. A reserved size smaller than the drivers need is a startup error.
- `%M` is not touched by drivers. Its size is **%M marker bytes** in **Controller Settings** (default 4096).
- Within its slice, each device places its blocks at the offsets in its configuration.

The size each driver needs is determined like this:

Driver	Size comes from
Modbus TCP	The blocks: bits rounded up to bytes, registers × 2
EtherCAT	The segment itself, read at startup; or <code>override_input_bytes</code> / <code>override_output_bytes</code> computed from the ESI; or <code>input_bytes</code> / <code>output_bytes</code> when the bus is absent and Require bus at startup is off
Profinet	<code>override_input_bytes</code> / <code>override_output_bytes</code> , computed from the GSDML for the fitted rack
Simulated	<code>input_bytes</code> / <code>output_bytes</code>

Important: Block offsets are stored per device, counted from the start of that device's slice. The IDE lays the devices out the same way the runtime does - enabled devices end to end in configuration order, disabled devices skipped - and adds each device's base wherever it shows or declares an address: the device tree, the device page (block and card tables, aliases, the signal-range dialog), **Browse IO**, the generated alias **AT** declarations and the **Add Device** preview (a new device starts where the existing devices end). In the example above, the first input word of `pn0` is shown and declared as `%IW12`, not `%IW0`. The offsets written to `controller.json`, including the signal-range (scaling) offsets, stay relative to their device. The IDE sizes each device from its configuration: a Modbus TCP device from the end of its blocks, EtherCAT and Profinet from `override_input_bytes` / `override_output_bytes` (else `input_bytes` / `output_bytes`, else the blocks), EtherNet/IP from `t2o_size` / `o2t_size`, a simulated device from `input_bytes` / `output_bytes` (default 32). An EtherCAT segment with no override is sized by the bus at startup; if the bus reports a different size, the devices after it move. **Compare...** reports each device's size as the controller built it.

A program whose **AT** address lies beyond the end of the image is refused when it is loaded: 'x' is at %I40, which needs %I to be at least 42 bytes, but it is 36 - add the IO device this address belongs to, or drop the AT clause.

7.9 Device states and diagnostics

Every driver reports one of these states:

State	Meaning	Program sees
CONFIGURED / STARTING	Created, not yet exchanging	Fail-safe %I
OPERATIONAL	Every block or slave is exchanging	Live %I
DEGRADED	Some blocks or slaves exchange, some do not; or a card is missing; or a short gap in cyclic data	Live data where available
FAULT	Nothing is exchanging	Fail-safe %I (or held, Section 7.10.2)
STOPPED	The driver was stopped	—

While connected, the device page shows a live line such as:

```
OPERATIONAL — 4/4 block(s) up, 0 error(s), 0 timeout(s).
192.168.1.10:502 unit 1, 4/4 block(s) up, via eth0 [192.168.1.11]
Connected for 3h 12m · no reconnects since the runtime started
```

Connected for and the reconnect count together answer the question actually asked on site: has this link been solid since I arrived? A count alone cannot tell a link that dropped an hour ago from one that dropped last week.

The **IO** lamp on the controller summarises all devices:

Lamp	Meaning
IO solid	Every configured device is OPERATIONAL
IO blink	At least one configured device is not OPERATIONAL
IO off	No remote IO is configured

Inhibited devices and the simulated driver are not counted.

7.10 What outputs and inputs do on STOP and FAULT

Three per-device settings decide what the field sees when things stop. Do not confuse them:

Setting	Acts on	Where
On stop — outputs (<code>output_on_stop</code>)	Physical outputs of this device when the controller stops	Device page
On fault — outputs (<code>output_on_fault</code>)	Physical outputs of this device when the program faults	Device page
<code>hold_last_on_fault</code>	What the program reads in <code>%I</code> when this device is not exchanging	<code>controller.json</code> only

7.10.1 Output behaviour: clear or hold

Each of the two output settings is either:

- **clear — drive to safe state (default):** the outputs are driven to zero.
- **hold — keep last written values:** the outputs keep the last values the program wrote.

When each one acts:

Event	Effect
Program FAULT (a runtime error, a critical device, overruns with the <code>fault</code> policy)	On the transition into FAULT, the scan engine zeroes the <code>%Q</code> slice of every device set to clear , writes the outputs once, and then stops driving outputs from the faulted program. Devices set to hold keep their last values
Controller stop (runtime stopped or restarted, including a hardware download)	Modbus TCP: clear writes zeros to every output block before the connection closes; hold leaves the last values on the head station and writes an audit record. EtherCAT: the master takes the segment to INIT, and every slave applies its own fail-safe outputs. Profinet: the connection is released and the device applies its own substitute values
Pause	Not a stop. IO keeps running, the program is frozen, outputs hold their last values
Config mode	Outputs stop being driven and the fieldbus watchdogs release them (Chapter 21)

For EtherCAT and Profinet the device's own fail-safe or watchdog configuration is authoritative on stop. **On stop — outputs** is honoured on those buses only where the driver can. In the current release it is acted on by the Modbus TCP driver only.

WARNING: hold keeps a valve open, a heater on, or a drive running after the program has stopped controlling it. Choose it only where holding is the documented safe state, and record why. Whatever you choose, a power failure, a crash or a cable pull falls back to the module's own watchdog behaviour, which is configured in the module, not here.

7.10.2 Input behaviour: hold_last_on_fault

When a device is not `OPERATIONAL` or `DEGRADED`, the program does not see stale bytes. Its `%I` slice is replaced every scan with either:

- the **fail-safe pattern**, all zeros (the default, `"hold_last_on_fault": false`), or
- the **last good values** (`"hold_last_on_fault": true`).

This setting is only in `controller.json`:

```
{
  "name": "cn8031_1",
  "driver": "modbus_tcp",
  "mode": "async",
  "hold_last_on_fault": false,
  "output_on_stop": "clear",
  "output_on_fault": "clear",
  "params": { "host": "192.168.1.10", "blocks": [ ] }
}
```

Zeros are the right default for most machines, because they can be made safe by design: wire every permissive so that 0 means "not safe". An E-stop chain, a guard switch and a stop button wired normally closed all read `FALSE` when the device drops out, and the logic stops the machine without any extra code.

```
PROGRAM Conveyor
VAR
  (* Wire every permissive so that 0 means "not safe". When the
   remote IO device stops exchanging, its %I slice drops to the
   fail-safe pattern (all zeros) and every permissive below drops
   with it. *)
  estop_ok    AT %IX0.0 : BOOL;  (* NC E-stop chain: TRUE = healthy    *)
  guard_shut  AT %IX0.1 : BOOL;  (* guard switch: TRUE = closed  *)
  start_pb    AT %IX0.2 : BOOL;  (* NO push button              *)
  stop_pb_ok  AT %IX0.3 : BOOL;  (* NC stop button: TRUE = not pressed *)
  motor_run   AT %QX0.0 : BOOL;
  permissive  : BOOL;
  running     : BOOL;
END_VAR
permissive := estop_ok AND guard_shut AND stop_pb_ok;
IF NOT permissive THEN
  running := FALSE;
ELSIF start_pb THEN
  running := TRUE;
END_IF;
motor_run := running;
END_PROGRAM
```

Use **hold last** for inputs only where a stale value is safer than zero, for example a level transmitter whose zero would start a fill pump. Then detect the stale data in the program: see the heartbeat example in Chapter 8.

Note: Analogue scaling (Chapter 11) is applied after the fail-safe pattern, so a held or cleared channel reaches the program in the same engineering units as a live one.

7.11 Checklist before a hardware download

- ☐ Every device has a unique name, and there are no more than 16 `io` entries.
- ☐ Devices that are not wired yet are **Inhibited**, not disabled or deleted.
- ☐ Modbus register maps have been checked against the head station.
- ☐ EtherCAT and Profinet racks were read from the hardware (scan), or re-scanned after being laid out offline.
- ☐ **On stop** and **On fault** are set per device, and the choice is documented.
- ☐ **Critical** is set only where running blind is the hazard.
- ☐ The fieldbus port has a static address and no gateway.
- ☐ You will use **Download all** if the program depends on the new addresses.

8. Modbus TCP devices

This chapter describes the Modbus TCP client driver: how to add a Modbus TCP IO device, how its data is described as blocks, how it is polled, which addresses it produces, how to handle byte and word order, and how to read its diagnostics. You need it for ODOT CN-8031 head stations and for any other Modbus TCP server: drives, power meters, weighing terminals, third-party IO. For the controller acting as a Modbus TCP **server** for a SCADA system, see Chapter 31.

8.1 Overview

The driver name is `modbus_tcp`. One IO device is one TCP connection to one Modbus server at one unit id. The driver:

- polls a list of **blocks**, one Modbus request per block per cycle
- copies each read block into `%I` and each written block from `%Q`
- reconnects with back-off (250 ms growing to 5 s) when the connection drops
- runs either in its own thread (**async**) or inside the scan (**sync**)

The controller does not discover a Modbus register map. Modbus carries no module inventory, so every block is declared in the configuration. A declared map that is wrong fails visibly at commissioning. A map guessed automatically that is subtly wrong produces a controller that runs and does the wrong thing.

8.2 Adding a Modbus TCP device

There are three ways to add one:

Method	Use it when
Add Device → CN-8031	An ODOT C-series rack. The IDE assigns register addresses from its card catalogue
Add Device → Modbus TCP device	Any other server. You enter the blocks from the device manual
Scan for RIO... → Scan for remote IO...	You want the controller to find the servers on a subnet and size their object spaces

8.2.1 An ODOT CN-8031 rack

To add a CN-8031 head station:

1. In the **Devices** pane, click **+**.
2. Click **CN-8031** under **1 · Device type**.
3. Enter an **Instance name**, for example `rack1`. Enter **Host / IP, Port** (502) and **Unit id** (1).
4. Leave **Exchange mode** on **async — driver thread (recommended)**.
5. Under **3 · Modules (slot order)**, add each card in rail order with **Add to rack**. Check the addresses shown beside each card and the total in the note under the list.
6. Click **Add device**.

The device is created with a 10 ms poll cycle and a 50 ms response timeout. The IDE writes one block per card, named after the card and its slot (for example `CT-3108 slot2`); a CT-1616 mixed card becomes two blocks, the second named `(out)`.

WARNING: The catalogue's register addresses assume the ODOT defaults for a rack populated left to right with no gaps. Compare every block against the CN-8031's web page or the ODOT internal-bus mapping table before you connect outputs. Module order determines the map.

8.2.2 Any other Modbus TCP server

To add a generic Modbus TCP server:

1. Click **+**, then **Modbus TCP device**.
2. Enter **Instance name**, **Host / IP**, **Port** and **Unit id**.
3. Click **Add device**.

The device is created with a 20 ms poll cycle, a 100 ms response timeout and **no blocks**. A Modbus device with no blocks is refused when the runtime starts (*params.blocks must be a non-empty array*), so add the blocks before you download.

Note: The IDE has no block editor for Modbus devices. Blocks come from the Add Device dialog (CN-8031), from a scan, or from editing the device's `params.blocks` in `controller.json` (Appendix H). After a hand edit, reopen the project so the IDE reads the file again.

8.2.3 From a scan

Scan for remote IO... (Chapter 7, Section 7.5.1) finds every server in an IP range and measures, for each object space, the largest block it will read starting at address

0. A selected device is added with one block per object space: `DI` (discrete inputs),

`AI` (input registers), `DO` (coils) and `AO` (holding registers).

That is a correct description of what the server will read. It is not a card list. Split the blocks into per-card blocks in `controller.json` if you want each card named in the tree and in the alias table.

8.3 Blocks

A block is one Modbus request, repeated every poll cycle. Its fields:

Field	Meaning
<code>name</code>	Shown in the device tree and in the alias table. Use the physical slot, for example <code>AI-8 slot2</code>
<code>fc</code>	The Modbus object type: <code>coils</code> , <code>discrete_inputs</code> , <code>holding_registers</code> , <code>input_registers</code>
<code>addr</code>	Start address, 0-based as sent on the wire . Not the 40001 / 30001 convention
<code>count</code>	Number of bits (coils, discrete inputs) or registers
<code>area</code>	"I" (read into <code>%I</code>) or "Q" (written from <code>%Q</code>)
<code>offset</code>	Byte offset inside this device's slice. Omit it to place the block directly after the previous block in the same area

8.3.1 Object types and function codes

	Area	Function code used	Process image per item
<code>discrete_inputs</code>	I only	02 Read Discrete Inputs	1 bit, packed 8 per byte
<code>coils</code>	Q	15 Write Multiple Coils	1 bit, packed 8 per byte
<code>coils</code>	I	01 Read Coils	1 bit (read back what the device holds)
<code>input_registers</code>	I only	04 Read Input Registers	16 bits, one word
<code>holding_registers</code>	Q	16 Write Multiple Registers	16 bits
<code>holding_registers</code>	I	03 Read Holding Registers	16 bits (settings, status, or read-back)

Writes always use the "multiple" function codes (15 and 16), even for a block of one coil or one register. A server that only implements 05 and 06 cannot be written by this driver.

Mapping a writable type (`coils` , `holding_registers`) to `%I` is allowed and useful: it reads back what the head station actually applied to an output, or reads a drive's parameter registers. Mapping a read-only type (`discrete_inputs` , `input_registers`) to `%Q` is refused at startup.

8.3.2 Limits

Limit	Value	What happens
Blocks per device	64	Startup error <i>too many blocks (max 64)</i>
Registers per read block	125	Startup error naming the block; split it
Registers per write block	123	Startup error; split it
Bits per block	2000	Startup error; split it
<code>count</code>	at least 1	Startup error
Overlap	Two blocks may not share bytes in one area	Startup error naming both blocks and their byte ranges
Devices	16 IO devices per controller, all drivers together	See Chapter 7

Large blocks are never split silently. If the controller split a block, you would no longer know which requests were on the wire, and a partial failure would be invisible.

8.3.3 Example

```
{
  "name": "rack1",
  "driver": "modbus_tcp",
  "mode": "async",
  "enabled": true,
  "params": {
    "host": "192.168.10.20",
    "port": 502,
    "unit_id": 1,
    "cycle_ms": 10,
    "response_timeout_ms": 50,
    "connect_timeout_ms": 1000,
    "fail_strikes": 3,
    "blocks": [
      { "name": "DI-16 slot1", "fc": "discrete_inputs", "addr": 0, "count": 16, "area": "I", "offset":
0 },
      { "name": "AI-4 slot2", "fc": "input_registers", "addr": 0, "count": 4, "area": "I", "offset":
2 },
      { "name": "DO-16 slot3", "fc": "coils", "addr": 0, "count": 16, "area": "Q", "offset":
0 },
      { "name": "AO-2 slot4", "fc": "holding_registers", "addr": 0, "count": 2, "area": "Q", "offset":
2 }
    ]
  }
}
```

This produces a slice of %I 10 bytes and %Q 6 bytes.

8.4 Addresses produced

A block's channels land at consecutive addresses from its offset :

Block	Channel <i>n</i> is at	Type the alias gets
Bits (coils , discrete_inputs)	%IX(offset + n / 8).(n mod 8) or %QX...	BOOL
Registers	%IW(offset + 2n) or %QW...	INT

For the example in Section 8.3.3 (the first device, so its base is 0 and the addresses equal the offsets):

Signal	Address
DI slot 1, input 0 ... 15	%IX0.0 ... %IX1.7
AI slot 2, channel 0 ... 3	%IW2 , %IW4 , %IW6 , %IW8
DO slot 3, output 0 ... 15	%QX0.0 ... %QX1.7
AO slot 4, channel 0 ... 1	%QW2 , %QW4

The device page lists every block under **Modules — process image mapping**, with its object type, Modbus address, count and process-image range, for example %IW2 ... %IW8 . Give the channels names in **Channel aliases** (Chapter 11); the program then uses the names and never the addresses.

Note: The offsets in `controller.json` are relative to the device's own slice. The runtime stacks the enabled devices end to end, and the IDE adds the same base: for a second or later device, the device page, **Browse IO** and the alias declarations show the offset **plus** the size of the enabled devices before it. If this device followed one with 12 bytes of `%I`, its AI slot 2 channel 0 would be `%IW14`. See Chapter 7, Section 7.8.

A register is 16 bits. Declare it as `INT` for a signed value, `WORD` or `UINT` for an unsigned value or a bit field. Bits inside a word can be read directly with integer bit access (`status.3` , Chapter 14).

8.5 Byte order and word order

Modbus sends each register most significant byte first. The driver converts every register to the controller's native order as it arrives, and back again on the way out. A single register is therefore always read correctly as `%IW` : **never byte-swap a single Modbus register in the program.**

Values of 32 bits span two registers, and Modbus does not define which of the two holds the high half. Devices differ, and many have a setting for it. The controller's process image is little-endian, so:

Device's word order	32-bit value at registers <i>r</i> and <i>r+1</i>	How to read it
Low word first (<i>r</i> = low)	Lands correctly in a double word	AT <code>%ID...</code> as <code>DINT</code> , <code>UDINT</code> or <code>REAL</code>
High word first (<i>r</i> = high), the more common case	The halves are reversed in a double word	Read two <code>WORD</code> s and combine them in the program

The following program reads a meter whose 32-bit values are high word first. The block is `holding_registers` , address 100, count 6, area `I` , offset 20.

```

PROGRAM MeterRead
VAR
    (* One Modbus block: holding registers 100..105 read into %I,
       block offset 20, so register 100 is at %IW20. *)
    volts_x10    AT %IW20 : INT;    (* reg 100: signed, 0.1 V per count    *)
    status_word  AT %IW22 : WORD;  (* reg 101: bit field          *)
    energy_hi    AT %IW24 : WORD;  (* reg 102: high word of a 32-bit count *)
    energy_lo    AT %IW26 : WORD;  (* reg 103: low word           *)
    flow_hi      AT %IW28 : WORD;  (* reg 104: high word of an IEEE float *)
    flow_lo      AT %IW30 : WORD;  (* reg 105: low word           *)
    volts        : REAL;
    meter_alarm  : BOOL;
    energy_wh    : UDINT;
    flow_m3h     : REAL;
    hi, lo       : DINT;
END_VAR
    volts := INT_TO_REAL(volts_x10) / 10.0;
    meter_alarm := status_word.3;          (* bit 3 of register 101 *)

    (* High word first: combine by arithmetic, never by reading %ID24,
       which would put register 103 in the high half. *)
    hi := energy_hi;
    lo := energy_lo;
    energy_wh := hi * 65536 + lo;

    hi := flow_hi;
    lo := flow_lo;
    flow_m3h := DWORD_AS_REAL(hi * 65536 + lo);
END_PROGRAM

```

DWORD_AS_REAL reinterprets a 32-bit pattern as an IEEE-754 **REAL** without converting its value (Appendix A).

Writing a 32-bit value high word first is the reverse: split it into two words.

```

PROGRAM DriveWrite
VAR
    (* Holding registers 200..203 written from %Q, block offset 0. *)
    speed_ref  AT %QW0 : INT;      (* reg 200: speed, 0.1 Hz per count      *)
    pos_hi     AT %QW2 : WORD;      (* reg 201: high word of a 32-bit target *)
    pos_lo     AT %QW4 : WORD;      (* reg 202: low word                      *)
    ctrl_word  AT %QW6 : WORD;      (* reg 203: control bits                  *)
    speed_hz   : REAL := 25.0;
    target_pos : DINT := 150000;
    run_cmd    : BOOL;
    reset_cmd  : BOOL;
    u          : LINT;
END_VAR
    speed_ref := REAL_TO_INT(speed_hz * 10.0);

    (* Split a 32-bit value high word first. *)
    u := target_pos;
    IF u < 0 THEN u := u + 4294967296; END_IF;
    pos_hi := SHR(u, 16) AND 16#FFFF;
    pos_lo := u AND 16#FFFF;

    ctrl_word.0 := run_cmd;
    ctrl_word.7 := reset_cmd;
END_PROGRAM

```

Tip: Confirm the word order on the bench with a known value before trusting it: set a 32-bit parameter to 65 537 (0x0001_0001) or 1.0 (0x3F80_0000) and look at the two registers in the online view.

8.6 Polling and timing

8.6.1 Parameters

Page field	Key	Default (dialog)	Meaning
Host	host	— (required)	IP address or name of the server
Port	port	502	TCP port
Unit id	unit_id	1	Unit identifier; the slave address behind a gateway
Poll cycle (ms)	cycle_ms	10 (CN-8031), 20 (generic)	Poll period, async mode only
Response timeout (ms)	response_timeout_ms	50 (CN-8031), 100 (generic)	Per request
—	connect_timeout_ms	1000	TCP connect timeout; set in controller.json
—	fail_strikes	3	Consecutive completely failed polls before FAULT ; set in controller.json

8.6.2 One poll

In each poll the driver sends every block once, in configuration order, one request at a time on the one connection:

1. The outputs published by the last scan are taken from the shadow buffer (async), or from %Q directly (sync).
2. Each read block is requested and its data stored; each write block is written.
3. The inputs are published to the scan.

Every request has its own **Response timeout**. A poll as a whole has a budget of two poll cycles (async) or *blocks × response timeout* (sync). A block that has no time left in the budget is skipped for that poll and keeps its previous value, so the image is never half-written.

The time one poll takes is roughly the sum of the round trips of all blocks. With a CN-8031 on a dedicated switch, a round trip is about 1 ms. Keep the number of blocks small by grouping contiguous registers into one block where the device allows it.

8.6.3 Async and sync

In **async** mode the driver thread polls at **Poll cycle (ms)** on its own schedule, and the scan only copies a buffer. A slow or absent server never stretches the scan. A poll that overruns its cycle is not caught up: the next one starts a full cycle later.

In **sync** mode the poll happens inside the scan: reads and writes of all blocks are done in one pass at the start of the scan, using the outputs of the previous scan. The runtime warns at startup with the worst case, for example:

```
modbus 'rack1' is in SYNC mode: worst-case 4 transaction(s) x 50 ms = 200 ms is
charged to every scan. Use "mode":"async" if the task period is anywhere near that.
```

Use sync only when the scan is slow compared with the round trip, and the device is on a dedicated network. Sync gives you inputs that are exactly one scan old, which is sometimes needed for a test rig.

8.7 Connection handling and failures

Situation	Driver behaviour	State
Server not reachable at start	Warns <i>initial connection ... failed - the driver will keep retrying with backoff</i> , and starts anyway	STARTING / FAULT
Connection lost	Reconnects with back-off, 250 ms growing to 5 s	FAULT
Every block answered	—	OPERATIONAL
Some blocks failed, some answered	Keeps exchanging the healthy blocks	DEGRADED
Every block failed, fewer than <code>fail_strikes</code> times in a row	Keeps the last state	unchanged
Every block failed <code>fail_strikes</code> times in a row	Logs <i>N consecutive failed poll(s) - FAULT</i> with the last error	FAULT
Modbus exception from the server (for example <i>illegal data address</i>)	The block counts as failed; the connection is kept	DEGRADED or FAULT
Transport error (timeout, reset)	The block counts as failed; the connection is dropped and reopened	as above

One dropped frame on an industrial network is normal. Three polls in a row with nothing answering is not. That is why `fail_strikes` defaults to 3.

While the device is in **FAULT**, the program reads the fail-safe pattern (zeros) in the device's **%I** slice, or the last good values if **hold_last_on_fault** is set (Chapter 7, Section 7.10.2). In **DEGRADED**, a block that failed keeps its previous values.

A stopped controller writes zeros to every output block before it closes the connection, unless **On stop — outputs** is **hold** (Chapter 7, Section 7.10). The head station's own watchdog then decides what happens if the connection is simply lost.

WARNING: Configure the head station's own communication watchdog and output fault state (on a CN-8031, in its web interface). The controller cannot clear outputs over a cable that has been pulled. Only the head station can.

8.7.1 Detecting stale data in the program

A block that keeps failing in a **DEGRADED** device, or a device configured to hold its last inputs, delivers values that look plausible and are old. If the device has a counter it increments by itself, a program can check that it moves:

```
PROGRAM LinkWatch
VAR
    (* The head station publishes a counter that it increments itself -
       many gateways and drives have one. If it stops changing, the data
       is stale even if the link is technically up. *)
    heartbeat AT %IW40 : WORD;
    level_raw AT %IW42 : INT;
    last_hb   : WORD;
    stale     : TON;
    link_ok   : BOOL;
    level     : SCALE;
    level_pct : REAL;
END_VAR
    stale(IN := heartbeat = last_hb, PT := T#2s);
    last_hb := heartbeat;
    link_ok := NOT stale.Q;

    level(raw := level_raw, raw_min := 0, raw_max := 27700,
          eu_min := 0.0, eu_max := 100.0);
    IF link_ok THEN
        level_pct := level.eu;
    END_IF;
    (* otherwise hold the last good value *)
END_PROGRAM
```

SCALE is from the function block library (Appendix C). When the device is in **FAULT** the heartbeat reads 0 every scan, so the same check also catches a lost device.

8.8 Diagnostics

8.8.1 On the device page

While the IDE is connected, the device page shows the live state line (Chapter 7, Section 7.9). For Modbus the detail part is one of:

Detail	Meaning
192.168.10.20:502 unit 1, 4/4 block(s) up, via eth0 [192.168.10.1]	Healthy. <code>via</code> names the controller port the connection actually uses
192.168.10.20:502 last exception: illegal data address	The server answered a request with a Modbus exception. A block's address or count is outside what the server has
192.168.10.20:502 <error>	The last transport error, for example a timeout or a refused connection

The counters are:

Counter	Meaning
<i>N/M block(s) up</i>	Blocks whose last request succeeded, out of all blocks
<i>error(s)</i>	Requests that failed
<i>timeout(s)</i>	Requests that got no answer in time
<i>reconnect(s)</i>	TCP connections re-established since the runtime started
Connected for	How long the current connection has lasted

The `via` port is the only place the actual interface is visible without an SSH session. On a two-port controller, check it: a device reached through the plant LAN port is a routing mistake.

8.8.2 In the controller log

At startup the runtime logs each device's configuration, for example:

```
modbus 'rack1': 192.168.10.20:502 unit 1, 4 block(s) (2 read / 2 write), %I=10 B %Q=6 B, async mode
modbus 'rack1': all 4 block(s) healthy - OPERATIONAL
```

Open the log with **Online** ▾ → **Controller log...** (Chapter 25).

8.8.3 Troubleshooting

Symptom	Likely cause	What to do
FAULT , <i>connection refused</i>	Wrong port, or the server accepts only a limited number of clients and they are in use	Check the port; close other Modbus clients (engineering tools)
FAULT , timeouts, <i>via</i> shows the wrong port	The controller routes to the device through the wrong interface	Put the two ports on different subnets (Chapter 7, Section 7.6.2)
DEGRADED , <i>illegal data address</i>	One block's <code>addr</code> or <code>count</code> is outside the device's map	Compare the block with the device manual; remember addresses are 0-based
DEGRADED , <i>illegal function</i>	The device does not support that object type, or not the multiple-write codes 15/16	Map the data to another object type, if the device offers one
<i>gateway path unavailable or gateway target device failed to respond</i>	A Modbus TCP-to-serial gateway cannot reach the unit id	Check Unit id and the serial side of the gateway

Symptom	Likely cause	What to do
<i>slave device busy</i>	The server is processing a long request	Lengthen Poll cycle (ms) ; reduce the number of blocks
Values look like garbage in a 32-bit variable	Word order	See Section 8.5
A single register looks byte-swapped	It is not: the driver converts registers. The value itself is encoded differently (BCD, scaled, signed/unsigned)	Check the device manual's data format
Scan overruns after adding a device	Sync mode on a fast scan	Switch the device to async
Startup refused: <i>overlaps</i>	Two blocks share bytes	Fix the <code>offset</code> values, or omit them to auto-pack
Startup refused: <i>is read-only and cannot be mapped to %Q</i>	<code>discrete_inputs</code> or <code>input_registers</code> with <code>area Q</code>	Use <code>coils</code> / <code>holding_registers</code> for outputs

9. EtherCAT

This chapter describes the controller's EtherCAT master: choosing the network port, scanning a segment, filing the vendor's ESI files, laying out and editing a modular rack, the start-up SDOs, distributed clocks and the cycle time, and the bring-up procedure with the failure codes you will meet. Read it before connecting an EtherCAT segment to the controller, and keep Section 9.10 at hand during commissioning.

9.1 Status and scope

The EtherCAT master is built on the open-source SOEM stack and runs as the `ethercat` IO driver. It has been exercised on the bench against an ODOT CN-8033 modular coupler; that work is where the ESI-based configuration in Section 9.6 comes from. It has **not** been validated for production use, and parts of it (distributed-clock tuning, recovery after a slave is unplugged) have not been proven on hardware.

WARNING: Walk the bring-up procedure in Section 9.9 on a bench, with the outputs disconnected from anything that can move, before you let the controller drive a machine over EtherCAT.

What is supported:

Item	Support
Masters per controller	1
Exchange mode	async only; sync is refused
Cycle time	125 µs minimum accepted by the driver; start at 2000 µs
Process image	4096 bytes for the whole segment (inputs and outputs together)
Fixed-image slaves	Any number, up to the 4096-byte image
Modular coupler	One per segment , and it must be the only slave, at position 1, when a module list, PDO assignment, start-up SDOs or image override is configured
Cards behind a modular coupler	Up to 64 module idents; up to 64 PDOs per SyncManager
Mailbox	CoE SDO writes at start-up (module list, PDO assignment, start-up SDOs)
Distributed clocks	SYNC0 on every DC-capable slave, proportional drift correction
Not supported	Multiple groups, cable redundancy, EoE, FoE, a second master

9.2 Preparing the controller

9.2.1 The network port

An EtherCAT master owns its network port. It sends raw Ethernet frames and expects every frame back. Choose the port with these rules:

- Use a **dedicated** port. Do not share it with the plant LAN or with Modbus or Profinet traffic: a broadcast storm on a shared port becomes a missed cycle.

- Do not use a **USB** network adapter. The scan dialogs mark it *USB, not suitable*, and the port page warns about it (Chapter 7, Section 7.6).
- Take the port away from NetworkManager, so it does not reconfigure the NIC under the master. Do not use **Configure addressing...** on this port. It needs no IP address.

On the controller, as an administrator:

```
sudo ip link set eth1 up
sudo nmcli device set eth1 managed no
```

9.2.2 Runtime privileges and kernel

The master needs a raw socket. The runtime service is installed with the capabilities it needs (`CAP_NET_RAW` , `CAP_SYS_NICE` , `CAP_IPC_LOCK`). If they are missing, the driver fails with *cannot open eth1 (...). An EtherCAT master needs a raw socket: grant CAP_NET_RAW, and check the interface is up and not managed by NetworkManager.*

Distributed clocks need a real-time kernel (PREEMPT_RT) to hold phase. Without one, run with DC off and a conservative cycle. The runtime warns at startup when the kernel is not real-time.

9.3 ESI files

An ESI (EtherCAT Slave Information) file is the vendor's XML description of a slave: its identity, its PDOs, its modules, its start-up commands. The IDE reads ESI files; the controller never does. Everything the controller needs from the ESI is computed in the IDE and written into `controller.json` as numbers.

9.3.1 Where to put them

Import them with **Project ▾ → Import device description (GSDML / ESI / EDS)...** (the window is described in Section 10.3.2), or put the vendor's `.xml` files in an `esi` folder by hand. The IDE looks in two places, in this order:

1. `esi\` inside the project folder
2. `esi\` in the IDE installation folder

The first file that defines a module ident wins, so a project can pin a particular ESI revision without changing what other projects use. The same folders also hold Profinet GSDML files and EtherNet/IP EDS files; the IDE tells them apart by content.

The files are read each time a dialog needs them, so a file you drop into the folder is used at the next scan or rack edit. No restart is needed.

Important: Use the vendor's **full** ESI. Some vendors also ship a reduced file (ODOT's "NS" variant) that has no module list object and no port parameters, and expects the rack to be configured with the vendor's own tool.

9.3.2 What the IDE takes from an ESI

ESI content	Used for
Vendor id, product code, revision	Matching a scanned coupler to its description
Modules (<code>ModuleIdent</code> , <code>name</code> , <code>ModuleClass</code>)	Naming the cards a scan reports, grouping the rack editor's card list

ESI content	Used for
PDOs and their entries	The size of each card, its channels, their width and names
<code>DependOnSlot</code> , <code>SlotPdoIncrement</code> , <code>SlotIndexIncrement</code>	Resolving slot-dependent PDO and object indices (Section 9.6.3)
<code>InitCmds</code> with transition <code>PS</code> or <code>IP</code>	Start-up SDOs (Section 9.7)
DC operation modes	Whether Distributed clocks can be offered
SyncManager sizes, <code>MaxInstances</code>	Warnings when a rack exceeds what the coupler can carry

9.4 Scanning a segment

A scan asks the segment what is on it. It is the preferred way to create the EtherCAT device, because the slaves report exactly what they are.

To scan an EtherCAT segment:

1. Connect the IDE to the controller and log in as an engineer.
2. Open **IO Devices** → **Scan for RIO...** → **Scan EtherCAT segment....**
3. Select the **Interface**. A USB adapter is marked *USB, not suitable*.
4. Leave **Cycle (µs)** at 2000 and **Distributed clocks** off for the first bring-up.
5. Optionally, in **Name cards from ESI**, pick the vendor's ESI to label the cards for this scan. This is only needed if the ESI is not already in the `esi` folder.
6. Click **Scan**.

The scan takes the segment to PRE_OP to read the mailboxes, then to SAFE_OP to measure the process image, then back to INIT. It never reaches OPERATIONAL, so no outputs are driven.

Note: A scan is refused while the controller's own master is cycling the segment, because SOEM keeps its bus state in globals and the scan would corrupt it. The message tells you to set `"enabled": false` on the EtherCAT device and restart the runtime (**Download hardware**), then scan. Stopping the runtime instead would close the connection the scan request arrives on. A master configured with **Require bus at startup** off, whose segment never answered, does not hold the bus and does not block a scan.

9.4.1 The result

The result table lists one row per slave:

Column	Meaning
#	Position on the segment, from 1
Name	The name the slave reports
Vendor, Product, Rev	Identity from the slave's EEPROM
%I, %Q	Bytes the slave contributes, as measured
DC	The slave has a distributed-clock unit

Under a **modular coupler** a second table lists the cards it reports: **Slot**, **Module ident**, **Card** (from the ESI), **%I bit** and **%Q bit**. The cards behind a modular coupler are not EtherCAT slaves. They are modules on the coupler's own backplane, and the coupler reports them as 32-bit identifiers in object `0xF050` (detected modules). The wire carries no part numbers; the ESI turns the identifiers into names.

The scan also reads `0xF030` (configured modules) and explains what it found:

Message	Meaning
<i>N module(s) already configured in the coupler.</i>	The coupler holds a module list
<i>The coupler has no configured module list, so it builds no process image...</i>	Normal for a new rack. Adding the segment writes the detected idents at startup
<i>The coupler would not let its configured module list be read (0xF030 refused)...</i>	Some couplers do not implement the object. Not necessarily fatal; see Section 9.6
<i>The coupler answered 0xF050 with zero modules...</i>	No cards seated, or the rack's field side has no 24 V

9.4.2 Adding the segment

Click **Add this segment**. The IDE creates one device named `ecat0` with:

Key	Value
<code>ifname</code> , <code>cycle_us</code> , <code>dc</code>	From the dialog
<code>expected_slaves</code>	The number of slaves found
<code>require_bus</code>	<code>true</code>
<code>product_code</code>	The coupler's product code, used later by the rack editor
<code>modules</code>	The detected card idents, in slot order
<code>pdo_assign</code>	The PDO assignment for those cards, computed from the ESI
<code>startup_sdo</code>	The ESI's start-up commands for the coupler and each card
<code>override_input_bytes</code> , <code>override_output_bytes</code>	The process image computed from the ESI
<code>blocks</code>	One addressed block per card and direction, for aliases and <code>AT</code> declarations

If the ESI is missing, or a detected card is not in any filed ESI, the PDO assignment is refused and a warning names the missing slots. The segment is still added, but a modular coupler without an assignment has no process image. File the ESI and edit the rack (Section 9.6) to complete it.

Only one EtherCAT master is possible. If the project already has one, **Add this segment** refuses with *This project already has an EtherCAT master — only one is possible*. Remove the old device first.

The new device reaches the controller with **Download hardware** (Chapter 7, Section 7.7).

9.5 Laying out a rack from an ESI, offline

Add EtherCAT rack from ESI... lays out a rack with no bus present: before the panel is built, or on a desk. It is a drawing of the rack you say will be fitted.

To lay out a rack from an ESI:

1. Open **IO Devices** → **Scan for RIO...** → **Add EtherCAT rack from ESI...**
2. In **ESI file**, pick the vendor's `.xml` (some vendors use `.exml`).
3. Select the **Coupler**. Modular couplers are marked (*modular*).
4. Select the **Interface**, **Cycle (µs)**, **Distributed clocks** (disabled when the coupler declares no DC mode), and **Require bus at startup**.
5. Under **Fit cards, in physical slot order**, select each card and click **Add card**. The list is grouped by the class the coupler declares (DI, DO, AI, AO...). Gateway submodules are listed last: they belong inside a gateway card, not on the rail. **Clear** empties the rack.
6. Check the table: **Slot**, **Card**, **%I bit, bits**, **%Q bit, bits**. Offsets are shown in bits, because two 8-channel digital cards share one byte.
7. Click **Add this segment**.

The device gets `expected_slaves` 1 (the cards are not slaves), the module list, the start-up SDOs, and `input_bytes / output_bytes`. With **Require bus at startup** off, the controller starts and runs the program with the EtherCAT driver in **FAULT** and all its inputs at the fail-safe pattern. That is what makes an offline layout usable in the simulator and on a desk.

Important: A rack laid out offline has no addressed blocks, so it has no channel aliases and its device page shows only the slot list. When the hardware is wired, delete the offline device and use **Scan EtherCAT segment... → Add this segment**, which gives the full configuration. Set **Require bus at startup** back on before the panel goes into service: with it off, the controller runs with no EtherCAT IO and does not stop.

9.6 The rack of a modular coupler

A modular coupler such as the ODOT CN-8033 is one EtherCAT slave with cards behind it. Before it can build a process image, it must know which cards are fitted and which PDOs to carry. EtherCAT offers three ways to tell it, and couplers differ in which ones they accept. The controller configures all three and uses what the hardware accepts:

Step at start-up (PRE_OP)	Config key	If the coupler refuses
1. Write the configured module list, object <code>0xF030</code>	<code>modules</code>	Warning; the coupler keeps whatever list it holds. Not fatal
2. Write the start-up SDOs	<code>startup_sdo</code>	Warning per refused write, unless the entry is <code>required</code> (Section 9.7)
3. Write the PDO assignment, objects <code>0x1C12</code> (outputs) and <code>0x1C13</code> (inputs)	<code>pdo_assign</code>	<i>Read only:</i> warning, the coupler's own assignment is used. Any other refusal: fatal , the segment does not start
4. Take the process image from the configuration	<code>override_input_bytes</code> , <code>override_output_bytes</code>	—

Step at start-up (PRE_OP)	Config key	If the coupler refuses
5. Map the segment and go to SAFE_OP, then OP	—	See Section 9.10

9.6.1 Module list (0xF030)

The module list is written in the order ETG.5001 prescribes: subindex 0 set to 0, the idents written to subindices 1...n, then subindex 0 set to n. After a successful write the master re-reads the segment, because the coupler rebuilds its PDO assignment from the list. A card ident refused for a slot means the card fitted there is not the one the configuration names: *slave 1 rejected module ident 0x20003238 for slot 3. The card fitted there is not the one the configuration names.*

9.6.2 PDO assignment

The PDO assignment tells each SyncManager which PDOs to carry. The IDE computes it from the ESI for the cards in the rack and writes it as, for example:

```
"pdo_assign": {
  "0x1C12": ["0x1600", "0x1601"],
  "0x1C13": ["0x1A00", "0x1A01", "0x1A02", "0x1A03", "0x1A04"]
}
```

A PDO refused in a slot means the card in that slot does not offer it: the rack in the configuration is not the rack fitted.

9.6.3 Slot-dependent indices (DependOnSlot)

Two identical cards in different slots cannot use the same PDO index. In the ESI, a PDO or an InitCmd marked `DependOnSlot="1"` prints a **base** index. The IDE resolves it per slot, counting slots from zero:

What	Resolved index
PDO	$\text{base} + \text{slot} \times \text{SlotPdoIncrement}$
Object (InitCmd)	$\text{base} + \text{slot} \times \text{SlotIndexIncrement}$

So the first card keeps the index the file prints. A PDO without `DependOnSlot` is the same object in every slot and is assigned once. If two cards resolve to the same object, the IDE warns that the slot increment in the ESI does not match the rack.

9.6.4 Image override

Some couplers refuse every configuration path over CoE. The ODOT CN-8033 is one: it has no `0xF030` object (abort `0x06020000`, *object does not exist*), and its `0x1C12` / `0x1C13` are read-only (abort `0x06010002`) and report every PDO index as `0x0000`. The master then sizes the coupler at 0 bytes, and the coupler refuses OPERATIONAL with AL status `0x001E`, *Invalid input configuration*.

For such a coupler the process image comes from the ESI. The IDE computes the total `%I` and `%Q` for the fitted rack and writes `override_input_bytes` and `override_output_bytes`. The driver then marks slave 1 as manually configured and programs its SyncManager lengths directly, and logs:

`ecat 'ecat0': slave 1 process image taken from the configuration, not the bus:
%I 36 B, %Q 24 B. This is correct only if the rack fitted is the rack the ESI was
read for - a card added or moved will silently shift every address after it.`

A verified example is the ODOT rack below: CT-121F, 2 × CT-3238, CT-4234, CT-4158 on a CN-8033, giving %I 36 bytes and %Q 24 bytes, expected working counter 3.

Slot	Ident	Card	%I	%Q
1	0x2000121f	CT-121F, 16 DI	16 bit	—
2	0x20003238	CT-3238, 8 AI	128 bit	—
3	0x20003238	CT-3238, 8 AI	128 bit	—
4	0x20004234	CT-4234, 4 AO	8 bit (diagnostics)	64 bit
5	0x20004158	CT-4158	8 bit (diagnostics)	128 bit

WARNING: With an image override, nothing on the wire checks the rack. A card added, removed or swapped moves every address after it, and the controller exchanges data at addresses that no longer mean what the program thinks. After any change to the physical rack, edit the rack in the IDE (or re-scan) and download the hardware again.

9.6.5 Editing the rack

Once a segment has been added by a scan, its device page shows the card table and an **Edit rack...** button.

To edit a rack:

1. Open the EtherCAT device page and click **Edit rack...**. The dialog is titled *Rack — ecat0*.
2. To add a card, type part of its name, ident or class in the filter box (for example CT-32 or AI), select it, set **at slot** (default: the end), and click **Add card**.
3. Move a card with ↑ and ↓; delete it with **Remove**.
4. Check the note under the table: the new process image size, and any warnings from the ESI (for example a card that does not start on a byte boundary).
5. Click **Save**, or **Cancel** to discard every change.

Saving recomputes the module list, the PDO assignment, the start-up SDOs, the image override and the blocks from the ESI. It keeps:

- **Aliases**, which follow their card. An alias on the card in slot 2 is still on that card after a card is inserted before it, at the card's new addresses. Aliases on a card that was removed are dropped, and a message lists them.
- **Start-up SDO edits** (for example IO-Link port settings, Chapter 12) for every card that is still fitted, following the card if it moved.

The rack editor needs the ESI: if no ESI is filed, it says so and cannot edit. It also does not read the bus.

Important: After **Save** the controller still has the old rack. Use **Download hardware**, then **Download program** (or **Download all**), and re-scan once the hardware is attached to confirm the two agree.

9.6.6 Blocks and addresses

Each card contributes one block per direction, named `<card>_<slot>`, with `_I` or `_Q` added when the card has both. Offsets accumulate in bits across the whole rack, in slot order, so two 8-bit digital cards share a byte. The width of a channel comes from the ESI (1, 8, 16 or 32 bits), and that sets the address form and the alias type:

Channel width	Address	Alias type
1 bit	<code>%IX...</code> / <code>%QX...</code>	<code>BOOL</code>
8 bits	<code>%IB...</code> / <code>%QB...</code>	<code>BYTE</code>
16 bits	<code>%IW...</code> / <code>%QW...</code>	<code>INT</code>
32 bits	<code>%ID...</code> / <code>%QD...</code>	<code>DINT</code>

EtherCAT process data is little-endian, like the controller's process image, so 16- and 32-bit channels are read directly with no byte swapping.

The device page's card table shows each card's **Slot**, **Card**, **%I — inputs**, **%Q — outputs**, **Type** (for example $8 \times AI \cdot 0-20mA/4-20mA$) and **Signal range** (Chapter 11). An analogue output card also has an input byte: that byte carries per-channel diagnostics (wire break, over-temperature), and it is real space in the input image. Removing it from the layout would shift every address after the card.

A program for the rack in Section 9.6.4, with four channels named on the device page. The `VAR_GLOBAL` block is what the IDE generates from the aliases at build time (Chapter 11); you do not type it.

```

VAR_GLOBAL
  start_pb  AT %IX0.0    : BOOL;  (* ecat0 / CT-121F_1 Ch#0 *)
  stop_pb   AT %IX0.1    : BOOL;  (* ecat0 / CT-121F_1 Ch#1 *)
  tank_level AT %IW2      : INT;   (* ecat0 / CT-3238_2 Ch#0 *)
  valve_pos  AT %QW0      : INT;   (* ecat0 / CT-4234_4_Q Ch#0 *)
END_VAR

PROGRAM Main
VAR
  running : BOOL;
END_VAR
  running := (start_pb OR running) AND NOT stop_pb;
  IF running THEN
    valve_pos := tank_level / 2;
  ELSE
    valve_pos := 0;
  END_IF;
END_PROGRAM

```

9.7 Start-up SDOs

Many slaves need parameters written before they go to `SAFE_OP`: an input filter time, an analogue range, an IO-Link port mode. The ESI declares these as `InitCmds`. The IDE takes every `InitCmd` with transition `PS` (`PRE_OP` to `SAFE_OP`) or `IP` (`INIT` to `PRE_OP`) from the coupler and from each card, resolves slot-dependent indices (Section 9.6.3), and writes the list to the device as `startup_sdo`:

```
"startup_sdo": [
  { "index": "0x2000", "sub": 1, "data": "00", "slot": null, "comment": "Input Fault Action" },
  { "index": "0x8003", "sub": 3, "data": "01", "slot": 4, "comment": "Port Mode" },
  { "index": "0x8001", "sub": 1, "data": "0A00", "slot": 2, "comment": "Filter time", "required": true
}
]
```

Field	Meaning
<code>index</code> , <code>sub</code>	The object and subindex, after slot resolution
<code>data</code>	The bytes as they go on the wire, in hex, exactly as the ESI prints them (little-endian for a number). The runtime writes them unchanged
<code>slot</code>	The card the write is for, from 1; <code>null</code> for the coupler itself
<code>comment</code>	The vendor's name for the parameter
<code>required</code>	Optional. <code>true</code> makes a refusal fatal

The master writes the list in PRE_OP, **after** the module list (the per-slot objects exist only once the coupler knows what is fitted) and **before** the PDO assignment (a parameter can change what a slot carries). Each entry is one SDO write. It logs a summary:

```
ecat 'ecat0': slave 1 start-up SDOs: 633 written, 0 refused
```

A refused write is a warning, and the start continues with the value the slave already holds: a coupler that keeps its configuration in non-volatile memory still runs, and a controller in a restart loop over one parameter is worse. Mark an entry `"required": true` in `controller.json` when running without it is unsafe; a refusal then stops the start.

Start-up SDOs, like the module list and the PDO assignment, apply only to the slave at position 1, and are refused on a segment with more than one slave.

Tip: An 8-port IO-Link master generates over 600 start-up SDOs. They are edited through the **IO-Link ports** page (Chapter 12), not by hand.

9.8 Cycle time and distributed clocks

9.8.1 Cycle time

Cycle (µs) is the period of the EtherCAT frame. It is independent of the PLC scan: the master thread sends a frame every cycle on its own schedule, and the scan copies a shadow buffer. The driver refuses a cycle below 125 µs.

Start at 2000 µs. When the segment is stable, reduce the cycle in steps (1000, 500, 250 µs) and stop one step above the value at which the working-counter error count starts to grow. On this controller the floor is set by the operating system's scheduling latency, not by EtherCAT.

A cycle faster than the PLC scan is useful only if the slaves themselves use it (for example oversampling cards or drives with their own control loop). For plain IO, a cycle equal to the scan period, or half of it, is enough.

9.8.2 Distributed clocks

With **Distributed clocks** ticked (`"dc": true`), the master enables SYNC0 on every DC-capable slave at the cycle time, and disciplines its own send time to the reference clock with a proportional correction. The driver's diagnostic line shows the phase error, for example `dc_err 850 ns` . It should settle into a stable band within a few seconds. If it walks steadily in one direction, the correction does not suit your cycle time; turn DC off and report it.

Rules:

- Bring the segment up with DC **off** first. Change one variable at a time.
- If no slave supports DC, the master warns and runs without it.
- The offline rack dialog disables **Distributed clocks** when the coupler's ESI declares only SM-synchronous modes (`AssignActivate 0`): enabling a DC unit the slave does not have is refused by the slave.
- `dc_shift_ns` in `controller.json` shifts SYNC0 relative to the frame (default 0).

Note: If the `dc` key is absent from `controller.json` , the driver enables DC. The IDE always writes the key explicitly.

9.8.3 Other parameters

Page field	Key	Default	Meaning
Interface	<code>ifname</code>	<code>eth0</code>	The port the master binds
Cycle (µs)	<code>cycle_us</code>	2000 (IDE)	Frame period
Distributed clocks	<code>dc</code>	off (IDE)	See above
Expected slaves	<code>expected_slaves</code>	0 = accept any	A different count is refused at startup, because the image layout depends on the exact segment
Require bus at startup	<code>require_bus</code>	on	Off: the controller starts even with no segment, the driver sits in FAULT
%I bytes (bus absent)	<code>input_bytes</code>	0	Image size used only when the bus is absent and Require bus is off
%Q bytes (bus absent)	<code>output_bytes</code>	0	As above
—	<code>recovery_ms</code>	500	Timeout used when reconfiguring or recovering a lost slave
—	<code>dc_shift_ns</code>	0	SYNC0 shift

With **Require bus at startup** on, a segment that is absent at startup stops the controller from starting: a fieldbus that was declared and is not there is a fault. With it off and no `input_bytes` / `output_bytes` , the start is refused because the image cannot be sized.

9.9 Bring-up procedure

Do this on a bench, with the outputs disconnected from anything that can move.

Step 1 — Does the bus enumerate?

1. Cable the controller's EtherCAT port to the **IN** port of the first slave. Power the segment, including the field side of each rack.
2. Run **Scan EtherCAT segment...** at 2000 µs, DC off.
3. Check that every slave appears, with its identity and sizes, and that the cards behind a modular coupler are listed and named.

If nothing answers: the cable goes into the IN port, the segment needs its own power, and the port must be up and unmanaged.

Step 2 — Does it reach OPERATIONAL?

1. Add this segment, then **Download hardware**.
2. Open **Online ▾** → **Controller log....** Expect:

```
ecat 'ecat0': 1 slave(s) on eth1, %I=36 B %Q=24 B, expected wkc 3, cycle 2000 us, dc off
ecat 'ecat0':  slave 1: CN-8033 (%I 36 B, %Q 24 B), DC
ecat 'ecat0': running at 2000 us
ecat: segment OPERATIONAL (wkc 3, all slaves in OP)
```

A segment stuck below OP prints each slave's state and its AL status code with the decoded text. That text is the diagnosis (Section 9.10).

Step 3 — Does data move?

1. Go online. Watch an input channel in the device's aliases or the online variables and operate the physical input.
2. Force one digital output (Chapter 22) and check the physical output.
3. Drive one analogue input and check the value and its scaling.

This step is what proves the configuration. Steps 1 and 2 only prove that the bus enumerated and started.

Step 4 — Distributed clocks, then the cycle time.

1. Tick **Distributed clocks**, keep 2000 µs, **Apply**, **Download hardware**. Watch `dc_err` on the device page settle.
2. Reduce **Cycle (µs)** one step at a time, as in Section 9.8.1.

Step 5 — Restore the service settings. Require bus at startup on, **Expected slaves** equal to the segment, and **Critical** ticked if the machine must not run without the segment (Chapter 7, Section 7.4.5).

9.10 Diagnostics and failure codes

9.10.1 States

Driver state	Meaning
OPERATIONAL	Working counter as expected and every slave in OP
DEGRADED	Frames are answered but a slave is not in OP: <i>frames are answered but a slave is not in OP - its outputs are not being driven</i> . The master keeps trying to bring it back
FAULT	Working counter below expected: <i>working counter N < expected M - segment degraded</i> . Inputs are not published; the program sees the fail-safe pattern. The master tries to recover

The master checks each slave's AL state once a second while **OPERATIONAL**, and every cycle while recovering. A slave in **SAFE_OP** still answers frames, so the working counter alone would call it healthy while it drives no outputs; the state check catches that.

Recovery: a slave in **SAFE_OP+ERROR** is acknowledged and asked for **OP**; a slave in **SAFE_OP** is asked for **OP**; a slave in a lower state is reconfigured; a slave that has disappeared is reported **LOST** and recovered when it comes back.

The device page's detail line reads, for example: `eth1 1/1 slave(s) OP, 2000 us, dc off, dc_err 0 ns`. The counters are errors, timeouts and recoveries.

9.10.2 Startup errors

Log message	Cause	Remedy
<i>cannot open eth1 (...). An EtherCAT master needs a raw socket...</i>	Missing CAP_NET_RAW , port down, or port managed by NetworkManager	Section 9.2
<i>no slaves answered on eth1...</i>	Cable in the OUT port, segment unpowered, wrong interface	Check cabling and power; check Interface
<i>found N slave(s), expected M...</i>	A slave was added, removed or is unpowered	Fix the segment, or re-scan and re-add
<i>SYNC mode is not supported...</i>	"mode": "sync"	Use async
<i>only one EtherCAT master instance is possible...</i>	Two ethercat devices	Keep one
<i>cycle_us N is below the 125 us floor...</i>	Cycle too short	Raise Cycle (µs)
<i>"modules" configures the coupler at position 1, but this segment has N slaves...</i>	A modular rack configuration on a segment with more than one slave	One modular coupler per segment, alone
<i>segment needs N B of IOMap, this build has 4096 B</i>	The segment's image exceeds 4096 bytes	Split the IO across controllers
<i>"require_bus" is false but no "input_bytes"/"output_bytes"...</i>	Bus-absent mode without sizes	Enter the %I/%Q bytes (bus absent)
<i>slave 1 refused the module list (0xF030:00)...</i>	The coupler does not implement 0xF030	Usually harmless; see Section 9.6.4
<i>slave 1 rejected PDO 0x..., The card in that slot does not offer it</i>	Rack in the configuration differs from the rack fitted	Re-scan
<i>slave 1 refused required start-up SDO...</i>	A required start-up SDO was refused	Check the value and the card
<i>slave N stuck in state 0x..., StatusCode 0x...: <text></i>	The slave refused a state change; see Table 9-1	Table 9-1

9.10.3 AL status codes

The slave's AL status code says why it refused a state. The log prints the code and the standard text. The ones met most often in commissioning:

Table 9-1 — Common AL status codes

Code	Text	Usual cause here
0x0011	Invalid requested state change	A transition the slave does not allow from its current state; usually after an earlier error. Power-cycle the slave
0x0016	Invalid mailbox configuration	Mailbox SyncManager settings differ from the slave's EEPROM
0x0017	Invalid sync manager configuration	SyncManager lengths or addresses wrong; check the image override against the ESI
0x001B	Sync manager watchdog	Process data stopped arriving: master thread stalled, cable, or cycle too short
0x001D	Invalid output configuration	Output PDO assignment or size does not match what the slave expects
0x001E	Invalid input configuration	Input PDO assignment or size wrong. On a modular coupler: no module list, no assignment and no override (Section 9.6)
0x0030	Invalid DC SYNC configuration	DC enabled on a slave or mode that does not support it
0x0035	Invalid sync cycle time	The cycle is outside what the slave accepts in DC mode

The state is printed as a hex number: 0x02 PRE_OP, 0x04 SAFE_OP, 0x08 OP; 0x10 added means the error flag is set, so 0x12 is PRE_OP+ERROR and 0x14 is SAFE_OP+ERROR.

9.10.4 SDO abort codes

A refused SDO write is logged with its abort code:

Abort code	Meaning
0x06010002	Attempt to write a read-only object. For a PDO assignment the coupler fixes its own; the master continues
0x06020000	Object does not exist. For 0xF030, the coupler has no module list object
0x06090011	Subindex does not exist
0x06070010	Data type does not match, length of service parameter does not match
0x08000022	Data cannot be transferred because of the present device state

9.11 Limits

Limit	Value
EtherCAT masters per controller	1
Process image of the segment	4096 bytes
Modular couplers per segment	1, alone on the segment, at position 1
Module idents in <code>modules</code>	64
PDOs per SyncManager in <code>pdo_assign</code>	64
Minimum cycle	125 µs (driver); practical floor set by the kernel

Limit	Value
Exchange mode	async only
Groups, cable redundancy, EoE, FoE	Not supported

10. Profinet

This chapter describes the controller as a Profinet IO controller: discovering devices with DCP, giving them a station name and an IP address, filing GSDML files, building the rack, the cyclic exchange and its timing, and how to bring a device up and find faults. You need it when the machine's remote IO is a Profinet coupler, such as the ODOT CN-8032.

10.1 Status and scope

The Profinet IO controller is the `profinet` driver. It is built into every runtime and needs no third-party stack. It has been verified on the bench against an ODOT CN-8032 coupler with a five-card C-series rack: discovery, naming and IP assignment, reading the fitted cards, and stable cyclic exchange at a 4 ms cycle with no reconnects.

Know these limits before you plan a machine around it:

Limit	Detail
Devices	One Profinet IO device per controller in this release. Each connection binds UDP port 34964 on the controller, and a second device fails with <i>cannot bind UDP 34964 (another Profinet controller running?)</i>
Parameterisation	At connect the controller writes one parameter record set (MultipleWrite, index <code>0xE040</code>). In this release that record set is the one for the ODOT CN-8032 with default module parameters; it is not generated from the GSDML. Another device, or different card parameters, may refuse it (Section 10.8.3)
Real-time class	RT (cyclic frames on EtherType 0x8892, VLAN priority 6). No IRT
Alarms	Diagnosis alarms are acknowledged at transport level only; they are not shown in the IDE
Topology	No LLDP-based topology, no automatic device replacement by neighbourhood
Timing	Send clock × reduction ratio; the watchdog is floored at about 150 ms (Section 10.7)

WARNING: Commission a new rack on the bench with the outputs disconnected from anything that can move before trusting the controller to drive it.

10.2 Concepts

Term	Meaning
IO controller	The controller that owns the cyclic exchange: here, the Zeus runtime
IO device	The coupler (head station) and its cards
Station name	The device's Profinet name, for example <code>rio-line1</code> . The controller finds the device by this name , not by its IP address
DCP	Discovery and Configuration Protocol. Layer-2 multicast used to find devices, name them, set their IP, and flash their LEDs. It does not cross a router or a VLAN boundary
GSDML	The vendor's XML description of the device: its access point (DAP), the modules that can be plugged, their data sizes and parameters

Term	Meaning
DAP	Device access point: slot 0, the coupler itself, with its interface and port submodules
Slot / subslot	Where a module sits; cards are slots 1, 2, 3...; each has one or more submodules
AR	Application relation: the connection between controller and device, set up by an RPC Connect over UDP 34964
Send clock, reduction ratio	Together they set the cyclic period: $send\ clock \times reduction\ ratio \times 31.25\ \mu s$
Watchdog factor	How many missed cycles the device and the controller tolerate before declaring the connection lost

The setup order on a new device is always:

1. Discover it (DCP).
2. Give it a station name and an IP address on the controller's fieldbus subnet (DCP Set).
3. Read its fitted cards and lay out the rack from the GSDML.
4. Add it to the project and download the hardware.
5. The runtime connects by station name and starts the cyclic exchange.

10.3 Preparing

10.3.1 The network port

- Use a dedicated port for Profinet, or at least one that does not carry the plant LAN.
- Do not use a USB network adapter.
- The controller's port needs an IP address: the Connect is a UDP RPC, so the device's IP must be on the **same subnet** as the controller's port. DCP itself is layer 2 and finds devices on any subnet.

WARNING: Never add an address to the controller's fieldbus port that duplicates its other port's subnet to reach a device. Move the **device** to the fieldbus subnet with DCP Set instead (Section 10.5). Two ports on one subnet can cut the IDE's connection to the controller.

10.3.2 GSDML files

To import a GSDML: **Project** ▾ → **Import device description (GSDML / ESI / EDS)...**, or **IO Devices** → **Scan for RIO...** → **Device descriptions (GSDML / ESI / EDS)...**. Choose **Save to** (*This project* or *All projects on this PC*), click **Import files...** and select the `.xml` — or the vendor's `.zip`, which is opened for you — or drop the files on the window. Each file is read with the same parser the scan uses before it is filed; a file that is not a GSDML, ESI or EDS is refused with the reason, pictures in a zip are left out, and a different file with the same name is not overwritten unless **Replace** is ticked. The window also lists every filed description by protocol, with its device, vendor and module count, and removes one with X. The same window files EtherCAT ESI and EtherNet/IP EDS files (Chapter 9, Chapter 8).

Importing is the same as copying the file by hand: the device's GSDML (`GSDML-V2.xx-<vendor>-<device>-<date>.xml`) goes in the `esi` folder of the project, or in the `esi` folder of the IDE installation, alongside the EtherCAT ESI files. The IDE recognises GSDML files by their content. It reads them each time a dialog opens, so a newly filed file needs no restart.

The IDE matches a discovered device to its GSDML by **vendor id and device id**, the two 16-bit numbers DCP reports. A device whose GSDML is not filed shows *not filed* in the scan, and cannot be added.

From the GSDML the IDE takes the DAP and its interface/port submodules, each module's ident, submodules and data sizes, channel names, the card class (DI, DO, AI, AO), the signal ranges it declares, the supported send clocks, and the module parameter records.

Note: Some vendors publish their GSDML only in encrypted form. The IDE cannot read such a file; ask the vendor for the plain XML.

10.4 Discovering devices

To scan a Profinet network:

1. Connect to the controller and log in as an engineer.
2. Open **IO Devices** → **Scan for RIO...** → **Scan Profinet network....**
3. Select the **Interface** the devices are cabled to.
4. Leave **Listen (ms)** at 2000 (250 to 10 000).
5. Click **Scan**.

Discovery sends one multicast frame (DCP Identify-All) and collects the answers for the listen time. It drives no outputs and needs no bus stopped, so it is safe on a running line.

The result table has one row per device:

Column	Content
Station	The station name, <i>no name</i> for a device fresh from the box, and a <i>controller</i> tag for another IO controller on the network
Type	The device type string, and its MAC address
Address	IP address, <i>not set</i> , or the address with conflict when DCP reports an address conflict
Device	Vendor id and device id, for example 011e:8032
GSDML	The order number or name from the filed GSDML, or <i>not filed</i>
Buttons	Signal, Set..., Read cards, Add

If you are logged in, the IDE reads the fitted cards of every addressed device straight away and lists them under the device, one line per slot, named from the GSDML.

If nothing answers: check the cable and the device's power. A device on another VLAN or behind a router is not seen.

10.4.1 Signal

Signal makes the device flash its LEDs for about 3 seconds. Use it to find which physical coupler a row is before you rename it.

10.4.2 Read cards

Read cards asks the device which modules are physically plugged, with an implicit RPC read (record `0xF000`, `RealIdentificationData`). The result is the real rack, slot by slot, named from the GSDML; a module the GSDML does not describe is shown as its ident with (*not in GSDML*).

The button is disabled until the device has an IP address, because the read is a UDP RPC. This is the Profinet equivalent of an EtherCAT bus scan: DCP alone finds only the coupler.

10.5 Naming and addressing a device (commissioning)

A new device has no station name and no IP address. It must have both before the controller can connect to it.

To name and address a device:

1. In the scan result, click **Signal** to confirm which device the row is.
2. Click **Set....**
3. Enter the **Station name**. Rules: lower-case letters, digits and `-`, in labels separated by `.`, up to 240 characters. Not an IP address, and not starting with `port-`. Example: `rio-line1`.
4. Enter the **IP address** on the controller's fieldbus subnet, the **Subnet mask** (default `255.255.255.0`) and the **Gateway** (default `0.0.0.0`, none). Leave the IP empty to set only the name.
5. Click **Apply**. The IDE re-scans so the list shows the new name and address.

The name and the address are written to the device **permanently**: they survive a power cycle.

Note: A device refuses DCP Set while it has an active connection to a controller. Stop or disable the controller's Profinet device first (or the other controller that owns the coupler).

10.5.1 Automatic IP assignment at connect

When the device's configuration contains an **IP address** (key `ip`), the runtime checks the device's address every time it connects. If the device is at another address, or has none, the runtime sets the configured address by DCP before connecting and logs *DCP Set-IP 10.10.0.200 -> station 'rio-line1' (temporary)*. This brings back a coupler that reverted to a factory address after a power cycle, without a manual step.

The automatic setting is **temporary** by default: the device keeps it until its next power cycle, and the runtime sets it again then. Set `"set_ip_permanent": true` in `controller.json` to write it permanently. The mask and gateway come from `subnet_mask` (or `mask`) and `gateway`.

If the configuration has no IP, the runtime uses whatever address the device reports.

10.6 Adding the device and its rack

To add a discovered device:

1. Make sure the device has a station name and an IP, and that its GSDML is filed (the **GSDML** column is not *not filed*).
2. Click **Add** on its row.

The IDE reads the fitted cards from the device, lays out the rack from the GSDML, and creates a device named `pn0` (or the next free `pnN`), **enabled**, with:

Key	Content
<code>ifname</code>	The scan's interface
<code>station_name</code> , <code>mac</code> , <code>ip</code>	From the scan
<code>vendor_id</code> , <code>device_id</code> , <code>gsdml</code>	Identity and the GSDML file used
<code>send_clock</code>	The first send clock the GSDML supports (usually 32 = 1 ms)
<code>reduction_ratio</code>	4
<code>watchdog_factor</code>	40
<code>modules</code>	The fitted card idents, slot 1 upward
<code>expected</code>	The expected submodule list for the Connect: the DAP and every fitted card, with their data sizes and parameter records
<code>override_input_bytes</code> , <code>override_output_bytes</code>	The process image size
<code>blocks</code>	One addressed block per card and direction

The **Messages** panel confirms the size, for example *Added Profinet device "pn0" (CN-8032, station "rio-line1") on eth0 with %I 36 B / %Q 24 B. Read 5 fitted card(s) from the coupler.*

If the cards could not be read (no IP, unreachable), the rack starts empty: fit the cards with **Edit rack...** (Section 10.6.2).

WARNING: The device is added **enabled**. After **Download hardware** it exchanges data and drives the coupler's outputs from `%Q`. If outputs must stay dark during commissioning, clear **Enabled** or tick **Inhibited** on the device page before the download.

10.6.1 The device page

Field	Key	Meaning
Interface	<code>ifname</code>	The controller port the device is on
Station name	<code>station_name</code>	The name the runtime connects to. Must match the device exactly
IP address	<code>ip</code>	The address the runtime assigns and connects to (Section 10.5.1)
Vendor / Device	<code>vendor_id</code> , <code>device_id</code>	Read only
Send clock (×31.25µs)	<code>send_clock</code>	32 = 1 ms
Reduction ratio	<code>reduction_ratio</code>	Cycle = send clock × reduction
Watchdog (cycles)	<code>watchdog_factor</code>	See Section 10.7
On stop — outputs, On fault — outputs		Chapter 7, Section 7.10
Missing card	<code>module_policy</code>	Chapter 7, Section 7.4.6

Field	Key	Meaning
Enabled, Inhibited, Critical — stop the controller if it fails, Wait for connection at start (s)	startup_timeout_ms (the wait)	Chapter 7, Section 7.4

The note under the form shows the resulting cycle, for example $\approx 4.0\text{ ms}$. **Apply** keeps everything the scan and the GSDML built (station, expected list, blocks) and only updates the fields on the form.

10.6.2 Editing the rack

Edit rack... on the device page opens the same rack editor as for EtherCAT (Chapter 9, Section 9.6.5), with the card list taken from the filed GSDML files. Add, move and remove cards, then **Save**. The IDE recomputes the expected submodule list, the process image size and the blocks; aliases follow their cards.

A Profinet frame is laid out slot by slot, so every card must be described: a card that is not in any filed GSDML is refused (*N card(s) are not in the filed GSDML...*) rather than guessed.

Important: The rack in the configuration must be the rack fitted. Profinet reports a mismatch at connect (a ModuleDiffBlock), and the runtime then applies the **Missing card** policy. Re-read the cards after any hardware change.

10.6.3 Addresses

Blocks are laid out exactly as for EtherCAT (Chapter 9, Section 9.6.6): in slot order, bit offsets accumulating across the rack, one block per card and direction, with the channel width from the GSDML. The runtime builds its cyclic frame from the same rule, so the addresses the IDE shows are the addresses in the frame.

For the verified ODOT rack (CT-121F, 2 × CT-3238, CT-4234, CT-4158 on a CN-8032):

Slot	Card	%I	%Q
1	CT-121F, 16 DI	%IX0.0 ... %IX1.7	—
2	CT-3238, 8 AI	%IW2 ... %IW16	—
3	CT-3238, 8 AI	%IW18 ... %IW32	—
4	CT-4234, 4 AO	%IB34 (diagnostics)	%QW0 ... %QW6
5	CT-4158	%IB35 (diagnostics)	%QW8 ... %QW22

This is the same layout as the same cards on the EtherCAT CN-8033.

10.6.4 Byte order

The driver copies each submodule's data between the frame and the process image unchanged. With the ODOT C-series cards, 16-bit channels read and write correctly as %IW / %QW: verified on the bench. Profinet's own convention is most significant byte first, and some other devices present their data that way. If a 16-bit value from a device reads as if its bytes were swapped (256 reads as 1), swap it in the program:

```

(* For a device whose 16-bit data arrives most significant byte first. *)
FUNCTION SWAP16 : INT
VAR_INPUT
    w : WORD;
END_VAR
VAR
    v : DINT;
END_VAR
v := SHL(w AND 16#FF, 8) OR SHR(w, 8);
IF v > 32767 THEN v := v - 65536; END_IF;
SWAP16 := v;
END_FUNCTION

PROGRAM PnRack
VAR
    raw_ai    AT %IW2 : WORD;
    ai_swapped : INT;
END_VAR
    ai_swapped := SWAP16(raw_ai);
END_PROGRAM

```

The IO-Link blocks of Chapter 12 do this for IO-Link data, which is always most significant byte first.

10.7 Cyclic exchange and timing

10.7.1 The cycle

The cyclic period is:

```

cycle = send_clock × reduction_ratio × 31.25 μs
       = 32 × 4 × 31.25 μs = 4 ms    (the defaults)

```

The driver runs in its own thread at this period, independently of the PLC scan (async only; sync is refused). Choose a send clock the GSDML lists, and a reduction ratio that makes the cycle no faster than you need: for plain IO, a cycle equal to the PLC scan or half of it is enough. 4 ms has been verified stable on the CM4 controller.

10.7.2 The watchdog

The watchdog factor is handed to the device as both its watchdog and its data-hold factor. It is also the controller's own input watchdog. Because the controller does not run a hard real-time Profinet stack, a tight watchdog trips on ordinary scheduling delays and makes the connection flap. So:

- The runtime **floors** the watchdog at about **150 ms**, whatever the configuration says: the factor is raised to at least $\text{ceil}(150\,000\ \mu\text{s} / \text{cycle})$.
- The IDE's default factor is 40, about 160 ms at a 4 ms cycle.

The controller then applies two levels when input frames stop:

Gap in input frames	Driver state	Action
Longer than the watchdog	DEGRADED	Keeps the connection; the next frame clears it

Gap in input frames	Driver state	Action
Longer than $\max(8 \times \text{watchdog}, 8 \text{ s})$	FAULT	Logs <i>no input frame for a full data-hold time (N ms) - reconnecting</i> , releases the connection and reconnects

After a Connect, the device has a start-up allowance before its first frame: the activity timeout (`activity_timeout_100ms`, default 100 = 10 s, minimum 3 s). If no frame arrives in that time the runtime logs *the device never reached data exchange* and reconnects. Between connection attempts it waits `recovery_ms` (default 1000 ms).

While `DEGRADED` or reconnecting, the device's inputs follow Chapter 7, Section 7.10.2, and the device itself applies its own substitute output values when its watchdog expires.

10.7.3 Configuration keys

Key	Default	Meaning
<code>ifname</code>	<code>eth0</code>	Port
<code>station_name</code>	— (required)	Name to connect to
<code>ip</code> , <code>subnet_mask (mask)</code> , <code>gateway</code>	—	Address to assign and connect to
<code>set_ip_permanent</code>	<code>false</code>	Automatic DCP Set-IP writes permanently
<code>mac</code>	—	Optional; otherwise found by station name
<code>vendor_id</code> , <code>device_id</code>	—	Identity
<code>send_clock</code>	32	$\times 31.25 \mu\text{s}$
<code>reduction_ratio</code>	4	
<code>watchdog_factor</code>	3 in the runtime, 40 from the IDE	Floored to about 150 ms
<code>data_hold_factor</code>	<code>= watchdog_factor</code>	
<code>activity_timeout_100ms</code>	100	Start-up allowance, $\times 100 \text{ ms}$
<code>recovery_ms</code>	1000	Delay between connection attempts
<code>expected</code>	— (required)	Expected submodules, from the GSDML
<code>override_input_bytes</code> , <code>override_output_bytes</code>	from <code>expected</code>	If they disagree with <code>expected</code> , the runtime warns and uses <code>expected</code>

10.8 Bring-up procedure

1. **File the GSDML** in the project's `esi` folder.
2. **Cable** the device to the controller's fieldbus port and power it, including the field supply of the rack.
3. **Scan** (Section 10.4). Use **Signal** to identify the device.
4. **Name and address** it with **Set...** (Section 10.5). Use an address on the controller's fieldbus subnet.
5. **Read cards** and compare the list with the rack in the cabinet.

6. **Add** the device. Before downloading, decide whether outputs may be driven: if not, clear **Enabled** or tick **Inhibited**.
7. **Download hardware**. In **Online** ▾ → **Controller log...**, expect:

```
profinet 'pn0': eth0, station 'rio-line1', 9 submodule(s), %I 36 B / %Q 24 B,
  cycle 4000 us (send_clock 32 x reduction 4), watchdog 40
profinet 'pn0': started on eth0, connecting to 'rio-line1'
pn-cm: rack parameterised via MultipleWrite(0xE040)
profinet 'pn0': connected, input frame 0x8001, output frame 0x8000
```

8. **Check the state** on the device page: **OPERATIONAL**, and **Connected for** growing with no reconnects. On an ODOT C-series coupler the red BF LED goes off when the connection is established.
9. **Check data**: operate an input, force an output (outputs disconnected), check an analogue channel and its signal range.
10. Watch the reconnect count for a few minutes. It must stay at zero.

10.8.1 LEDs on an ODOT C-series Profinet coupler

From the ODOT C-series manual:

LED	State	Meaning
BF (red)	On	Both ports link-down
BF (red)	Flashing 2.5 Hz	Offline: no established connection with a controller
BF (red)	Off	Online
BF (red)	Flashing 10 Hz	MAC address error
SF (red)	On	System failure or topology error
IER	Double flash	IO communication failure on the backplane

A BF LED that keeps flashing while the IDE shows the device connecting and dropping is a connection that is not settling: see the reconnect count and Section 10.9.

10.8.2 Using another controller on the same coupler

A coupler accepts one controller connection at a time. While another controller holds it, this controller's Connect is refused, and DCP Set is refused too. Disconnect the other controller first.

10.8.3 Devices other than the ODOT CN-8032

The parameter record set written at connect is fixed in this release (Section 10.1). With another device or another rack, the log may show:

```
pn-cm: MultipleWrite(0xE040) refused: PNIOStatus 0x... - continuing to PrmEnd
```

The runtime then continues without parameterising. Some devices run on their power-up defaults; others abort the connection a few seconds later, which shows as a reconnect every few seconds. In that case the device cannot yet be run reliably by this controller; report the device and the log.

10.9 Troubleshooting

Log message or symptom	Cause	Remedy
<i>no "station_name" - a Profinet controller connects by station name</i>	Configuration without a name	Enter Station name
<i>no "expected" submodule list - the device was added without a GSDML rack</i>	Device not added through the scan	Open Edit rack... and fit the cards
<i>no device answers to station name '...' on eth0 - give the coupler that name</i>	Name differs, device off, wrong port	Scan; compare the name exactly; use Set...
<i>DCP Set-IP to ... failed: ...</i>	The device refuses (it has an active connection), or the port is wrong	Release the other controller; check Interface
<i>the device has no IP set - a Profinet connection is over UDP and needs one</i>	No IP on the device and none configured	Set one with Set... , or enter IP address
<i>no RPC response within N ms</i>	The device's IP is not reachable from the controller's port	Put the device on the controller's fieldbus subnet
<i>cannot bind UDP 34964 (another Profinet controller running?)</i>	A second Profinet device in the project	One Profinet device per controller (Section 10.1)
<i>device rejected Connect, PNIOStatus 0x... (ErrorCode ... Code1 ... Code2 ...)</i>	The device refuses the connection parameters: often a rack or timing it does not support	Re-read the cards; use a send clock from the GSDML; check vendor/device id
<i>device rejected PrmEnd, PNIOStatus 0x...</i>	Parameterisation refused	See Section 10.8.3
<i>the device's rack differs from the configured one: ... - this device reports DEGRADED...</i>	A card is missing or different	Fix the rack or Edit rack... ; see Missing card
<i>the device never reached data exchange</i>	The device accepted the Connect but produced no data	Check the log before it; power-cycle the device
Reconnect count rising, DEGRADED flicker	Scheduling delays on the controller or a watchdog too tight	Keep Watchdog (cycles) at the default; reduce CPU load; use a real-time kernel
<i>config override_input/output_bytes ... differ from the submodule plan</i>	Hand-edited configuration	Re-add or Edit rack...
Analogue values look byte-swapped	The device sends MSB first	Section 10.6.4

The monitor commands behind the scan (`scan_profinet` , `profinet_set` , `profinet_read_modules`) require the **engineer** role.

11. Addresses, aliases and signal ranges

This chapter describes the process image and how a program reaches it: direct addresses (`%IX0.0` , `%IW2` , `%MD100`), located variables (`AT`), channel aliases that the IDE binds to addresses at build time, per-channel signal ranges that turn raw analogue counts into engineering units, the inhibit and critical flags as they affect addressing, the clock memory bits in `%M` , and byte order. Read it before you write the first line of IO code.

11.1 The process image

The controller keeps three memory areas that a program can address directly:

Area	Prefix	Written by	Read by	Size
Inputs	<code>%I</code>	The IO drivers, at the start of every scan	The program	Sum of the enabled devices' input slices (Chapter 7, Section 7.8)
Outputs	<code>%Q</code>	The program	The IO drivers, at the end of every scan	Sum of the devices' output slices
Markers	<code>%M</code>	The program (and the clock bits, Section 11.7)	The program, servers	%M marker bytes in Controller Settings , default 4096

Each scan does, in order: read inputs into `%I` , apply input forces, update the clock bits, execute the program, apply output forces, write `%Q` to the drivers. Every POU in one scan therefore sees the same `%I` , even if a driver receives new data in the middle of the scan.

`%M` is never touched by a driver. It is retained across restarts (Chapter 33), and it is the area the alarms, the batch context and the servers usually point at.

The sizes of `%I` and `%Q` are shown on the controller node in the device tree, for example `%I 36 B %Q 24 B` .

11.2 Direct addresses

A direct address names a location by area, size and byte offset:

```
% <area> <size> <byte> [ . <bit> ]
```

Size letter	Size	Example	Meaning	Natural type
<code>X</code> (or none)	bit	<code>%IX0.3</code> , <code>%I0.3</code>	Input byte 0, bit 3	<code>BOOL</code>
<code>B</code>	byte	<code>%IB10</code>	Input byte 10	<code>BYTE</code> , <code>SINT</code> , <code>USINT</code>
<code>W</code>	word, 2 bytes	<code>%IW2</code>	Input bytes 2–3	<code>INT</code> , <code>UINT</code> , <code>WORD</code>
<code>D</code>	double word, 4 bytes	<code>%QD8</code>	Output bytes 8–11	<code>DINT</code> , <code>UDINT</code> , <code>DWORD</code> , <code>REAL</code>
<code>L</code>	long word, 8 bytes	<code>%ML16</code>	Marker bytes 16–23	<code>LINT</code> , <code>LWORD</code> , <code>LREAL</code>

Rules:

- The number is always a **byte** offset, from 0. `%IW2` and `%IW3` overlap: word addresses of consecutive 16-bit channels go up in steps of 2 (`%IW2` , `%IW4` , `%IW6` ...).
- A bit number is 0 to 7. A bit address without a size letter (`%I0.1`) is a `BOOL` .
- The area letter is `I` , `Q` or `M` . Anything else is a compile error: 'x' is not a valid memory area - use `I` , `Q` or `M` .
- Multi-byte values are little-endian: `%IW2` has its low byte at `%IB2` .

11.2.1 Located variables (AT)

Give an address a name by declaring a located variable. `AT` is allowed in a POU's `VAR` block and in `VAR_GLOBAL` :

```
PROGRAM Main
VAR
    start_pb    AT %IX0.0 : BOOL;
    stop_pb     AT %I0.1  : BOOL;
    level_raw   AT %IW2    : INT;
    status      AT %IB10   : BYTE;
    total       AT %ID12   : DINT;
    flow        AT %ID16   : REAL;
    pump        AT %QX0.0  : BOOL;
    speed_sp    AT %QW2    : INT;
    blink_1hz   AT %MX100.5 : BOOL;
    word_m      AT %MW200  : WORD;
    running : BOOL;
END_VAR
    running := (start_pb OR running) AND NOT stop_pb;
    pump := running AND blink_1hz;
    speed_sp := level_raw / 2;
    IF %IX1.7 THEN %QX0.1 := TRUE; END_IF;
    word_m.3 := running;
END_PROGRAM
```

The example also shows that a direct address can be used in an expression or as the target of an assignment without a declaration (`%IX1.7` , `%QX0.1`), and that a single bit of an integer can be read or written with integer bit access (`word_m.3` , Chapter 14).

One address per declaration: `a, b AT %IW2 : INT;` is refused (*a direct address cannot be shared by 2 variables*).

When the program is loaded, every located variable is checked against the process image the controller actually has. An address beyond its end refuses the download with a message that names the variable and the size needed, for example 'level' is at `%I40`, which needs `%I` to be at least 42 bytes, but it is 36 - add the IO device this address belongs to, or drop the `AT` clause.

Tip: Declare an IO point in the POU that owns it. A `VAR_GLOBAL ... AT %Q...` that two programs both write is a machine with two masters.

11.2.2 Writing inputs

The program can assign to an `%I` address, but the next scan overwrites it with the driver's data. To hold an input at a value for testing, **force** it (Chapter 22). Only bits can be forced; a word output can be written from the online view but not pinned against the program.

11.3 Channel aliases

An alias is a name given to one channel on the device page. The program uses the name only. The IDE turns the alias into an `AT` declaration **at every build**, from the device tree as it is at that moment. You never paste an address into the program.

This is why aliases exist: when a card moves in the rack, its addresses change. With pasted `AT` declarations, every one after the change is silently wrong. With aliases, the alias follows the card (the rack editor carries it across, Chapter 9, Section 9.6.5), and the next build puts the new address behind the same name.

11.3.1 Naming a channel

To give a channel an alias:

1. Open the device page. Under **Channel aliases**, each block is listed with its channels (up to 32 per block are shown).
2. Each row shows the channel number, its address, the vendor's channel name (for example *Analog Input Data(CH 0)*), and an input box.
3. Type the name in the box (for example `tank_level`) and press Enter or leave the box. The alias is saved at once (*Alias saved*).

Or click  beside the box to open **Project tags**: the list of names already in use — aliases in this project, globals in `globals.st`, and, while connected, the running program's variables. Filter it, and click a name to bind the channel to it. By default only tags of the channel's type are listed; tick *show every type* to see the rest.

Once a channel has an alias its address is shown faint: the program no longer refers to it. Clear the box to remove the alias.

11.3.2 The type an alias gets

Channel	Address form	Type
1-bit (DI, DO, coil, discrete input)	<code>%IX / %QX</code>	<code>BOOL</code>
8-bit (for example a diagnostics byte)	<code>%IB / %QB</code>	<code>BYTE</code>
16-bit (register, analogue channel)	<code>%IW / %QW</code>	<code>INT</code>
32-bit	<code>%ID / %QD</code>	<code>DINT</code>

11.3.3 How binding works at build time

At every build, **Build All** and download, the IDE links the program in this order: data types, `globals.st`, the IO aliases, other global variable lists, the library, the POU's. For each alias:

- **The name is already declared in `globals.st`** (for example `pump : BOOL;`, because the program existed before the rack): the declaration line is rewritten in place with the channel's address, `pump AT %QX0.0 : BOOL;`. The file on disk is not changed; the rewrite is part of the linked program. The declared type must match the channel's type: a `REAL` bound to a `BOOL` card is an error, not a conversion: *'pump' is declared as REAL but is bound to rack1 / DO-16 slot3 Ch#0 (%QX0.0), whose type is BOOL...*
- **The name is declared nowhere**: it is declared in a generated `VAR_GLOBAL` block named **IO aliases**, one line per alias, with a comment naming the device, block and channel. A compiler message in that block points to the channel.

What the linked program contains, for one alias bound in place and one generated:

```

VAR_GLOBAL
    pump      AT %QX0.0 : BOOL;    (* bound in place by the IDE at link time *)
    tank_hi   AT %IX0.4 : BOOL;
END_VAR
VAR_GLOBAL
    tank_level AT %IW2      : INT;    (* ecat0 / CT-3238_2 Ch#0 *)
END_VAR

PROGRAM Fill
VAR
    demand : BOOL;
END_VAR
    demand := tank_level < 12000;
    pump := demand AND NOT tank_hi;
END_PROGRAM

```

Rules the IDE enforces:

- An alias is unique across all devices. A second channel with the same name is refused when you type it: *alias "x" is already used by rack1 / DI slot1.*
- **Do not declare an alias inside a POU.** The local variable hides the global, and the program reads its own variable while the card is never looked at. The IDE reports it as a warning on the POU line: *'pump' is the alias of ... This local declaration hides it, so the program never reads the channel - delete the line and use the name directly.*
- Aliases of a **disabled** device are not declared.

Important: Aliases are saved to `controller.json` with the device. They reach the controller with the **program** (they are declarations), but the addresses behind them are only right if the controller has the same hardware configuration: after a rack change, use **Download all**.

11.3.4 Browse IO (Ctrl+Space)

Browse IO (Ctrl+Space anywhere except the body pane of the code editor, where Ctrl+Space opens the completion list) lists every configured IO point with its alias, address, type, device and direction. Tick points and click **Insert**, or click one row, and the IDE writes `name AT address : type;` lines at the caret, in the POU or in `globals.st` (**Insert into**). An unaliased point is inserted with a name made from its module.

Prefer aliases to Browse IO: an inserted `AT` line is a pasted address, and does not follow the card.

Note: Browse IO takes bit, byte, word or double word from the block's channel width, as the device tree does, not from the Modbus function code. An EtherCAT or Profinet digital card is listed as `%IX / %QX` bits.

11.3.5 More than one device

Block offsets in `controller.json` are counted from the start of each device's own slice. The runtime stacks the enabled devices end to end in configuration order (a disabled device is skipped) and adds each device's base to its offsets. The IDE adds the same base everywhere it shows or declares an address - the device tree, the device page, the signal-range dialog, **Browse IO**, the alias declarations and the **Add Device** preview - so the address you see is the address the program uses. See Chapter 7, Section 7.8.

Disabling, enabling, resizing or reordering a device moves every enabled device after it. Aliases follow automatically; a pasted `AT` address does not.

11.4 Signal ranges (analogue scaling)

An analogue card puts raw counts on the wire. The range it is wired for changes what those counts mean, not the counts themselves. A **signal range** set on a channel makes the controller convert in the IO layer, so the process image holds engineering units and the program carries no scaling factor.

Range	The program reads / writes	Unit
4-20mA	4000 ... 20000	µA
0-20mA	0 ... 20000	µA
0-10V	0 ... 10000	mV
-10-10V (±10 V)	-10000 ... 10000	mV
raw counts	the card's counts	—

Outputs are converted the other way: the program writes 4000...20000 and the card receives its counts.

11.4.1 Setting ranges

Ranges are set **per channel**. A rack routinely has a 4-20 mA transmitter and a 0-10 V positioner on one card, and a per-card setting would scale one of them wrongly.

To set the signal ranges of a card:

1. Open the device page. In the **Modules — process image mapping** table, the **Signal range** column summarises each analogue card: one range, *mixed*, or *raw counts*.
2. Click the cell (... · *set per channel...*). The dialog *Signal ranges — CT-3238 (slot 2)* lists every analogue channel of the card, inputs and outputs, with its address and alias.
3. Use **Set every channel to** for the common case, then change the channels that differ.
4. Click **Apply**.

The ranges offered are the ones the card's GSDML or ESI declares. If the file declares none, the standard four are offered with a warning.

The setting takes effect at the next **Download hardware**.

Note: The **Signal range** column is available for EtherCAT and Profinet racks (and the simulator standing in for them). For a Modbus device, scale in the program with the `SCALE` block, or add a `scale` entry to the device in `controller.json`.

11.4.2 What is stored

Consecutive channels with the same range are stored as one entry in the device's `scale` list (at most 32 entries per device):

```
"scale": [
  { "area": "I", "offset": 2, "count": 8, "width": 16, "signed": true, "clamp": true,
    "raw": [0, 27700], "eng": [4000, 20000], "range": "4-20mA" }
]
```

Key	Meaning
area , offset , count , width	The channels: first byte (relative to the device, like the block offsets), number of channels, 16 or 32 bits
raw	The card's counts at the two ends of the range
eng	What those ends mean
signed	The card's counts are signed (default true)
clamp	Hold at the ends instead of extrapolating (default true)

The conversion is linear from `raw` to `eng`, rounded to the nearest unit, and is applied after the fail-safe pattern, so a held or cleared channel reaches the program in the same units as a live one.

WARNING: The raw counts come from the **card's manual**, not from the vendor file, which names the ranges but not the count scale. The IDE assumes **0...27700** full scale, the ODOT C-series figure, for every range. For another card, or for a bipolar range where the card delivers signed counts (–27700...27700), correct `raw` in `controller.json`. A wrong raw limit gives a plausible value that is wrong by a constant.

11.4.3 Using a scaled channel

With clamping on, a broken 4-20 mA loop reads exactly 4000. To detect a wire break, set `"clamp": false` on the entry so that under-range values come through, and test against NAMUR limits:

```
PROGRAM FlowLoop
VAR
  (* Channel set to 4-20mA on the device page: the process image
   holds microamps, 4000..20000, not the card's raw counts.
   This range entry has "clamp": false in controller.json, so a
   signal outside 4-20 mA is extrapolated instead of held at the
   end of the range - which is what makes a wire break visible. *)
  ft101_ua  AT %IW4 : INT;
  ft101     : REAL;      (* m3/h, 0..120 *)
  ft101_bad : BOOL;
  (* Output channel set to 4-20mA: the program writes microamps. *)
  fcv101_ua AT %QW2 : INT;
  fcv101_pct : REAL := 35.0;
END_VAR
(* NAMUR NE 43: below 3.8 mA or above 20.5 mA is a signal fault. *)
ft101_bad := ft101_ua < 3800 OR ft101_ua > 20500;
IF NOT ft101_bad THEN
  ft101 := INT_TO_REAL(ft101_ua - 4000) / 16000.0 * 120.0;
END_IF;

fcv101_ua := REAL_TO_INT(4000.0 + fcv101_pct / 100.0 * 16000.0);
END_PROGRAM
```

The `%Q` area holds what the **program** wrote (microamps). The conversion to counts is made in a separate buffer on the way to the driver, so a held setpoint is never converted twice.

11.5 Inhibit and critical

Both flags are described in Chapter 7, Section 7.4. Their effect on addresses:

Flag	Addresses	Inputs
Inhibited	Kept: the device keeps its slice, nothing moves	Fail-safe pattern
Enabled cleared	Removed: every later device's slice moves down	—
Critical	Unchanged	As for any device

Inhibit a device that is not wired yet; do not disable or delete it.

11.6 Marker memory

`%M` is ordinary memory with fixed addresses. Use it for values other systems read by address (alarm bits, the batch context, the Modbus TCP slave), for the clock bits, and for retained data. Program variables without `AT` are allocated by the compiler and do not use `%M`.

11.7 Clock memory bits

The controller can drive a **clock byte** in `%M`: eight bits that toggle at fixed rates, written by the runtime once per scan before the program runs. Every POU in a scan sees the same value, and the phase comes from elapsed time, not from the scan count, so a long scan or a spell in PAUSED does not shift it.

Bit	Period	Frequency
0	100 ms	10 Hz
1	200 ms	5 Hz
2	400 ms	2.5 Hz
3	500 ms	2 Hz
4	800 ms	1.25 Hz
5	1000 ms	1 Hz
6	1600 ms	0.625 Hz
7	2000 ms	0.5 Hz

Each bit is TRUE for the first half of its period. The clock is off by default. Enable it, and add up to 8 extra pulses for other rates, in `controller.json`:

```
"clock": {
  "byte": 100,
  "pulses": [
    { "byte": 101, "bit": 0, "period_ms": 10000 },
    { "byte": 101, "bit": 1, "period_ms": 60000 }
  ]
}
```

Validation at load:

- The byte must be inside %M.
- A bit cannot be faster than the scan can observe: the period must be at least 4 scan periods (two scans per half cycle). The standard byte therefore needs a scan of 25 ms or faster; with a slower scan use `pulses`.
- A pulse may not use the standard clock byte, and two pulses may not use the same bit.

```
PROGRAM Lamps
VAR
  (* "clock": { "byte": 100 } in controller.json: %MB100 is the clock
    byte. Bit 5 is 1 Hz, bit 3 is 2 Hz. *)
  clk_1hz   AT %MX100.5 : BOOL;
  clk_2hz   AT %MX100.3 : BOOL;
  alarm_on  : BOOL;
  alarm_ack : BOOL;
  horn      AT %QX0.6 : BOOL;
  lamp      AT %QX0.7 : BOOL;
END_VAR
(* Unacknowledged: fast flash. Acknowledged: steady. *)
IF alarm_on AND NOT alarm_ack THEN
  lamp := clk_2hz;
  horn := TRUE;
ELSIF alarm_on THEN
  lamp := TRUE;
  horn := FALSE;
ELSE
  lamp := FALSE;
  horn := FALSE;
END_IF;
END_PROGRAM
```

11.8 Byte order

Source	What arrives in the process image	What to do
Modbus TCP register	Converted by the driver to native order	Read %IW directly; combine 32-bit values by word order (Chapter 8, Section 8.5)
EtherCAT	Little-endian, like the image	Read directly
Profinet, ODOT C-series cards	Correct as %IW (verified)	Read directly
Profinet, other devices	May be most significant byte first	Swap in the program (Chapter 10, Section 10.6.4)

Source	What arrives in the process image	What to do
IO-Link process data and ISDU	Always most significant byte first	Use <code>IOL_IN_INT</code> and the other helpers (Chapter 12)

11.9 Summary of address limits

Item	Limit
<code>%M</code>	4096 bytes by default (%M marker bytes)
<code>%I</code> , <code>%Q</code>	Sized from the devices; EtherCAT segment 4096 bytes
Bit number	0–7
Signal range entries	32 per device
Clock pulses	8, plus the standard byte

12. IO-Link

This chapter describes IO-Link through the ODOT IP-series IO-Link masters: how the master and its ports appear in the rack, how port settings are configured on the **IO-Link ports** page, how a program reads and writes sensor process data, and how it reads and writes device parameters and handles events with the `zeus-iolink` library blocks. Read it when a machine uses IO-Link sensors or actuators behind an IP-EC-8IA or IP-PN-8IA.

Important: IO-Link support is built and tested against a simulated master, but **has not yet been validated on a real IP-EC-8IA**. Several details (byte order of the IDs and ISDU index, how the master reports a refused parameter, acceptance of the module list) are taken from the vendor's manual and still need confirming on hardware. Run the bench checklist in Section 12.8 before using IO-Link on a machine.

12.1 How IO-Link reaches a program

The controller has no IO-Link stack of its own. The master owns IO-Link; the controller sees the master through ordinary fieldbus IO:

Data	Path	Zeus part
Process data (the sensor value)	A port module in the rack, mapped to <code>%I / %Q</code> like any card	The rack; <code>IOL_IN_INT</code> and related functions for byte order
Port configuration (mode, device check, IDs, start-up parameters)	Per-port object <code>0x8000 + slot</code> , written at start-up	ESI InitCmds → start-up SDOs (Chapter 9, Section 9.7); the IO-Link ports page
Parameters at runtime (ISDU) and events	The <i>ISDU</i> and <i>IO-Link Events</i> modules, which are cyclic process data	Library blocks <code>IOL_ReadParam</code> , <code>IOL_WriteParam</code> , <code>IOL_Events</code>

Because ISDU and events travel in cyclic data, the library blocks are plain Structured Text over the process image. They work the same on EtherCAT and on Profinet.

Supported hardware: ODOT **IP-EC-8IA** (EtherCAT) and **IP-PN-8IA** (Profinet), 8 Class A ports each, with the ODOT IP-IA hubs. The **IO-Link ports** page exists only for the EtherCAT master (Section 12.9).

12.2 The rack

The IO-Link master is a modular device. Its rack, in slot order:

Slot	Module
1	The master module: IP-IOLM-8A0B, 0A8B or 4A4B
2–9	Ports 1–8. Each takes an <i>Iolink Input/Output N bytes</i> module sized to the device's process data
after the ports	Function modules: <i>ISDU</i> , <i>IO-Link Events/1...15</i> , <i>ComErr</i> , <i>PD Valid</i> , <i>Event Flag</i>

To set up the rack:

1. File the vendor's **full** ESI in the project's `esi` folder. The reduced "NS" ESI has no port parameters and expects the ports to be configured with the vendor's own tool.

2. Scan the segment (**Scan EtherCAT segment...**, Chapter 9) and **Add this segment**, or edit the rack with **Edit rack...**
3. Choose each port's module to match the device's process data length (from its IODD). Add the *ISDU* module if the program will read or write parameters, and an *IO-Link Events/N* module if it will handle events.
4. Note, from the device page, the **byte offsets** of each port module and of the ISDU and Events modules, in `%I` and in `%Q`. The library blocks take these offsets as `InBase` and `OutBase`.

WARNING: Nothing in the library can discover a module's offset. Moving or resizing a port module moves every module after it. After any rack change, re-read the offsets from the device page and update the program.

12.3 The IO-Link ports page

The port parameters are the start-up SDOs the ESI declares for each port module: 76 per port. The page edits those entries in place; there is no second copy. It finds each field by the vendor's own name (*Port Mode*, *ISDU Index(3)...*), not by subindex number.

To configure the ports:

1. Open the EtherCAT device page. Under the card table, click **IO-Link ports (N)...**. The button appears when the rack's start-up SDOs contain port entries.
2. Select a port with the **Port 1 ... Port 8** buttons. The line under them shows the slot and the object, for example *Slot 2 · object 0x8001 · written by the master in PRE_OP at every start*.
3. Set the fields (Table 12-1).
4. Switch ports as needed; each port's values are checked when you leave it.
5. Click **Save**. The message reads *IO-Link port settings saved — Download hardware to apply*.
6. **Download hardware**. The settings apply at the next start of the EtherCAT master.

Table 12-1 — Port fields

Field	Values	Notes
Port mode	Manual (port off) · AutoStart - IO-Link, any device	ESI default is AutoStart
Device check	No device check · Type compatible (Data Storage off) · Backup + Restore · Restore	Anything above <i>No device check</i> needs the Vendor ID and Device ID
Pin 2 (IQ)	Not used · Digital input · Digital output	
Cycle time	0 to 132.8 ms	0 = the fastest the device allows. Rounded to the IO-Link time-base steps (0.1 ms up to 6.3 ms, 0.4 ms up to 31.6 ms, 1.6 ms up to 132.8 ms)
Vendor ID	0 to 0xFFFF	From the device's IODD
Device ID	0 to 0xFFFFFFFF	From the device's IODD
Start-up ISDU writes 1–16	Index (1 to 0xFFFF), Sub (0 to 255), Data (hex, up to 32 bytes, MSB first)	Sent to the device every time the port comes up

The start-up ISDU writes are how a replaced sensor gets its settings back: the master writes them each time the port connects. Leave a row empty to leave it unused; the heading shows how many are used.

With **Backup + Restore** or **Restore**, the master's own Data Storage keeps a copy of the device's parameters and writes them to a replacement device of the same type.

When the rack is edited, the port settings follow their port module to its new slot (Chapter 9, Section 9.6.5). A port module added new starts with the ESI defaults.

Note: The page follows the ODOT manual: Vendor ID, Device ID and the ISDU index are sent high byte first. This has not been confirmed on a real IP-EC-8IA. Check one port with a known device (bench item B2) before relying on device validation on a line.

12.4 The zeus-iolink library

The blocks are in `stdlib/70_iolink.st`, library **zeus-iolink** version 1.0.0.

Block	Kind	Purpose
<code>IOL_IN_INT(off)</code>	FUNCTION : INT	16-bit signed process data, MSB first, from <code>%I</code> byte <code>off</code>
<code>IOL_IN_UINT(off)</code>	FUNCTION : DINT	16-bit unsigned, MSB first
<code>IOL_IN_DINT(off)</code>	FUNCTION : DINT	32-bit signed, MSB first
<code>IOL_IN_REAL(off)</code>	FUNCTION : REAL	32-bit IEEE float, MSB first
<code>IOL_OUT_INT(off, val)</code>	FUNCTION : BOOL	Write a 16-bit value MSB first to <code>%Q</code> byte <code>off</code> ; FALSE if outside the image
<code>IOL_ISDU</code>	FUNCTION_BLOCK	The ISDU engine: one read or write through the ISDU module
<code>IOL_ReadParam</code>	FUNCTION_BLOCK	Read one parameter
<code>IOL_WriteParam</code>	FUNCTION_BLOCK	Write one parameter
<code>IOL_Events</code>	FUNCTION_BLOCK	Decode and acknowledge an Events module

They rely on four built-in functions (Appendix A): `PI_IN_BYTE(off)` and `PI_OUT_BYTE(off)` read a byte of `%I` or `%Q` and return `-1` when the offset is outside the image; `PI_SET_OUT_BYTE(off, v)` writes a byte of `%Q`; `DWORD_AS_REAL(bits)` reinterprets a bit pattern as a `REAL`.

12.4.1 Adding the library to a project

- A new project gets the library automatically.
- An existing project with its own `lib` folder must add it: open **Library versions** and use **Add to this project**. A locked library then shows the file as *added* until it is locked again (Chapter 19).

WARNING: Add the library only when the controller runs a runtime that has the `PI_*` built-in functions. Compiles for a hardware target run on the controller's own compiler, and every POU is compiled together with the library. On an older runtime, **every** compile in the project fails with *unknown function PI_IN_BYTE*. Deploy the current runtime first (Chapter 6).

12.5 Process data

IO-Link sends multi-byte values most significant byte first. The process image is little-endian, so a 16-bit sensor value read as `%IW` comes out byte-swapped. Read IO-Link process data with the helper functions, at the port module's **byte** offset:

```
PROGRAM IolProcessData
VAR
  (* Byte offsets of the port modules, from the device page:
   port 1 "Input 2 bytes"   at %IB2
   port 2 "Input 4 bytes"   at %IB4   (32-bit counter)
   port 3 "Input 4 bytes"   at %IB8   (Float32 temperature)
   port 4 "Output 2 bytes"  at %QB2 *)
  pressure_raw : INT;
  pressure_bar : REAL;
  counts       : DINT;
  temp_c       : REAL;
  lamp_level   : DINT := 3;
  ok           : BOOL;
END_VAR
pressure_raw := IOL_IN_INT(2);           (* MSB first, signed *)
pressure_bar := INT_TO_REAL(pressure_raw) * 0.01; (* 0.01 bar/count per IODD *)
counts       := IOL_IN_DINT(4);
temp_c       := IOL_IN_REAL(8);
ok           := IOL_OUT_INT(2, lamp_level); (* FALSE if outside %Q *)
END_PROGRAM
```

Many IO-Link sensors pack status bits into their process data alongside the value (for example the value in the upper 14 bits and two switching bits in the lowest). The layout is in the device's IODD; extract the fields with `SHR` and `AND`.

12.6 Parameters (ISDU)

12.6.1 The ISDU module

Parameter access goes through the *ISDU* module in the rack. The library drives its handshake; you only need its two offsets. For reference, from the ODOT manual:

Direction	Size	Layout
Out (<code>%Q</code>)	39 bytes	[0] port, [1..2] index high/low, [3] subindex, [4] write length, [5] 1 = write / 0 = read, [6] operation ID, [7..38] data
In (<code>%I</code>)	34 bytes	[0] echoed operation ID, [1] length, [2..33] data

Each request takes a **new** operation ID from the one currently in the module, never 0 and never the value being echoed. The module has one request slot: several block instances may share it, but **only one may be busy at a time**. Sequence them (the examples below do).

12.6.2 The blocks

All three parameter blocks follow the PLCopen execute convention: a rising edge on `Execute` starts one transfer and latches the inputs; `Done` or `Error` stays TRUE while `Execute` is held, or for one scan if `Execute` was already

FALSE. Drop `Execute` for at least one scan before starting the next transfer.

`IOL_ReadParam`

Input	Type	Default	Meaning
<code>Execute</code>	BOOL		Rising edge starts the read
<code>InBase</code> , <code>OutBase</code>	DINT		Byte offsets of the ISDU module in <code>%I</code> and <code>%Q</code>
<code>Port</code>	INT	1	1..8
<code>Index</code>	DINT		ISDU index, 0..65535
<code>SubIndex</code>	INT	0	0..255
<code>Timeout</code>	TIME	T#2s	

Output	Type	Meaning
<code>Done</code> , <code>Busy</code> , <code>Error</code>	BOOL	
<code>ErrorID</code>	INT	Table 12-2
<code>RxLength</code>	INT	Bytes returned, 0..32
<code>RxData</code>	ARRAY[1..32] OF BYTE	The bytes, MSB first
<code>Value</code>	DINT	The first 1, 2 or 4 bytes as a big-endian number

`IOL_WriteParam` has the same inputs plus `Length` (0..32), `Data` (ARRAY[1..32] OF BYTE), `value` (DINT) and `UseValue` (BOOL, default FALSE). Put the bytes in `Data[1..Length]`, MSB first; or, for a plain number, set `Length` to 1, 2 or 4, `UseValue` TRUE and the number in `Value`. Its outputs are `Done` , `Busy` , `Error` , `ErrorID` .

Table 12-2 — *ErrorID*

ErrorID	Meaning
1	Timeout: no reply with this request's operation ID within <code>Timeout</code>
2	A read came back with zero bytes, taken as a refused index or subindex
3	Bad argument: <code>Port</code> not 1..8, <code>Length</code> not 0..32, <code>Index</code> or <code>SubIndex</code> out of range
4	<code>InBase</code> or <code>OutBase</code> puts the module outside the process image

The ISDU reply carries no status byte, so a refusal can only be inferred from a zero-length reply (ErrorID 2) or no reply (ErrorID 1). Which of the two a real master produces for a non-existent index is bench item B4.

12.6.3 Reading identification

Standard IO-Link indices include `0x0010` Vendor Name, `0x0012` Product Name, `0x0015` Serial Number, `0x0017` Firmware Revision and `0x0024` Device Status (one byte: 0 OK ... 4 failure). Device-specific parameters are in the IODD.

```

PROGRAM IoIdentify
VAR
    read_req    : BOOL;           (* set by the operator or a sequence *)
    rd          : IOL_ReadParam;
    name_len    : INT;
    name_buf    : ARRAY[1..32] OF BYTE;
    dev_status  : DINT;           (* 0 OK .. 4 failure *)
    last_error  : INT;
    step        : INT := 0;
    i           : INT;
END_VAR
    (* The ISDU module sits at %IB40 (34 bytes in) and %QB12 (39 bytes
       out) on this rack - read both offsets off the device page. *)
CASE step OF
    0: IF read_req THEN last_error := 0; step := 10; END_IF;

    10: (* Index 0x0012, Product Name: a string of up to 32 bytes *)
        rd(Execute := TRUE, InBase := 40, OutBase := 12,
           Port := 3, Index := 16#0012, SubIndex := 0);
        IF rd.Done THEN
            name_len := rd.RxLength;
            FOR i := 1 TO 32 DO name_buf[i] := rd.RxData[i]; END_FOR;
            step := 20;
        ELSIF rd.Error THEN
            last_error := rd.ErrorID;
            step := 90;
        END_IF;

    20: (* Drop Execute for one scan so the next request is a new edge *)
        rd(Execute := FALSE, InBase := 40, OutBase := 12);
        step := 30;

    30: (* Index 0x0024, Device Status: one byte, returned in Value *)
        rd(Execute := TRUE, InBase := 40, OutBase := 12,
           Port := 3, Index := 16#0024, SubIndex := 0);
        IF rd.Done THEN
            dev_status := rd.Value;
            step := 40;
        ELSIF rd.Error THEN
            last_error := rd.ErrorID;
            step := 90;
        END_IF;

    40, 90:
        (* 90 is the error exit: last_error holds 1 timeout, 2 refused,
           3 bad argument, 4 bad base offset. *)
        rd(Execute := FALSE, InBase := 40, OutBase := 12);
        IF NOT read_req THEN step := 0; END_IF;
END_CASE;
END_PROGRAM

```

12.6.4 Writing and verifying a parameter

Write, release the module, then read back:

```

PROGRAM IolSetpoint
VAR
    write_req : BOOL;
    wr        : IOL_WriteParam;
    rd        : IOL_ReadParam;
    sp1_mbar  : DINT := 2500;      (* switch point 1 of a pressure sensor *)
    verified  : BOOL;
    failed    : BOOL;
    err_id    : INT;
    step      : INT := 0;
END_VAR
(* Index 0x003C / sub 1 is the switch point on many IO-Link pressure
   sensors - take the real index, subindex and length from the IODD. *)
CASE step OF
0: IF write_req THEN verified := FALSE; failed := FALSE; step := 10; END_IF;

10: wr(Execute := TRUE, InBase := 40, OutBase := 12, Port := 2,
      Index := 16#003C, SubIndex := 1,
      Length := 2, Value := sp1_mbar, UseValue := TRUE);
   IF wr.Done THEN step := 20;
   ELSIF wr.Error THEN err_id := wr.ErrorID; step := 90;
   END_IF;

20: (* Release the module before the read-back uses it *)
   wr(Execute := FALSE, InBase := 40, OutBase := 12);
   step := 30;

30: rd(Execute := TRUE, InBase := 40, OutBase := 12, Port := 2,
      Index := 16#003C, SubIndex := 1);
   IF rd.Done THEN
       verified := rd.Value = sp1_mbar;
       failed := NOT verified;
       step := 40;
   ELSIF rd.Error THEN err_id := rd.ErrorID; step := 90;
   END_IF;

40: rd(Execute := FALSE, InBase := 40, OutBase := 12);
   IF NOT write_req THEN step := 0; END_IF;

90: failed := TRUE;
   wr(Execute := FALSE, InBase := 40, OutBase := 12);
   rd(Execute := FALSE, InBase := 40, OutBase := 12);
   IF NOT write_req THEN step := 0; END_IF;
END_CASE;
END_PROGRAM

```

Tip: A parameter written at runtime is lost when the sensor is replaced. For settings the machine depends on, also enter them as **Start-up ISDU writes** on the **IO-Link ports** page, or use a Data Storage device check.

12.7 Events

An *IO-Link Events/N* module holds up to N events (1, 2, 4, 6, 10 or 15). Each event is 4 bytes: byte 0 has the type in the high nibble (1 message, 2 warning, 3 error) and the source port in the low nibble, byte 1 is reserved, bytes 2–3 are the 16-bit event code, MSB first. A slot with code 0 is empty.

The module's output holds one "clear" bit per event and, at bit N, a toggle that must **change** before the master acts on the clear bits. `IOL_Events` handles this: a rising edge on `Ack` sets the clear bit of every event present and flips the toggle, so repeated acknowledgements keep working.

`IOL_Events`

Input	Type	Default	Meaning
<code>InBase</code> , <code>OutBase</code>	DINT		Byte offsets of the Events module
<code>Count</code>	INT	1	N of the module: 1, 2, 4, 6, 10 or 15
<code>Ack</code>	BOOL		Rising edge acknowledges every event shown

Output	Type	Meaning
<code>Active</code>	INT	Number of slots holding an event
<code>AnyError</code> , <code>AnyWarn</code>	BOOL	At least one error / warning
<code>EvType</code> , <code>EvPort</code>	ARRAY[1..15] OF INT	Per slot
<code>EvCode</code>	ARRAY[1..15] OF DINT	Per slot
<code>FirstCode</code> , <code>FirstPort</code> , <code>FirstType</code>		The first occupied slot
<code>Fault</code>	BOOL	The module offsets are outside the image

Table 12-3 — Master event codes

Code	Meaning
<code>16#1800</code>	Port communication lost
<code>16#1805</code>	Over-temperature
<code>16#1807</code>	Port power overcurrent
<code>16#1813</code>	CQ overcurrent
<code>16#180A</code> / <code>16#180B</code>	US under / over voltage
<code>16#180E</code> / <code>16#180F</code>	UA under / over voltage

Other codes come from the device; see its IODD.


```

PROGRAM IoDiagnostics
VAR
    ev      : IOL_Events;
    ack_pb  : BOOL;          (* HMI acknowledge button *)
    port_lost : ARRAY[1..8] OF BOOL;
    overcurrent : BOOL;
    supply_bad : BOOL;
    any_error : BOOL;
    i        : INT;
    p        : INT;
END_VAR
(* An "IO-Link Events/2" module at %IB74 (8 bytes) / %QB51 (1 byte) *)
ev(InBase := 74, OutBase := 51, Count := 2, Ack := ack_pb);

FOR i := 1 TO 8 DO port_lost[i] := FALSE; END_FOR;
overcurrent := FALSE;
supply_bad := FALSE;
FOR i := 1 TO 2 DO
    p := ev.EvPort[i];
    IF ev.EvCode[i] = 16#1800 AND p >= 1 AND p <= 8 THEN
        port_lost[p] := TRUE;
    END_IF;
    IF ev.EvCode[i] = 16#1807 OR ev.EvCode[i] = 16#1813 THEN
        overcurrent := TRUE;
    END_IF;
    IF ev.EvCode[i] >= 16#180A AND ev.EvCode[i] <= 16#180F
        AND ev.EvCode[i] <> 16#180C AND ev.EvCode[i] <> 16#180D THEN
        supply_bad := TRUE;
    END_IF;
END_FOR;
any_error := ev.AnyError OR ev.Fault;
END_PROGRAM

```

Map the resulting bits to **%M** alarm bits (Chapter 27) so they are timestamped by the controller and appear in the alarm journal.

12.8 Bench checklist

Run this on a real IP-EC-8IA with one IO-Link sensor before using IO-Link on a machine. Record the results; items B2, B4 and B7 settle details this chapter cannot yet state as fact.

Item	Test	Expected
B1	Start the segment; read the log	The coupler accepts the module list (0xF030), or refuses it with only a warning
B2	Start-up SDOs	Log shows <i>start-up SDOs: N written, 0 refused</i> . Port 1 set to Manual : port LED off, sensor unpowered or silent. Device check 2 with the correct Vendor and Device ID: port comes up; with a wrong Device ID it refuses. This also verifies the byte order of the IDs
B3	IOL_ReadParam of 0x0010 and 0x0012	Vendor and product name in RxData
B4	Read a non-existent index	Record whether it gives ErrorID 2 (zero length) or 1 (no echo)

Item	Test	Expected
B5	<code>IOL_WriteParam</code> of a known parameter, then read back	Same value
B6	Unplug the sensor	Event <code>16#1800</code> on that port and the ComErr bit set; <code>Ack</code> clears the event
B7	Compare <code>IOL_IN_INT</code> with the sensor's own display	Same value (process-data byte order)
B8	Time the start-up with all 633 SDOs	Record it; set the device's Wait for connection at start (s) above it if the device is Critical

12.9 Limits and what is not done

Item	Status
Hardware validation	Not yet run on a real IP-EC-8IA (Section 12.8)
Profinet master IP-PN-8IA	Process data, ISDU and events work unchanged through the library. Port parameters are not configurable from the IDE: they are a record written at connect, and the vendor's GSDML is encrypted, so there is nothing to build it from
IODD import	Not available. Indices, subindices and data layouts are entered from the device manual
Data Storage	Only what device check 2 or 3 does inside the master; no upload/download from the IDE
Concurrent ISDU transfers	One per ISDU module at a time
Parameter size	32 bytes per ISDU transfer

ZEUS PLC IDE · OPERATION MANUAL

Part IV — Programming

13. POU and editors

A Zeus application is made of program organisation units (POUs): programs, function blocks, functions and the methods of function blocks. This chapter explains how to create, organise, rename and delete POU in the IDE, how to read the status marks the POU tree shows, and how to use the Structured Text editor and its editing aids. The language itself is described in Chapter 14; the graphical Ladder and SFC editors in Chapters 16 and 17.

13.1 POU kinds

Kind	Keyword	Keeps state	Tree icon	Use for
Program	<code>PROGRAM</code>	Yes	green P	The top level of the application. A program runs when the task calls it (Chapter 18).
Function block	<code>FUNCTION_BLOCK</code>	Yes, per instance	orange F	Anything with memory that exists more than once: a pump, a valve, a control loop, a sequence.
Function	<code>FUNCTION</code>	No	teal F	A calculation that returns one value.
Method	<code>METHOD</code>	No (works on its block's state)	purple M	An operation of a function block. A method lives inside its block's file.

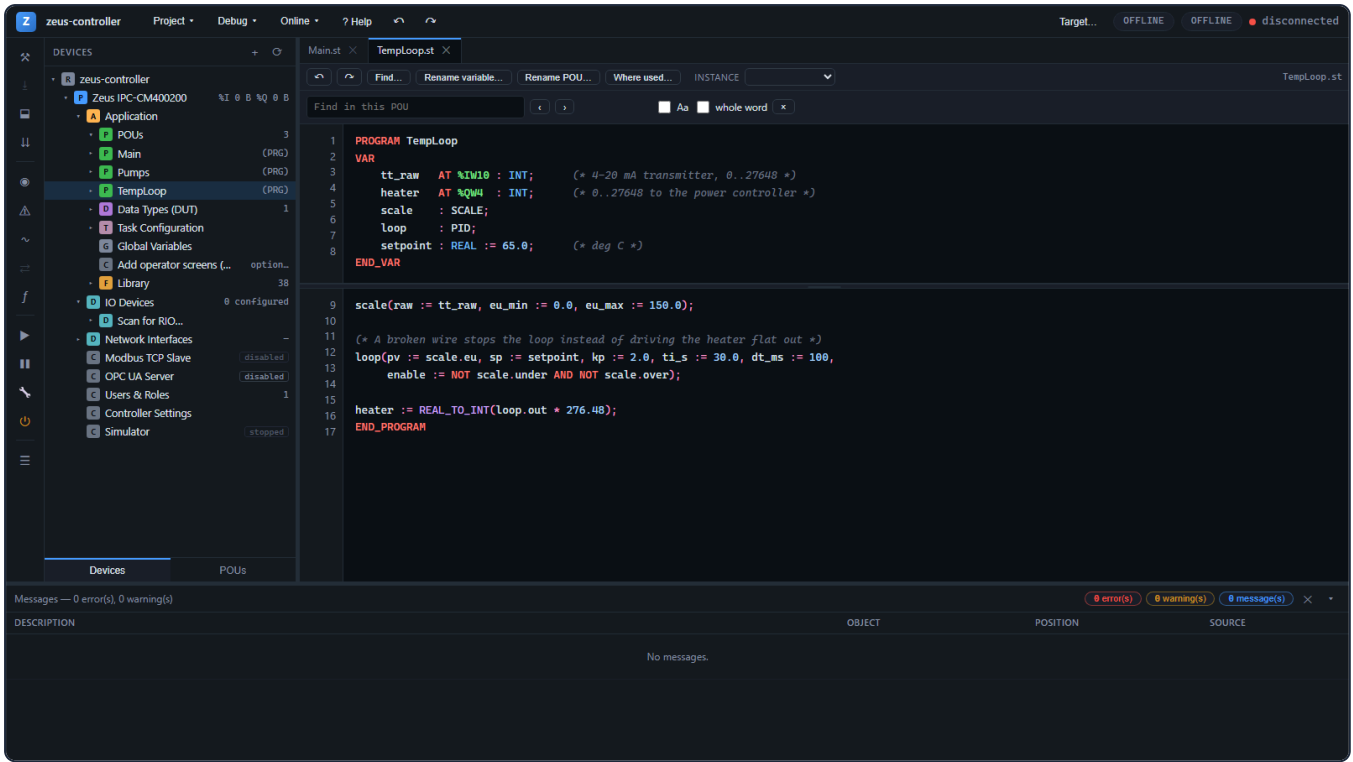
A program can be written in Structured Text, Ladder or SFC, and as an S88 phase (an SFC program with the ISA-88 phase state machine already wired in, Chapter 28). A function block can be written in Structured Text or Ladder. A function is always Structured Text.

Every POU is one `.st` file in the project's `pou/` directory. Ladder POUs are ST files too: the diagram is stored in an encoded header comment and the generated ST is what compiles (Chapter 16). The file is always the source of truth, and it is what the controller receives.

Tip: Choosing a language: Structured Text for calculations, recipes and anything with arithmetic or loops; Ladder for interlocks, permissives and motor starters that an electrician must be able to read; SFC for anything that happens in steps - phases, batches, start-up and shut-down sequences.

13.2 The POU tree

The POUs appear in the project tree under **Application > POUs**. The pane above the tree has two tabs, **Devices** and **POUs**; with **POUs** selected, the **+** button creates a new POU ("New POU (program, function block or function)").



Structured Text editor with the TempLoop program

Each POU node shows:

- the POU's name (its file name without `.st`);
- its kind in brackets - `(PRG)`, `(FB)`, `(FC)` - or `(excl'd)` if it is excluded from Build All (Section 13.5);
- a status dot (Section 13.2.1);
- the badge **RUNNING** on each program the controller is executing.

Under a POU node are its **Variables** (the online variable table, Chapter 22), its **Chart** if it is a sequence (Chapter 17), and one node per METHOD of a function block.

13.2.1 Status marks

When the IDE is connected to a controller and logged in, it compares every POU's saved file with the program the controller is running, and marks each POU with a dot:

Mark	Meaning
green • "Identical to the controller"	The controller runs exactly this POU.
orange • "Differs from the controller — download to update it"	The saved file is different from what the controller runs.
grey • "Not on the controller — download to send it"	The controller does not have this POU at all.
orange • "Changed — unsaved edits are not on the controller"	The POU (or one of its method tabs) has unsaved edits in the editor.
no dot	Not connected, not logged in, or the POU is excluded from Build All.

A folder shows one dot for everything in it - green if all its POU's are identical, orange if all differ, grey if none is on the controller - or a half-green, half-orange dot, "Mixed" - some POU's in this folder differ from the controller, some

match".

The comparison is refreshed when you connect, save or download, and when the running program changes; it is not polled continuously.

When the controller is running, each program it executes is labelled **RUNNING** in green, and the other programs are dimmed. A folder shows **RUNNING** when a program anywhere inside it runs (**N RUNNING** when several do), so a collapsed folder still shows where the running code is. A program that is dimmed is not in the task's call list (Chapter 18) - it compiles, but it never runs.

Note: The dots compare **saved files**, POU by POU. The **IN SYNC** badge at the top of the window answers the question for the whole application, including unsaved edits in the active editor and the hardware configuration (Chapter 20). An unsaved edit shows as an orange dot and as **PROGRAM CHANGED** at the same time.

Editor tabs with unsaved changes show a blue ● after their name. Closing such a tab asks for confirmation ("... has unsaved changes. Close anyway?"), and leaving the page with unsaved tabs gives the browser's warning.

13.3 Creating a POU

To create a POU:

1. Right-click **Application** (or **POUs**, or a folder) and choose **New POU...**, or click + above the tree with the **POUs** tab selected. The **New POU** dialog opens.
2. In **Name**, type the POU name. It becomes both the file name and the name in the POU's header, so use a valid identifier: a letter or underscore, then letters, digits and underscores.
3. In **Type**, choose the kind and language (table below).
4. In **Folder**, choose where to put it, or (**top level**). To make a new folder, type its path in the box under the form ("new folder, e.g. motors or motors/pumps") and click **Create folder**.
5. Click **Create**. The POU is written to disk and opens in its editor.

Type	Creates
Program (PRG) — ST	An empty <code>PROGRAM</code> with one <code>VAR</code> block.
Program — Ladder	A ladder program; opens in the Ladder editor (Chapter 16).
Program — SFC (sequence)	A <code>PROGRAM</code> holding a chart with one <code>INITIAL_STEP Start</code> ; the chart has to have an initial step to compile (Chapter 17).
Program — S88 phase (batch)	A program that calls the library's <code>S88_PHASE</code> state machine and holds a two-step chart that runs while <code>ph.run_logic</code> is TRUE, with the command, parameter and report variables the batch engine uses (Chapter 28).
Function Block (FB) — ST	An empty <code>FUNCTION_BLOCK</code> with <code>VAR_INPUT</code> , <code>VAR_OUTPUT</code> and <code>VAR</code> blocks.
Function Block — Ladder	A ladder function block (Chapter 16).

Type	Creates
Function (FC) — ST	A <code>FUNCTION ... : BOOL</code> with a <code>VAR_INPUT</code> block. Change the return type to what the function returns.

Apart from the SFC and S88 phase templates, a new POU is an empty frame, with no example variables or code to delete.

If the name is already taken, the dialog "*name* already exists" offers **Open the existing POU, Choose another name** or **Replace it**. **Replace it** asks again before overwriting the existing POU with an empty skeleton - everything in it is lost.

The new POU does not run until a program calls it (function blocks and functions) or the task calls it (programs). Add a new program to the task's call list in **Task Configuration** (Chapter 18).

The empty ST templates compile as they are:

```

FUNCTION_BLOCK NewBlock
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR

END_FUNCTION_BLOCK

FUNCTION NewFunction : BOOL
VAR_INPUT
END_VAR

END_FUNCTION

PROGRAM NewProgram
VAR
END_VAR
(* A sequence: steps, transitions between them, and actions that run
   while a step is active. See Help > SFC. *)
INITIAL_STEP Start:
END_STEP
END_PROGRAM

```

13.3.1 Importing POUs

Import ST file... (right-click **Application**, a folder or a POU; or **Project ▾ > Import ST...**) reads POUs exported from another IEC 61131-3 tool, in PLCopen XML or as a text export (`.exp` , `.export`). The dialog lists the POUs found with the issues found in each; constructs that Zeus does not support are commented out and listed as warnings. Choose the POUs and the **Into folder**, then click **Import selected**. Imported code is not verified - run **Build All** before you download it. Chapter 5 describes import in detail.

Note: The importer currently comments out `METHOD` blocks as unsupported. Re-enter imported methods with **Add method...** (Section 13.8).

13.4 Folders

Folders organise the POU tree. A folder is a real directory under the project's `pou/` directory; there is no separate project file listing them, so what you see on disk is the structure.

- **Create a folder:** in the **New POU** dialog (**Create folder**), or right-click a folder or **POUs** and choose **New sub-folder...** Folders may be nested (`motors/pumps`).
- **Move a POU:** drag it onto a folder in the tree. Drop it on **POUs** to move it back to the top level. The drop target is highlighted.
- **Rename or delete a folder:** not possible in the IDE. An empty folder stays in the tree; remove it in the file system if it is not wanted.

Important: A POU's project name is its path: `motors/Pump.st` is a different name from `Pump.st`. Moving a POU is a rename, with the same consequences (Section 13.6): the task call list and the Build All exclusion follow the move, in place, so a moved program keeps its position in the call list. The declaration header is not changed by a move (the base name is the same), so the POU's users need no change.

13.5 The POU context menu

Right-click a POU in the tree:

Item	Does
Open	Opens the POU in its editor. Clicking the POU node does the same.
Open Variables	Opens the POU's online variable table (Chapter 22).
Open Chart	SFC programs only: opens the chart view (Chapter 17).
Add method...	Function blocks only: adds a METHOD (Section 13.8).
Change to FUNCTION... / Change to FUNCTION_BLOCK...	Converts between function block and function (Section 13.6.3).
Save (Ctrl+S)	Saves the project; enabled when the POU has unsaved edits.
Save As... (Ctrl+Shift+S)	Saves a copy under a new name (Section 13.6.4).
New POU here...	New POU dialog with this POU's folder selected.
Import ST file...	Import into this POU's folder.
Build (F7)	Compiles this POU (Section 13.11).
Download to controller (F8)	Downloads the program (Chapter 20).
Upload from controller...	Reads the whole running program back as a read-only snapshot (Chapter 20).
Rename...	Renames the POU (Section 13.6.1).
Disable / Enable	Disables or re-enables the POU (Section 13.6.5).
Delete	Deletes the POU (Section 13.6.2).

13.6 Renaming, deleting, converting and disabling

13.6.1 Renaming a POU

To rename a POU:

1. Right-click it and choose **Rename...**, or click **Rename POU...** in its editor.
2. Type the new name at the prompt "Rename "*name*" to:" and confirm.

The POU stays in its folder. The file is renamed, and if the POU's header (`PROGRAM Old` , `FUNCTION_BLOCK Old` , `FUNCTION Old`) matches the old file name, the header is rewritten too, so the name the controller reports is the new one. If the header did not match, only the file is renamed and a warning in **Messages** says so.

The project configuration follows the rename: the entry in the task's call list is renamed in place (same position, so the execution order is unchanged), an exclusion from Build All moves to the new name, and, when the header was rewritten, the program's rate in **Task Configuration** (`task.programs` , keyed by the PROGRAM name) moves to the new name too.

Important: Rename does **not** update calls of a renamed function and instances of a renamed function block (`p1 : OldName;`) in other POUs - they fail to build with "unknown type". Use **Where used...** (Section 13.10.4) on the old name before renaming a block or function, and fix its users afterwards.

13.6.2 Deleting a POU

Right-click the POU and choose **Delete**; confirm "Delete "*name*"? This cannot be undone." The file is deleted and its tab closed. The POU is also removed from the task call list and from the Build All exclusion list, so no stale entry is left behind. A deleted POU stays on the controller until the next download.

13.6.3 Converting between function block and function

Change to FUNCTION... (on a function block) and **Change to FUNCTION_BLOCK...** (on a function) rewrite the POU's header and tell you in **Messages** what else needs to change.

A function block is converted to a function only if:

- it has no methods ("*name* has METHODS - a FUNCTION cannot have them"), and
- it declares no timer, counter, edge detector or latch (`TON` , `TOF` , `TP` , `CTU` , `CTD` , `CTUD` , `R_TRIG` , `F_TRIG` , `RS` , `SR`): a function keeps no state, and such a block inside a function would be shared by every call.

If the block is used elsewhere, the IDE lists up to five of the places and asks "Convert anyway and fix them yourself?". It then asks for the function's return type (default `BOOL`).

Converting a function to a function block warns you if the function assigns its own name - the return value - because a function block has no return value; use an output instead.

13.6.4 Save As

Save As... (POU context menu, or **Project ▾ > Save As... (POU)**, Ctrl+Shift+S, on a POU tab) writes a copy of the POU under a new **Name** (default `<name>_copy`) in a chosen **Folder**. The copy includes unsaved edits in the editor; the original file is left exactly as it is on disk. If the target exists you are asked before it is overwritten. Remember to rename the header inside the copy if it is a function block or function you intend to use alongside the original - two POUs with the same header name do not compile together.

13.6.5 Disabling a POU

Right-click a POU → **Disable** takes it out of the application without deleting it: the file is untouched, and the setting is stored in the controller configuration. A disabled POU:

- shows dimmed and struck through in the tree, marked (disabled) , with no status dot; its **Build** and **Download to controller** menu entries are greyed out;
- is skipped by **Build All** ("disabled - skipped");
- is left out of every other POU's compile. If it is a function block or function that other POUs use, those POUs now fail with "unknown type" or "unknown function";
- is left out of the download even if it is a program in the task call list, and is not listed under any task on the Task Configuration page;
- is not compared with the controller: a disabled POU that was never downloaded does not keep the IDE from showing **IN SYNC**, and editing it does not change the badge.

Use it to park work in progress or an alternative version of a program. Right-click → **Enable** restores it. To stop a program running while keeping it enabled, take it out of the task instead (Chapter 18: - **not running** - in its **Task** box).

13.7 The Structured Text editor

A Structured Text POU opens in a two-pane editor. The file is one text; the editor shows it in two parts:

- the **declaration pane** (top): the header line and the `VAR ... END_VAR` blocks;
- the **body pane** (bottom): the statements, up to `END_PROGRAM` / `END_FUNCTION_BLOCK` / `END_FUNCTION` .

The split is made after the last `END_VAR` before the first statement; joining the two panes gives back the file exactly. A file with no POU header is shown in a single pane. Drag the divider between the panes to resize them ("Drag to resize"); double-click it to reset the declaration pane to its default height.

In a function block, the `METHOD ... END_METHOD` blocks are hidden from both panes. They are edited in their own tabs (Section 13.8) and put back into the file when it is saved, compiled or downloaded.

13.7.1 The action bar

Above the declaration pane:

Control	Does
	Undo (Ctrl+Z) / Redo (Ctrl+Y) in this editor (Section 13.10.1).
Find...	Find in this POU (Ctrl+F, Section 13.10.2).
Rename variable...	Renames the variable at the caret (F2, Section 13.10.3).
Rename POU...	"Rename this POU, and the name it calls itself" (Section 13.6.1).
Where used...	Cross-reference of the name at the caret (Shift+F12, Section 13.10.4).
Instance	Function blocks only, online: chooses which instance's live values the editor shows (Chapter 22).

There is no Build button in the editor; use **Debug ▾** > **Build**, the  button on the left rail, or F7.

13.7.2 The gutter

Both panes have line numbers in a gutter on the left. The numbers are the **file's** line numbers: the body continues from where the declarations (and any hidden methods) end, so the line in a compiler message is the line you see.

Clicking a line number in the body pane sets or clears a **breakpoint** on that line when you are connected and logged in: a red ● marks it, and an amber ► marks the line where the program is halted. Read Chapter 24 before using breakpoints - a breakpoint stops the program in the middle of a scan while the outputs hold their last values.

13.7.3 Syntax colouring

Colour	Used for
red, bold	Keywords (IF , VAR , END_FOR , ...)
light blue	Elementary types (INT , REAL , ...)
violet	Standard function blocks and built-in functions (TON , ABS , ...)
pale blue	Numbers and strings
orange	Duration literals (T#5s)
green, bold	Direct addresses (%IX0.3)
pink	Operators
grey, italic	Comments, including nested (* *) and //

13.7.4 Compiler messages in the editor

After a build, a line with an error is tinted **red** in the editor, with its line number shown in **bold red** in the gutter; a line with a warning is tinted **amber**, with its number in **bold amber**. The marks appear in both panes and in method tabs. The full text is in the **Messages** panel at the bottom of the window:

- The title reads "Messages — N error(s), N warning(s), N message(s)"; the three chips filter by severity; × clears the list and ▾ collapses the panel.
- Columns: **Description**, **Object** (the file, or file · Method for a method), **Position** ("Line N, Column M") and **Source**.
- Click a row to jump to the line - in the right pane, in a method tab, in globals.st or in a global variable list. A message without a line opens the file at the declaration of the name it quotes.

The editor also shows each message **in the code**: a wavy underline (red for an error, amber for a warning) under the word the compiler points at, and the message itself at the end of the line, for example ◀ 'PT_03_04' is not declared (+1 more) . "(+N more)" means that line has several messages; all of them are in the Messages panel. The underlines appear after **Build** and after **Build All** - for every POU, including those that were not open at the time - and disappear as soon as you edit the code; the gutter marks stay until the next build.

Build All builds the text you see: an open POU with unsaved edits is built from the editor, not from the file, so every line number refers to what is on screen (a note at the top of the list names those POUs; save to keep the edits). Errors that the controller reports on a download - in any program of the task call list, or in a project function block - are placed on their own file and line in the same way.

13.7.5 Completion

While you type a name in the body pane (two letters or more), a completion list opens; Ctrl+Space opens it explicitly. It offers, each marked with its kind:

- local variables and globals (including IO aliases);
- function blocks and functions - the project's, the library's and the standard ones;
- after `instance.`, the members of that instance;
- after a block's type name and a dot (`TON.`), the instances of that type, plus "declare `x_1 : TON`", which adds the declaration to the POU's first `VAR` block.

↑ and ↓ move, Enter or Tab accepts, Esc closes. Completion is not offered in the declaration pane.

Note: Outside the body pane, Ctrl+Space is the shortcut for **Browse IO** (Section 13.10.9). In the body pane (and in a method tab) it opens only the completion list; to open Browse IO from a POU, click in the declaration pane first and press Ctrl+Space there.

13.7.6 Typing aids

Tab inserts four spaces (replacing any selection). There is no automatic indentation, no Shift+Tab outdent and no bracket matching; indent with Tab or with a column selection (Section 13.10.5).

13.7.7 The editor context menu

Right-click in the editor:

- on a name: **Where used...** (Shift+F12), **Find in this POU** (Ctrl+F), **Rename variable...** (F2), and Cut / Copy / Paste;
- elsewhere: **Find in this POU...** and the clipboard commands.

If the browser blocks the clipboard, **Paste** says "Paste needs clipboard permission — press Ctrl+V instead".

13.8 Methods

A METHOD belongs to one function block and runs on its instance (Chapter 14, Section 14.14). In the IDE a method is edited in its own tab, and the function block's tab hides the METHOD blocks, as in other IEC environments.

13.8.1 Adding a method

To add a method:

1. Right-click the function block in the tree and choose **Add method...**
2. Type the method name at the prompt "Method name" (default `Operation`). It must be an identifier and must not already exist in the block.
3. Confirm. The IDE inserts an empty method after the block's existing methods and **saves the function block file immediately** - together with any other unsaved edits in the block's tab - then opens the method's tab.

The skeleton is a bare `METHOD <name>` with no return type or parameters. Add them in the method tab: `METHOD Scaled : REAL`, a `VAR_INPUT` block, local `VAR` s. Access specifiers (`PUBLIC` , `PRIVATE`) may be written and are ignored.

13.8.2 The method tab

The tab is titled `<Block>·<Method>` and headed "METHOD **name** of **Block** — edits are saved back into the block". It has the same editor, gutter, colouring, completion and undo as a POU; `END_METHOD` is hidden and added back on save. A method tab:

- refuses to save text that contains `END_FUNCTION_BLOCK`, a second `METHOD`, or an `END_METHOD` in the middle;
- compiles as part of its function block when you press F7, with messages mapped to the method's own lines;
- downloads its block with F8;
- numbers its own lines; breakpoints are not available in a method tab;
- has no Find bar (Ctrl+F does nothing there).

Each method also appears as an **M** node under its block in the tree. Its context menu offers **Open**, **Rename method...** and **Delete method...**

- **Rename method...** renames the declaration, the calls by bare name (`Name( )`) and the assignments of the return value (`Name :=`) inside the block, and saves the block.
- **Delete method...** asks "Delete method "X" from Block?" and says how many calls will be left behind; it saves the block.

```
FUNCTION_BLOCK Tank
VAR_INPUT
    level : REAL;
    reset : BOOL;
END_VAR
VAR_OUTPUT
    high : BOOL;
    trips : INT;
END_VAR
METHOD Operation
    (* Reads and writes Tank's own
       variables directly - a method runs on the instance. *)
    IF level > 90.0 AND NOT high THEN
        trips := trips + 1;
    END_IF;
END_METHOD
METHOD Margin : REAL
VAR_INPUT
    limit : REAL;
END_VAR
    Margin := limit - level;
END_METHOD
    Operation();
    high := Margin(90.0) < 0.0;
    IF reset THEN
        trips := 0;
    END_IF;
END_FUNCTION_BLOCK
```

13.9 Variables of a POU

Declare variables in the declaration pane. The online **Variables** table (right-click the POU > **Open Variables**, or the **Variables** node under it) lists every variable of the POU with its type, address and - when online - its live value, and

is where values are prepared, written and forced (Chapter 22). Global variables are declared in **Global Variables** and in global variable lists (Chapter 19); data types in **Data Types (DUT)** (Chapter 15).

13.10 Editing aids

13.10.1 Undo and redo

Each pane has its own undo history of up to **500 steps**.

- **Ctrl+Z** undoes; **Ctrl+Y** or **Ctrl+Shift+Z** redoes. The ↶ / ↷ buttons in the action bar and in the top toolbar ("in the editor you are working in") act on the pane you last typed in.
- Typing is grouped into word-sized steps: a new step starts at a space or line break, when you switch between typing and deleting, when you move the caret elsewhere, or after a pause of one second.
- Every change the IDE makes to the text itself - an inserted library block, a completion, a column edit - is one step.
- The history is cleared when the text is reloaded from disk, for example after **Rename variable...** or **Add method...**, which save the file.

13.10.2 Find in this POU

Press **Ctrl+F** (or click **Find...**) to open the find bar over the pane that had the focus. A single-line selection of up to 80 characters is used as the search text.

Control	Does
Aa	"Match upper and lower case exactly"
whole word	"Only whole identifiers - finds the variable x, not every x"
⏪ / Shift+Enter / Shift+F3	Previous match
⏩ / Enter / F3	Next match
✕ / Esc	Close

The counter shows "n of N" or "no matches". The search is literal: there are no regular expressions.

The find bar also replaces: type the new text in **Replace with**, then **Replace** (the selected match, then the next one; Enter in the box does the same) or **Replace all** (every match in this POU - its declarations and its code).

Ctrl+Z undoes a replace. To rename a variable correctly in every POU that uses it - skipping comments and strings - use **Rename variable...** instead.

13.10.2a Find and replace in the whole project

Project → **Find / replace in project...** (**Ctrl+Shift+F**, or **In project...** in the find bar) searches every POU, `globals.st`, the global variable lists and the data types, with the same **Match case** and **Whole word** options.

1. Type the text in **Find** and press **Find all** (or Enter). Every match is listed under its file, with its line; the summary reads "N match(es) in M file(s)".
2. Click a match to open that file at that line.
3. To replace: type **Replace with**, untick any file you want to leave alone, and press **Replace in ticked files**. After a confirmation, open editors with unsaved edits are saved first, the files are changed on disk, and open editors are reloaded. The Messages panel lists what was replaced in each file.

A **ladder** POU is found (in its generated code) but never replaced as text - its diagram is what is stored, and it would overwrite the change; such files show "ladder POU - change it in the ladder editor" and cannot be ticked. A project-wide replace can be undone only by replacing back.

13.10.3 Rename variable (F2)

To rename a variable:

1. Put the caret on the name in the body pane and press **F2**, or click **Rename variable....** With no name at the caret, the dialog "Rename which variable?" lists the POU's variables to choose from.
2. Type the new name and confirm.

The scope follows the declaration:

- A name declared in `globals.st` is renamed in `globals.st` and **every** POU.
- Any other name is renamed in this POU only; for a method, in the whole function block file.

Comments and string literals are left alone. All unsaved tabs are saved first and the open POUs reloaded. The IDE reports the number of occurrences renamed and warns that the rename is in the project only: the controller keeps the old name until you download.

Rename variable refuses:

- a name that is not declared in this POU or in `globals.st` - including names declared only in a global variable list under `gv1/`, a DUT or the library (rename those in their own editors);
- an invalid identifier;
- a new name that differs from the old one only in upper/lower case ("that is the same name - ST identifiers are case-insensitive").

Note: F2 and Shift+F12 act on the caret in the **body** pane. To rename a variable from the declaration pane, place the caret on a use of it in the body, or use **Rename variable...** with no name at the caret and pick it from the list.

13.10.4 Where used (Shift+F12)

Put the caret on a name and press **Shift+F12**, click **Where used...**, or right-click the name. The dialog "Where is *name* used?" searches `globals.st`, every global variable list and every POU (excluded ones included), ignoring comments and strings, and classifies each hit:

Use	Recognised as
declaration	the name followed by <code>:</code> inside a <code>VAR</code> block
write	the name followed by <code>:=</code> , or bound by <code>=></code>
read	everything else

The summary line reads "N declaration(s) · N write(s) · N read(s)"; each row gives the use, the scope (which POU), the file, the line and the code. Click a row to go to it.

If the name is declared more than once - a local with the same name as a global - the dialog says so first. That situation compiles, but the program uses the local while the monitor, the ladder display, writes and forces use the global (Chapter 14, Section 14.7.1). Before changing a value on a running machine, use **Where used** to see everything that reads and writes it.

13.10.5 Column selection (Alt+drag)

Hold **Alt** and drag with the mouse to select a rectangle: the same columns on several lines. Then:

- typing a character replaces the block on every line (with a zero-width block, it inserts on every line - useful for indenting or commenting out a column);
- Backspace and Delete work on every line;
- Ctrl+C / Ctrl+X copy or cut the rectangle;
- Ctrl+V pastes a multi-line clipboard one line per row, or a single line into every row.

Esc, an arrow key, Home, End, Page Up, Page Down or a plain click leaves column mode. A tab counts as one column. Column selection works in both panes and in method tabs.

13.10.6 Hover information

Rest the mouse on a name for about a third of a second to see:

- its name and type;
- what it is: input, output, in/out (by reference), local, temporary, static, global, external, IO alias (a global bound to a channel), a function block's or method's input, a member of a type, a method of this block, or a function block type ("declare an instance to use it");
- `RETAIN`, `PERSISTENT` or `CONSTANT`, its `AT` address and its initial value;
- the comment written beside its declaration;
- where it is declared;
- online, its live value. In a function block, choose an **Instance** first; until then the hover says "choose an INSTANCE to see its value".

Write a comment on the declaration line - it is what the next engineer sees when they hover.

Note: The hover calls `VAR_TEMP` variables "Temporary (reset every call)". That is true in a FUNCTION or METHOD; in a PROGRAM or FUNCTION_BLOCK they keep their value like `VAR` (Chapter 14, Section 14.7.2).

13.10.7 Inserting a library block

The function block library (Chapter 19, Appendix C) holds ready-made blocks such as `SCALE`, `PID` and `MOTOR_DOL`. To use one:

1. Put the caret at the **start of the body** of the POU, before the first statement.
2. Open the library: the **Library** node in the tree, or the **f** button on the left rail. The **Function block library** dialog opens.
3. Choose the **Block**. The dialog shows its summary, its **Inputs** (with defaults) and **Outputs**, and **What gets inserted**.
4. Type an **Instance name** (default `<block>_1`).
5. Click **Insert into active POU** (or **Copy** to put the text on the clipboard).

The IDE inserts a `VAR` block declaring the instance, followed by a call listing every input with its default value. Placed at the start of the body, the `VAR` block joins the declarations; placed after a statement, it does not compile - move the `VAR ... END_VAR` lines up into the declaration pane. Edit the call's arguments, then read the outputs as `instance.output`. **Versions...** shows the versions of the library blocks the project uses.

For example, inserting `SCALE` as `level_scale`, then editing the call, gives a program like this:


```

PROGRAM LevelLoop
VAR
    level_raw  : INT;
    level_pct  : REAL;
    high_alarm : BOOL;
END_VAR
VAR
    level_scale : SCALE;
END_VAR

level_scale(
    raw      := level_raw,
    raw_min  := 0,
    raw_max  := 27648,
    eu_min   := 0.0,
    eu_max   := 100.0,
    clamp    := TRUE);
level_pct := level_scale.eu;
high_alarm := level_pct > 90.0;
END_PROGRAM

```

13.10.8 Instance declarations from completion

Typing a block's type name and a dot in the body (`TON.`) offers "declare x_1 : TON", which adds the declaration to the first `VAR` block for you - a quick way to add a timer without leaving the body.

13.10.9 Browse IO (Ctrl+Space)

Browse IO lists the IO channels of the configured devices with their addresses and aliases. **Insert into** chooses the POU (inserted as `VAR` declarations) or `globals.st` (as `VAR_GLOBAL`); filter with "filter by alias, address or module..." and **named only**, select channels (**Select all shown**, **Clear**) and click **Insert**. Each channel becomes a line such as `name AT %IX0.0 : BOOL;` at the caret; in a POU the caret is in the body pane, so move the lines into a `VAR` block of the declaration pane. In most projects, naming channels as IO aliases (Chapter 11) is simpler than located variables in POUs.

13.11 Build and Check

The IDE has no separate "check" step for POUs: **Build** is the check. It compiles; it never contacts the controller.

Command	Key	Where	Does
Build	F7	Debug ▾, rail ⚙, POU menu	Compiles the active POU as the controller would see it: DUTs, <code>globals.st</code> and the IO aliases, the global variable lists, the library, the project's other enabled function blocks and functions, then this POU's editor text - saved or not .
Build All	Ctrl+F7	Debug ▾	Saves <code>globals.st</code> if needed, then builds every enabled POU in turn. Ends with "Build All: N ok, N failed, N skipped".
Clean	-	Debug ▾	Clears the Messages panel. There is no incremental build cache to clear.
Clean all	-	Debug ▾	Clears the Messages panel and runs Build All.

Command	Key	Where	Does
Check	-	Global Variables , DUT and global- variable-list tabs	Saves that file and compiles it with the rest of the project.

Build reports "Compiling...", the diagnostics, then "Build succeeded." or the error count. Messages refer to the file and line they come from - an error in a DUT or in `globals.st` is reported there, even when you built a POU.

Build compiles one POU in context. The **download** compiles the whole bundle - every program in the task call list (Chapter 20) - so a project in which each POU builds on its own can still fail to download, because the programs of the call list are only compiled together at that point. Run **Build All** before you download, and fix every error: the controller never receives a program that does not compile.

Tip: Warnings do not stop a build, but read them. "shadows the global", "truncates the fractional part" and "mixed integer and real arithmetic" each point at code that does something other than it appears to (Chapter 14, Section 14.24).

13.12 Keyboard reference for editing

Key	Action
Ctrl+S	Save project (every unsaved POU and the globals)
Ctrl+Shift+S	Save As (POU)
F7	Build the active POU
Ctrl+F7	Build All
F8	Download the program
Ctrl+Z	Undo in the current pane
Ctrl+Y, Ctrl+Shift+Z	Redo
Ctrl+F	Find in this POU
F3 / Shift+F3	Next / previous match
F2	Rename the variable at the caret (body pane)
Shift+F12	Where used (body pane)
Ctrl+Space	Completion (body pane); Browse IO (declaration pane and elsewhere)
Alt+drag	Column selection
Tab	Insert four spaces
Esc	Close the find bar, the completion list or a dialog

Keys for online work and debugging (F5, F10, F11, Shift+F11, and the write and force keys of the variable tables) are listed in Chapters 22 and 24 and in Appendix E.

Note: In the online variable tables, F7, Ctrl+F7 and Alt+F7 force, write and release values (Chapter 22). While a variable table is on screen these keys act only on the table; they do not start **Build** or **Build All**.

14. Structured Text reference

This chapter is the language reference for Zeus Structured Text (ST): every lexical element, data type, declaration, operator and statement the compiler accepts, with the rules it applies and the run-time behaviour of the result. Structured Text is also what the other two languages become: a Ladder POU (Chapter 16) and a Sequential Function Chart (Chapter 17) are compiled as ST. Use this chapter when you write or review code, and when a construct you know from another IEC 61131-3 environment is rejected - Section 14.26 lists every such difference.

Everything here was checked against the compiler itself (`bin\zeus-check.exe` , the same compiler the controller runs). Every example in this chapter compiles as shown.

14.1 How a program is compiled

Zeus compiles a whole application as **one linked unit**. When you build a POU in the IDE, the compiler is given, in this order:

1. the data types (DUTs, `dut/*.st` , Chapter 15);
2. `globals.st` and the generated IO alias declarations (Chapter 11);
3. the further global variable lists (`gv1/*.st` , Chapter 19);
4. the function-block library (`lib/*.st` , Chapter 19);
5. the project's other enabled function blocks and functions;
6. the POU being built (its editor text, saved or not).

A download bundles the programs of the task call list instead of a single POU (Chapter 20). The consequences for the language are:

- There is **one symbol namespace** for globals and **one namespace for POU names**. A global list does not create a scope: `MyList.myVar` does not exist.
- A POU may call a function or function block declared later in the unit. Order does not matter between POUs.
- Order **does** matter between `TYPE` declarations: a type used inside another type must be declared first (Section 14.5).
- The unit may hold at most 256 POUs, counting library blocks, function blocks, methods, functions and programs (Appendix D).

A file that holds only statements (no `PROGRAM` header) is accepted as an implicit `PROGRAM Main` ; the IDE's editors never produce one, but it is convenient when you paste a snippet into `zeus-check` .

14.2 Lexical elements

14.2.1 Character set, case and layout

Source files are plain text, UTF-8 or ASCII. A UTF-8 byte-order mark at the start of a file is skipped. Identifiers and keywords use ASCII letters only; non-ASCII characters are allowed in comments and string literals.

Keywords, type names and identifiers are **case-insensitive**: `IF` , `If` and `if` are the same keyword, and `Speed` , `SPEED` and `speed` are the same variable. The spelling of the declaration is the one the IDE shows.

Spaces, tabs and line breaks separate tokens and are otherwise ignored. A statement may span several lines, and several statements may share a line.

14.2.2 Identifiers

An identifier starts with a letter or an underscore, followed by letters, digits and underscores (`_x`, `Pump_1`, `a__b` are all valid). The first **63 characters** are significant; two names that differ only after the 63rd character are the same name.

An identifier must not be a reserved word (Section 14.2.3).

14.2.3 Reserved words

The following words are reserved everywhere and cannot be used as names:

Group	Words
POUs	PROGRAM END_PROGRAM FUNCTION END_FUNCTION FUNCTION_BLOCK END_FUNCTION_BLOCK METHOD END_METHOD
Access specifiers	PUBLIC PRIVATE PROTECTED INTERNAL
Types	TYPE END_TYPE STRUCT END_STRUCT ARRAY OF
Declarations	VAR VAR_INPUT VAR_OUTPUT VAR_IN_OUT VAR_GLOBAL VAR_TEMP VAR_EXTERNAL END_VAR CONSTANT RETAIN PERSISTENT AT
Statements	IF THEN ELIF ELSE END_IF CASE END_CASE FOR TO BY DO END_FOR WHILE END_WHILE REPEAT UNTIL END_REPEAT EXIT CONTINUE RETURN
Operators and literals	AND OR XOR NOT MOD TRUE FALSE
Elementary types	BOOL BYTE WORD DWORD LWORD SINT INT DINT LINT USINT UINT UDINT ULINT REAL LREAL TIME
Charts	INITIAL_STEP

`STEP`, `END_STEP`, `TRANSITION`, `END_TRANSITION`, `FROM`, `ACTION` and `END_ACTION` are keywords **only inside a chart** (Chapter 17); everywhere else they are ordinary identifiers, so a variable may be called `step`. `STRING` is recognised as a type name but is not reserved.

Note: Short words that are common variable names in other code are reserved here: `by`, `to`, `do`, `of`, `at` and `time`. A declaration such as `time : INT;` is rejected with "expected a variable name".

14.2.4 Comments

Form	Extent
<code>(* ... *)</code>	Block comment. May span lines. Nests: <code>(* a (* b *) c *)</code> is one comment.
<code>// ...</code>	Line comment, to the end of the line.

Because block comments nest, you can comment out a section of code that already contains comments. An unclosed `(*` is reported as "unterminated comment - missing '*')'" at the position where it opens.

14.2.5 Pragmas

Text in braces, `{ ... }`, is a pragma. The compiler reads past it like a comment and it has no effect. This lets code that carries pragmas such as `{attribute 'qualified_only'}` or `{warning 'todo'}` be pasted in unchanged. A `{` without a closing `}` is an error.

```
{attribute 'qualified_only'}
PROGRAM PragmaDemo
VAR
    x : INT;    (* a comment (* nested *) still a comment *)
    y : INT;    // a line comment
END_VAR
    x := 1;
    y := x;
END_PROGRAM
```

14.2.6 Statement terminators

Every assignment and call statement ends with `;`. After `END_IF`, `END_CASE`, `END_FOR`, `END_WHILE`, `END_REPEAT`, `END_VAR`, `END_TYPE` and the POU end keywords the semicolon is optional. A lone `;` is an empty statement and is allowed anywhere a statement is.

14.3 Literals

14.3.1 Integer literals

Form	Example	Value
Decimal	1000, -42	1000, -42
With separators	1_000_000	1000000
Hexadecimal	16#FF, 16#ff_ff	255, 65535
Octal	8#777	511
Binary	2#1010_1100	172
Typed	INT#16, DINT#-5	16, -5

- Underscores may appear anywhere among the digits and are ignored.
- Only the bases 2, 8 and 16 are accepted. `3#12` is an error.
- Hexadecimal digits are case-insensitive.
- A minus sign is the unary operator applied to the literal; `-5` is `-(5)`.
- The type prefix of a typed literal (`INT#`) is accepted and ignored: the literal is an integer constant like any other. `BOOL#1` is therefore an integer, and assigning it to a `BOOL` is an error; write `TRUE`.
- An integer literal is typed `DINT` for type checking. Its value may be anything that fits in 64 bits.

14.3.2 Real literals

Form	Example
Decimal point	3.14159, 0.5, -2.0

Form	Example
Exponent	1.5E3 , 1e-3 , 2.5e+2
Typed	REAL#1.5 , LREAL#2

A real literal needs a digit before the decimal point: `.5` is rejected, write `0.5`. A real literal is typed `LREAL` and held in double precision; it is rounded to single precision when stored in a `REAL`.

14.3.3 Boolean literals

`TRUE` and `FALSE` (any case). There is no implicit conversion between `BOOL` and numbers (Section 14.10).

14.3.4 Duration literals

A duration literal is `T#` or `TIME#` followed by one or more components, largest unit first:

Unit	Meaning
d	days
h	hours
m	minutes
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds

Examples: `T#500ms`, `T#5s`, `T#1h30m`, `T#2m30s`, `TIME#10s`, `T#1.5s`, `T#1m_30s`, `T#1d2h3m4s5ms6us7ns`, `T#-5s`.

- Units and the `T#` prefix are case-insensitive (`t#500MS` is valid).
- A component may have a fraction (`T#1.5s` is 1500 ms).
- An underscore may separate components.
- A leading `-` makes the whole duration negative.
- `LTIME#`, `D#`, `TOD#` and `DT#` do not exist.

14.3.5 String literals

A string literal is enclosed in single quotes, `'Pump 1'`; double quotes, `"Pump 1"`, are accepted as well. The dollar sign is the escape character:

Escape	Character
<code>\$'</code>	single quote
<code>\$"</code>	double quote
<code>\$\$</code>	dollar sign
<code>\$L</code> or <code>\$N</code>	line feed
<code>\$R</code>	carriage return

Escape	Character
\$T	tab
\$P	form feed
\$hh	the byte with hexadecimal code hh, e.g. \$0A

Letters in escapes are case-insensitive. A literal longer than 254 characters is cut to 254, with a warning.

```

PROGRAM LiteralDemo
VAR
  a : DINT; r : REAL; t : TIME; b : BOOL; w : WORD; s : STRING;
END_VAR
a := 1_000_000;
a := 16#FF_FF;
a := 8#777;
a := 2#1010_1100;
a := INT#16;
r := 1.5E3;
r := 2.5e-2;
t := T#1h30m;
t := T#1.5s;
t := t#500ms;
b := TRUE;
w := 16#ffff;
s := 'It's 20$ - line one$Nline two';
END_PROGRAM

```

14.4 Elementary data types

Type	Size	Range / precision	Notes
BOOL	1 byte	FALSE, TRUE	Occupies a whole byte in data memory. At a direct address it is one bit.
BYTE	1 byte	0 .. 255	Bit string
WORD	2 bytes	0 .. 65 535	Bit string
DWORD	4 bytes	0 .. 4 294 967 295	Bit string
LWORD	8 bytes	0 .. 2 ⁶⁴ - 1	Bit string (see note)
SINT	1 byte	-128 .. 127	
INT	2 bytes	-32 768 .. 32 767	
DINT	4 bytes	-2 147 483 648 .. 2 147 483 647	
LINT	8 bytes	-2 ⁶³ .. 2 ⁶³ - 1	
USINT	1 byte	0 .. 255	
UINT	2 bytes	0 .. 65 535	
UDINT	4 bytes	0 .. 4 294 967 295	

Type	Size	Range / precision	Notes
ULINT	8 bytes	0 .. 2 ⁶⁴ - 1	See note
REAL	4 bytes	IEEE 754 single, about 7 significant digits	
LREAL	8 bytes	IEEE 754 double, about 15 significant digits	
TIME	8 bytes	signed, nanosecond resolution, about ±292 years	See Section 14.4.1
STRING[n]	n + 1 bytes	1 to 254 characters; STRING alone is STRING[80]	See Section 14.20

All integer arithmetic is carried out in 64 bits and the result is fitted to the destination when it is stored (Section 14.9.2). The bit-string types (BYTE .. LWORD) and the integer types of the same width are interchangeable in expressions.

Note: 64-bit values are compared as **signed** numbers. An ULINT or LWORD value of 2⁶³ or more takes part in a comparison as a negative number. Keep such values below 2⁶³ if you compare them, or compare bit patterns with AND masks instead.

14.4.1 TIME

A TIME holds a signed duration in **nanoseconds**. T# literals, the timer inputs PT, the timer outputs ET and TIME_NOW() all use this type.

- TIME + TIME, TIME - TIME, TIME * integer and TIME / integer give a TIME.
- TIME values compare with each other and with T# literals.
- TIME_TO_INT(t) converts to **milliseconds** (DINT); INT_TO_TIME(ms) converts back.

Important: Assigning a plain integer to a TIME, or a TIME to an integer, is accepted without a warning and moves the raw **nanosecond** count. t := 5; sets 5 ns, not 5 ms or 5 s. Always write a T# literal or use INT_TO_TIME / TIME_TO_INT.

```

PROGRAM TimeDemo
VAR
    fill_time : TIME := T#2m;
    elapsed   : TIME;
    started   : TIME;
    ms        : DINT;
    overdue   : BOOL;
END_VAR
elapsed := TIME_NOW() - started;
overdue := elapsed > fill_time + T#30s;
ms      := TIME_TO_INT(elapsed);      (* milliseconds *)
fill_time := INT_TO_TIME(90000);      (* 90 s *)
END_PROGRAM

```

14.4.2 Types that do not exist

DATE, TIME_OF_DAY / TOD, DATE_AND_TIME / DT, LTIME, WSTRING, CHAR, WCHAR, POINTER TO, REFERENCE TO, sub-ranges (INT(0..100)) and UNION are not supported. The controller has no calendar-time type in ST; the scan clock (TIME_NOW()) is monotonic.

14.5 Derived data types

Derived types are declared in a TYPE ... END_TYPE block. In a project they live in the Data Types (DUT) folder, one type per file (Chapter 15); in a plain file they may appear anywhere outside a POU. One TYPE block may hold several declarations.

Kind	Syntax	Section
Structure	Name : STRUCT ... END_STRUCT;	14.17
Enumeration	Name : (A, B, C := 10);	14.18
Alias	Name : BaseType;	below
Array (as an alias)	Name : ARRAY[1..10] OF INT;	14.16

An **alias** gives a new name to any type, including STRING[n] and arrays. It is the same type as its base: an alias of REAL is a REAL, and two variables of the same array alias can be copied whole.

A type used inside another type - a structure member of a structure type, say - must be declared **before** the type that uses it. The IDE links DUT files in a fixed order before everything else (Chapter 15).

```

TYPE
  Speed  : REAL;
  Name20 : STRING[20];
  Vec3   : ARRAY[1..3] OF REAL;
  Point  : STRUCT
    x : REAL;
    y : REAL;
  END_STRUCT;
  Path   : STRUCT
    start : Point;
    finish : Point;
    label  : Name20;
  END_STRUCT;
END_TYPE

PROGRAM TypeDemo
VAR
  v : Speed; n : Name20; a : Vec3; b : Vec3; p : Path;
END_VAR
  v := 1.5;
  n := 'Line A';
  a[1] := v;
  b := a;          (* whole-array copy: same alias *)
  p.start.x := 10.0;
  p.label := n;
END_PROGRAM

```

14.6 Program organisation units

14.6.1 The four kinds

Kind	Keeps state	Called by	Storage	Use for
PROGRAM	Yes	The task (Chapter 18)	Static, one copy	The top level of the application
FUNCTION_BLOCK	Yes, per instance	Its instances	Inside each instance	Anything with memory: a pump, a valve, a loop, a sequence
FUNCTION	No	Any POU	On the call stack, fresh each call	Pure calculations
METHOD	No (its own locals); works on the block's state	The function block it belongs to	Its locals on the call stack; the block's variables in the instance	An operation of a block

14.6.2 General structure

```

PROGRAM Name | FUNCTION_BLOCK Name | FUNCTION Name : ReturnType
  <declaration blocks, any number, any order>
  <METHOD ... END_METHOD blocks - FUNCTION_BLOCK only>
  <statements>
END_PROGRAM | END_FUNCTION_BLOCK | END_FUNCTION

```

- Declaration blocks come first, then (in a function block) the methods, then the body.
- A POU name is unique across the unit, case-insensitively. A second POU with the same name is an error, "a POU named 'X' is already defined".
- A FUNCTION_BLOCK name is also a type name: instances are declared with it.

14.6.3 PROGRAM

A program's variables are allocated once and keep their values from scan to scan. A program cannot be called from another POU ("A" is a PROGRAM and cannot be called as a function), and another POU cannot read a program's variables (A.x is not valid); share data through global variables (Chapter 19).

The task runs the programs of its call list in order, every scan (Chapter 18). A program may declare VAR_INPUT and VAR_OUTPUT blocks; they behave as ordinary program variables.

```

PROGRAM Conveyor
VAR
  start_pb : BOOL;
  stop_pb  : BOOL;
  running  : BOOL;
  run_timer : TON;
END_VAR
running := (start_pb OR running) AND NOT stop_pb;
run_timer(IN := running, PT := T#10m);
END_PROGRAM

```

14.6.4 FUNCTION_BLOCK

A function block is a type. Each **instance** holds its own copy of every variable the block declares, including the timers and counters inside it, so ten instances of a pump block are ten independent pumps. See Section 14.12 for calling instances.

```
FUNCTION_BLOCK PumpCtl
VAR_INPUT
    run_cmd   : BOOL;
    feedback  : BOOL;
    fb_time   : TIME := T#3s;
END_VAR
VAR_OUTPUT
    motor     : BOOL;
    fault     : BOOL;
END_VAR
VAR
    fb_timer  : TON;
END_VAR
motor := run_cmd AND NOT fault;
fb_timer(IN := motor AND NOT feedback, PT := fb_time);
IF fb_timer.Q THEN
    fault := TRUE;
END_IF;
IF NOT run_cmd THEN
    fault := FALSE;
END_IF;
END_FUNCTION_BLOCK
```

14.6.5 FUNCTION

A function returns one value of an **elementary type**, written by assigning to the function's own name. It keeps no state between calls. See Section 14.13.

```
FUNCTION Scale : REAL
VAR_INPUT
    raw      : INT;
    raw_max  : INT;
    eu_max   : REAL;
END_VAR
IF raw_max = 0 THEN
    Scale := 0.0;
    RETURN;
END_IF;
Scale := INT_TO_REAL(raw) * eu_max / INT_TO_REAL(raw_max);
END_FUNCTION
```

14.6.6 METHOD

A method is declared inside a function block, after its variable blocks and before its body. See Section 14.14.

14.7 Variable declarations

14.7.1 Syntax

```
<section> [CONSTANT] [RETAIN] [PERSISTENT]
  name1 [, name2 ...] [AT %address] : type [:= initial value];
  ...
END_VAR
```

- Several names may share one declaration line, **at most 16**. Names beyond the 16th on one line are silently dropped - use a new line.
- A name may be declared once per POU. A second declaration is "'x' is already declared on line n".
- A local variable with the same name as a global variable is legal, and the POU uses the local. The compiler warns, because the IDE's monitor, ladder display, write and force resolve the name to the **global** (Chapter 19). Rename one of them.

14.7.2 Declaration sections

Section	Allowed in	Meaning
VAR	all POU's	Local variables. In a PROGRAM and a FUNCTION_BLOCK instance they keep their values between scans; in a FUNCTION or METHOD they are created, zeroed, for every call.
VAR_INPUT	all POU's	Inputs. A function block's inputs are set by the call (inst(x := 1)) or by writing inst.x ; a function's and a method's are the call's arguments, in order.
VAR_OUTPUT	PROGRAM, FUNCTION_BLOCK	Outputs. A function block's outputs are read as inst.q or copied by the call with => . In a FUNCTION or METHOD the block is accepted but the caller cannot read it - do not use it there.
VAR_IN_OUT	FUNCTION_BLOCK	A reference to the caller's variable (Section 14.15). Rejected on a METHOD; on a FUNCTION it cannot be passed, so do not use it there.
VAR_TEMP	all POU's	Scratch variables. In a FUNCTION or METHOD they are fresh each call. In a PROGRAM or FUNCTION_BLOCK they are stored like VAR and keep their value - do not rely on them being reset.
VAR_GLOBAL	file level only	Global variables: globals.st and the lists under gv1/ (Chapter 19). Not allowed inside a POU.
VAR_EXTERNAL	all POU's	Accepted for compatibility, but it declares a global variable rather than referring to one. Every global is already visible in every POU, so it is never needed; redeclaring an existing global this way is a "already declared" error.

14.7.3 Qualifiers

Qualifier	Effect
CONSTANT	The variable cannot be assigned ("N' is CONSTANT and cannot be assigned"). It is still a variable: it cannot be used as an array bound, a CASE label or another variable's initial value.

Qualifier	Effect
RETAIN	The value survives a power cycle and a restart of the runtime (Chapter 33). Effective on globals and PROGRAM variables , including a whole function-block instance declared in a <code>VAR RETAIN</code> block of a program.
PERSISTENT	Treated as <code>RETAIN</code> .

Important: `VAR RETAIN` written **inside a FUNCTION_BLOCK** has no effect: the block's variables live in the instance, and it is the instance that is retained or not. Declare the instance in a `VAR RETAIN` block of the program, or in `VAR_GLOBAL RETAIN . RETAIN` in a FUNCTION or METHOD has no effect either.

14.7.4 Initial values

`:= value` after the type sets the value the variable has after a download, a cold start or a reset.

- The value must be a **literal**: an integer, real, duration or Boolean literal, optionally with a leading minus, or an enumerator (`Run` or `E_Mode.Run`). Expressions, other variables and constants are rejected ("initial values must be literal constants").
- Only **elementary** types take an initial value. A `STRING`, an array or a structure variable cannot have one; initialise it in the body, or give the members of a structure type their own initial values in the `TYPE` (Section 14.17), which every variable of that type then inherits.
- An integer literal given to a `REAL` is converted (`x : REAL := 3;` is 3.0).
- Variables without an initial value start at zero, `FALSE`, `T#0s` or the empty string.
- Initial values of a function block's variables are applied to **every instance**, including instances inside other instances and inside arrays.

Important: In a **FUNCTION** or **METHOD**, the declared initial value of a `VAR` is not applied: its locals start at zero on every call. Assign the starting value at the top of the body instead.

```

VAR_GLOBAL CONSTANT
    MAX_PUMPS : INT := 4;
END_VAR
VAR_GLOBAL RETAIN
    batch_count : DINT;
END_VAR

PROGRAM InitDemo
VAR
    setpoint : REAL := 72.5;
    offset : INT := -5;
    enabled : BOOL := TRUE;
    delay : TIME := T#2s;
    mask : WORD := 16#00FF;
    ratio : REAL := 3;
END_VAR
VAR RETAIN
    hours_run : DINT;
END_VAR
VAR_TEMP
    scratch : INT;
END_VAR
    scratch := offset * 2;
    IF enabled THEN
        batch_count := batch_count + 1;
    END_IF;
END_PROGRAM

```

14.8 Direct addresses

A variable can be placed on the process image with `AT`, and an address can be used directly in an expression. The process image and how channels are mapped onto it are described in Chapter 11; in most projects you name channels with IO aliases instead of writing `AT` yourself.

```
%<area><size><byte>[.<bit>]
```

Part	Values
Area	I inputs, Q outputs, M memory
Size	X bit (the default), B byte, W word (16 bit), D double word (32 bit), L long word (64 bit)
Byte	Byte offset in the area, decimal
Bit	.0 to .7, bit addresses only

`%IX0.3`, `%I0.3` (X is the default), `%IW2`, `%QD8`, `%MX10.3` and `%ML16` are all valid.

Rules:

- A bit address (`%IX0.3`) requires `BOOL`; any other address requires a type of **exactly its width**: `%IW2` : `INT` or `WORD` or `UINT`, `%ID4` : `REAL` or `DINT` or `DWORD`, `%QL8` : `LREAL`. A mismatch is an error that names both widths.
- Only elementary types can be placed at an address: not a structure, array or string.

- One address per declaration line: `a, b AT %QW0 : INT;` is an error.
- `%I` is written by the fieldbus every scan. Assigning to an `%I` variable or address is an error.
- A located variable cannot be passed to a `VAR_IN_OUT` (Section 14.15).
- Whether the address exists is checked when the program is loaded onto the controller, against the configured process image - not by the compiler. A **located variable** outside the image makes the load fail with a message naming the variable and telling you to add the IO device or drop the `AT`. A **direct address written in an expression** (`%QW900 := 1;`) is only checked when it executes, and stops the scan with a fault.
- A located variable declared inside a `FUNCTION_BLOCK` is **not** instance data: every instance reads and writes the same address. Pass IO to a block through its inputs and outputs instead.

```

PROGRAM IoDemo
VAR
    start_pb AT %IX0.0 : BOOL;
    stop_pb  AT %IX0.1 : BOOL;
    level_raw AT %IW2   : INT;
    flow     AT %ID4    : REAL;
    motor    AT %QX0.0  : BOOL;
    speed_ref AT %QW2   : WORD;
    counter  AT %MD12   : DINT;
END_VAR
motor := (start_pb OR motor) AND NOT stop_pb;
%QX0.1 := %IX0.2;      (* addresses used directly *)
%MW30 := level_raw;
speed_ref := 16#4000;
counter := counter + 1;
END_PROGRAM

```

14.9 Expressions and operators

14.9.1 Operators and precedence

From highest to lowest precedence. Operators on the same row have equal precedence and are evaluated left to right, except `**`.

Precedence	Operators	Operation	Operand types
1	<code>()</code> <code>f(...)</code> <code>a[i]</code> <code>s.m</code> <code>v.3</code>	Parentheses, call, index, member, bit	-
2	<code>-</code> <code>+</code> <code>NOT</code>	Negation, unary plus, complement	<code>-</code> : numeric; <code>NOT</code> : BOOL (logical) or integer (bitwise)
3	<code>**</code> or <code>^</code>	Power, right-associative	numeric
4	<code>*</code> <code>/</code> <code>MOD</code>	Multiply, divide, modulo	numeric; <code>MOD</code> integer only
5	<code>+</code> <code>-</code>	Add, subtract	numeric, TIME
6	<code><</code> <code>></code> <code><=</code> <code>>=</code>	Comparison	numeric, TIME, STRING
7	<code>=</code> <code><></code>	Equality	all elementary types, STRING
8	<code>AND</code>	Logical / bitwise AND	BOOL, or integers

Precedence	Operators	Operation	Operand types
9	XOR	Logical / bitwise exclusive OR	BOOL, or integers
10	OR	Logical / bitwise OR	BOOL, or integers

Consequences worth knowing:

- Unary operators bind tighter than `**`: `-2 ** 2` is `(-2) ** 2 = 4`.
- `NOT a = b` is `(NOT a) = b`.
- `a OR b AND c` is `a OR (b AND c)`.
- `2.0 ** 3.0 ** 2.0` is `2.0 ** 9.0`.
- `&` is not an operator; write `AND`. `AND_THEN` and `OR_ELSE` do not exist.
- **Both operands are always evaluated**; there is no short-circuit. `IF i <= 10 AND a[i] > 0 THEN` still indexes `a[i]` when `i` is 11 - guard with a nested `IF`.

14.9.2 Arithmetic

Result type. If either operand is `REAL` or `LREAL`, the operation is done in floating point (the result is `LREAL` if either is `LREAL`, otherwise `REAL`). Otherwise, if either operand is `TIME`, the result is `TIME`. Otherwise the result is the wider of the two integer types. Mixing an integer and a real operand compiles with the warning "mixed integer and real arithmetic - the result is promoted to ...".

Width. Integer arithmetic is done in 64 bits and floating-point arithmetic in double precision, whatever the declared types. The value is fitted to the destination only when it is stored: an integer is truncated to the destination's width (two's complement wrap), a floating-point value is rounded to single precision for a `REAL`. There is no overflow fault. So with `a : INT := 30000`, `a + a > 50000` is TRUE, while `a := a + a` stores -5536.

Division.

Operation	Result
integer / integer	Integer quotient, truncated towards zero: <code>7 / 2 = 3</code> , <code>-7 / 2 = -3</code> . This is so even when the result is assigned to a <code>REAL</code> - write <code>7.0 / 2.0</code> or convert first.
MOD	Remainder with the sign of the dividend: <code>-7 MOD 2 = -1</code> . Integers only.
integer division or MOD by zero	Result 0. A minor fault is counted; the scan continues (Section 14.23).
real division by zero	IEEE 754: $\pm\text{INF}$, or NaN for 0/0. A minor fault is counted.

Power. With real operands `**` is the floating-point power. With integer operands it is repeated multiplication; a negative exponent gives 0.

14.9.3 Logical and bitwise operators

`AND`, `OR`, `XOR` and `NOT` are logical on `BOOL` and bitwise on integers. Mixing a `BOOL` and an integer is an error ("cannot mix BOOL and INT in a logical operation"); they cannot be applied to `REAL`.

Note: A bitwise `NOT` inverts all 64 bits of the intermediate value. The stored result is right (`w := NOT w`; complements a WORD), but `NOT w` used directly in a comparison is a 64-bit value: `NOT w = 16#FFFF` is never TRUE. Store it in a variable of the right type first, or mask it: `(NOT w) AND 16#FFFF`.

14.9.4 Comparison

Comparisons give `BOOL`. Numbers of different types are compared after the same promotion as arithmetic. A `BOOL` can only be compared with a `BOOL` ("cannot compare `BOOL` with `INT`"). Strings are compared by content (Section 14.20). Structures and arrays cannot be compared as a whole; compare their members.

```
PROGRAM OperatorDemo
VAR
    i : INT; d : DINT; r : REAL; b : BOOL; w : WORD; t : TIME;
END_VAR
d := 2 + 3 * 4;           (* 14 *)
d := 17 MOD 5 + 2 ** 3;   (* 2 + 8 = 10 *)
r := 2.0 ** 0.5;
r := 7.0 / 2.0;          (* 3.5 - but 7 / 2 would be 3 *)
w := (w AND 16#00FF) OR 16#0100;
w := NOT w;
b := (i > 5) AND NOT (d >= 100);
b := b XOR (r <> 0.0);
t := t + T#1s;
b := t > T#5s;
END_PROGRAM
```

14.10 Assignment and type conversion

`target := expression;` evaluates the expression and stores it. The target must be a variable, a member, an array element, a bit of an integer, or a direct address.

Target	Value	Result
Integer	Integer of any width or sign	Stored, wrapped to the target's width. No warning.
Integer	<code>REAL</code> / <code>LREAL</code>	Fraction truncated towards zero. Warning "assigning <code>REAL</code> to <code>INT</code> truncates the fractional part".
<code>REAL</code> / <code>LREAL</code>	Integer	Converted.
<code>REAL</code>	<code>LREAL</code>	Rounded to single precision.
<code>BOOL</code>	Anything but <code>BOOL</code>	Error. Use a comparison (<code>b := i <> 0;</code>) or <code>INT_TO_BOOL</code> .
Numeric	<code>BOOL</code>	Error. Use <code>BOOL_TO_INT</code> or <code>SEL</code> .
<code>TIME</code>	Integer	Raw nanoseconds. No warning. Use <code>INT_TO_TIME</code> (milliseconds).
Integer	<code>TIME</code>	Raw nanoseconds. No warning. Use <code>TIME_TO_INT</code> (milliseconds).
Enumeration	Integer	Stored; no range check.
<code>STRING</code>	<code>STRING</code> or literal	Copied, truncated to the target's capacity.
Structure / array	Same type	Copied whole (Sections 14.16, 14.17).

Explicit conversion functions are limited to those in the table below (Appendix A lists all built-in functions). A type name cannot be used as a conversion function: `INT(r)` is rejected with "'INT' is a type name, not a value". Generic

names such as `INT_TO_DINT`, `REAL_TO_LREAL` or `TO_INT` do not exist; integer and real types convert implicitly.

Function	Converts	Rounding
<code>INT_TO_REAL(i)</code> , <code>DINT_TO_REAL(i)</code>	any integer to floating point	-
<code>REAL_TO_INT(r)</code> , <code>REAL_TO_DINT(r)</code>	floating point to integer	to nearest, halves away from zero
<code>ROUND(r)</code>	floating point to integer	to nearest, halves away from zero
<code>TRUNC(r)</code>	floating point to integer	towards zero
<code>BOOL_TO_INT(b)</code>	<code>BOOL</code> to 0 / 1	-
<code>INT_TO_BOOL(i)</code>	non-zero to <code>TRUE</code>	-
<code>TIME_TO_INT(t)</code>	<code>TIME</code> to milliseconds	towards zero
<code>INT_TO_TIME(ms)</code>	milliseconds to <code>TIME</code>	-
<code>DWORD_AS_REAL(dw)</code>	reinterprets a 32-bit IEEE 754 bit pattern as <code>REAL</code>	-

Note: `REAL_TO_INT(2.5)` is 3, but `i := 2.5`; stores 2: the function rounds, the implicit conversion truncates. Say which one you mean.

```

PROGRAM ConvertDemo
VAR
    raw : INT := 13824;  pct : REAL;  whole : DINT;  on : BOOL;
    n : INT;  ms : DINT;  t : TIME;  bits : DWORD := 16#3FC00000;  f : REAL;
END_VAR
pct := INT_TO_REAL(raw) / 27648.0 * 100.0;
whole := REAL_TO_DINT(pct);           (* rounded *)
whole := TRUNC(pct);                  (* truncated *)
on := n <> 0;                          (* integer to BOOL by comparison *)
n := BOOL_TO_INT(on);
t := INT_TO_TIME(1500);               (* T#1.5s *)
ms := TIME_TO_INT(t);                 (* 1500 *)
f := DWORD_AS_REAL(bits);             (* 1.5 *)
END_PROGRAM

```

14.11 Statements

14.11.1 IF

```
IF condition THEN
    statements
{ELSIF condition THEN
    statements}
[ELSE
    statements]
END_IF;
```

The condition must be `BOOL` ("the IF condition must be `BOOL`, got `INT`"); an integer is not taken as true or false. `ELSE IF` also works, as a nested `IF` needing its own `END_IF`.

14.11.2 CASE

```
CASE selector OF
    label_list : statements
    ...
[ELSE
    statements]
END_CASE;
```

- The selector is an integer, enumeration or `BOOL` expression, evaluated once. A `REAL` selector is an error.
- A label list is one or more labels separated by commas. A label is an integer literal (may be negative), a range `lo..hi`, an enumerator, or `TRUE / FALSE`.
- Labels must be literals: a `CONSTANT` variable is not accepted. Qualified enumerators (`E_Mode.Run:`) are not accepted as labels; write the bare enumerator (`Run:`).
- The arms are tested in order and the first match runs; there is no fall-through. The compiler does not report duplicate labels - the later arm is simply never reached.
- `ELSE` runs when no label matches.
- An arm may be empty (`1: ;`).
- `CASE TRUE OF` with conditions as labels is not valid ST; use `IF / ELSIF`.
- One arm may hold at most 16 labels. Keep a CASE to 64 arms or fewer: larger statements compile, but arms after the 64th are not handled correctly.

Note: An arm whose label list **starts with an enumerator followed by a comma** (`Run, Fault: ...`) is only recognised as the **first** arm of the CASE. In any later arm, write one enumerator per arm, or begin the list with an integer.

14.11.3 FOR

```
FOR control := start TO end [BY step] DO
    statements
END_FOR;
```

- The control variable must be an integer variable.

- `step` must be a **constant** (a literal, optionally negative); it defaults to 1. A step of 0 is an error. With a negative step the loop counts down.
- `start` is evaluated once. `end` is evaluated **before every iteration**.
- After a loop that runs to completion, the control variable holds the first value past `end`.

WARNING: Because the control variable is stored in its own type, a loop whose end is the largest value of that type never ends: with `i : INT`, `FOR i := 0 TO 32767 DO` wraps `i` to -32768 and runs until the instruction budget stops the program (Section 14.23). Use a wider control variable.

14.11.4 WHILE and REPEAT

```
WHILE condition DO statements END_WHILE;
REPEAT statements UNTIL condition END_REPEAT;
```

`WHILE` tests before each pass and may run zero times; `REPEAT` tests after each pass and runs at least once. Both conditions must be `BOOL`.

A scan is one pass of the program, so a loop must never wait for something to happen in the plant: the plant does not change while the loop runs. A loop that does not end is stopped by the instruction budget and faults the program. Wait across scans with a timer, a step or a state variable instead.

14.11.5 EXIT, CONTINUE and RETURN

Statement	Effect
<code>EXIT;</code>	Leaves the innermost enclosing loop. Only inside a loop.
<code>CONTINUE;</code>	Skips to the next iteration of the innermost loop (the <code>FOR</code> step is still applied). Only inside a loop.
<code>RETURN;</code>	Leaves the current POU. In a FUNCTION or METHOD the value already assigned to its name is returned. In a PROGRAM it ends that program for this scan; the next program in the task call list still runs.

There is no `JMP` and there are no labels.

14.11.6 Call statements

A call on its own is a statement: `inst(IN := x); , Reset(); , F(1);`. An expression that is not a call and not an assignment, such as `t;` or `a = b;`, is rejected with "this expression has no effect - did you mean '='?".

```

TYPE E_Mode : (Idle, Run, Fault := 10); END_TYPE

PROGRAM StatementDemo
VAR
    mode : E_Mode;  n : DINT;  i : INT;  x : BOOL;
    a : ARRAY[0..9] OF INT;  code : INT;  total : DINT;
END_VAR

IF x THEN
    n := 1;
ELSIF n > 5 THEN
    n := 2;
ELSE
    n := 3;
END_IF;

CASE code OF
    1, 3, 5:  n := 10;
    6..9, 12: n := 20;
    -5..-1:  n := 30;
ELSE
    n := 0;
END_CASE;

CASE mode OF
    Run, Fault: x := TRUE;      (* a list of enumerators: first arm only *)
    Idle:      x := FALSE;
END_CASE;

FOR i := 0 TO 9 DO
    a[i] := i * i;
END_FOR;

FOR i := 9 TO 0 BY -1 DO
    IF a[i] = 16 THEN
        EXIT;
    END_IF;
END_FOR;

total := 0;
FOR i := 0 TO 9 BY 2 DO
    IF i = 4 THEN
        CONTINUE;
    END_IF;
    total := total + a[i];
END_FOR;

WHILE n > 0 DO
    n := n - 1;
END_WHILE;

REPEAT
    n := n + 1;
UNTIL n >= 10
END_REPEAT;
END_PROGRAM

```

14.12 Calling function blocks

14.12.1 Instances

A function block is used through an **instance**, a variable whose type is the block:

```
VAR
    fill_timer : TON;
    pump1      : PumpCtl;
END_VAR
```

Instances can be declared in a PROGRAM, in a FUNCTION_BLOCK (a block inside a block), in `VAR_GLOBAL`, as structure members and as array elements. Each has its own state.

14.12.2 Call syntax

```
instance( input := expression, ..., output => variable, ... );
```

- Arguments must be **named**: `t(IN := x, PT := T#5s)`. Positional arguments are rejected ("arguments to 't' must be named, e.g. `t(IN := x)`").
- `:=` passes a value to an input (or binds a `VAR_IN_OUT`). `=>` copies an output to a variable after the block has run. Using the wrong one is an error that says which to use.
- Arguments may be given in any order, and any of them may be left out. An input that is not given **keeps the value it had** - the last one passed, or its initial value.
- `instance()`; runs the block with its inputs as they are.
- Every `VAR_IN_OUT` must be bound at every call.

14.12.3 Reading outputs and writing inputs

After a call, read outputs as members: `pump1.motor`, `fill_timer.Q`, `fill_timer.ET`. Inputs can be set by assigning the member before the call: `fill_timer.PT := T#10s`; . Outputs cannot be assigned from outside ("Q' is an output and cannot be assigned"). Members of nested instances are reached with a path: `line.pump1.fb_timer.ET`.

14.12.4 A call used as a value

A call in an expression runs the block and yields its **first** `VAR_OUTPUT` (for a standard block, `Q`); a block with no outputs yields `FALSE`. `IF t(IN := x, PT := T#1s) THEN` is valid. Read other outputs as members.

14.12.5 Where an instance can be called

An instance is called by its **name**. An instance reached through an index or a member path cannot be called:

Declaration	Callable?
<code>t : TON;</code> in a PROGRAM, FUNCTION_BLOCK or <code>VAR_GLOBAL</code>	Yes: <code>t(...)</code>
<code>pumps : ARRAY[1..4] OF PumpCtl;</code>	No. <code>pumps[i](...)</code> is rejected. Members (<code>pumps[2].run_cmd</code>) can be read and written, but the body never runs.

Declaration	Callable?
A block inside a structure (<code>s.t : TON</code>)	No. <code>s.t(...)</code> is rejected.

For several identical machines, declare one named instance each (`pump1` , `pump2` , ...) and call them in turn, or wrap the group in a function block of its own.

WARNING: Do not declare timers, counters or function-block instances inside a **FUNCTION**. It compiles, but a standard timer or counter there is not per call - every call of the function, from every place, shares the one instance - and a user function-block instance there does not have storage of its own and does not work correctly. A function cannot keep state; put the timer in a function block. The IDE refuses to convert a block that holds a timer or counter into a function for this reason (Chapter 13).

14.12.6 Standard function blocks

The runtime provides the IEC standard blocks natively. Their parameters are fixed; Appendix B describes each in full.

Block	Inputs	Outputs
<code>TON</code> , <code>TOF</code> , <code>TP</code>	<code>IN : BOOL</code> , <code>PT : TIME</code>	<code>Q : BOOL</code> , <code>ET : TIME</code>
<code>CTU</code>	<code>CU</code> , <code>R : BOOL</code> , <code>PV : INT</code>	<code>Q : BOOL</code> , <code>CV : INT</code>
<code>CTD</code>	<code>CD</code> , <code>LD : BOOL</code> , <code>PV : INT</code>	<code>Q : BOOL</code> , <code>CV : INT</code>
<code>CTUD</code>	<code>CU</code> , <code>CD</code> , <code>R</code> , <code>LD : BOOL</code> , <code>PV : INT</code>	<code>QU</code> , <code>QD : BOOL</code> , <code>CV : INT</code> (<code>Q</code> is an alias of <code>QU</code>)
<code>R_TRIG</code> , <code>F_TRIG</code>	<code>CLK : BOOL</code>	<code>Q : BOOL</code>
<code>RS</code>	<code>S</code> , <code>R1 : BOOL</code>	<code>Q1 : BOOL</code> (also <code>Q</code>)
<code>SR</code>	<code>S1</code> , <code>R : BOOL</code>	<code>Q1 : BOOL</code> (also <code>Q</code>)

All timers in one scan see the same instant, the scan's clock, and `ET` never exceeds `PT` .


```

FUNCTION_BLOCK Stage
VAR_INPUT  go : BOOL; END_VAR
VAR_OUTPUT done : BOOL; END_VAR
VAR        dwell : TON; END_VAR
    dwell(IN := go, PT := T#2s);
    done := dwell.Q;
END_FUNCTION_BLOCK

FUNCTION_BLOCK TwoStage
VAR_INPUT  go : BOOL; END_VAR
VAR_OUTPUT done : BOOL; END_VAR
VAR        first : Stage; second : Stage; END_VAR
    first(go := go);
    second(go := first.done, done => done);
END_FUNCTION_BLOCK

PROGRAM FbDemo
VAR
    line1, line2 : TwoStage;
    start, finished, both : BOOL;
    parts : CTU;
    edge : R_TRIG;
    t_left : TIME;
END_VAR
    line1(go := start, done => finished);
    both := line2(go := start) AND finished;      (* call used as a value *)

    edge(CLK := finished);
    parts(CU := edge.Q, R := FALSE, PV := 100);

    t_left := T#2s - line1.first.dwell.ET;      (* nested member path *)
    line2.go := FALSE;                          (* set an input directly *)
END_PROGRAM

```

14.13 Functions

14.13.1 Declaring and returning

```

FUNCTION Name : Return Type
VAR_INPUT ... END_VAR
VAR ... END_VAR
    statements, including Name := value;
END_FUNCTION

```

- The return type is required and must be **elementary**. A function cannot return a **STRING**, structure or array.
- The result is the last value assigned to the function's name. If none is assigned, the result is zero.
- **RETURN;** leaves the function early.
- A function may call other functions, and may call itself; the call depth is limited to 32 at run time (Section 14.23).

14.13.2 Arguments

A function takes its `VAR_INPUT` variables as arguments, in declaration order, and **exactly that many**: `F(1)` for one input. `F()` calls a function with no inputs.

WARNING: Arguments to a user function are bound **by position only**. The compiler accepts `F(b := 1, a := 5)`, but ignores the names and passes 1 to the first input and 5 to the second. Always write function arguments positionally, in the order of the declaration.

- A structure or array input is passed by value: the whole variable is copied into the function. The argument must be a variable of the same type.
- A `STRING` input is not supported: a string argument is rejected ("a string can only be assigned or compared, not used as a value here"). Pass a fixed-length code, or use a function block with a `VAR_IN_OUT` string.
- `VAR_OUTPUT` and `VAR_IN_OUT` on a function cannot be passed or read; a function returns exactly one value.
- Local variables are created, zeroed, on every call, and declared initial values are **not** applied (Section 14.7.4).

14.13.3 Calling

A function call is an expression and can appear anywhere a value can: `r := Scale(raw, 27648, 100.0) * 2.0;`. It can also stand alone as a statement, discarding the result.

```

TYPE Sample : STRUCT
    lo : REAL;
    hi : REAL;
END_STRUCT
END_TYPE

FUNCTION Clamp : REAL
VAR_INPUT
    x : REAL;
    lim : Sample;
END_VAR
    Clamp := LIMIT(lim.lo, x, lim.hi);
END_FUNCTION

FUNCTION Average4 : REAL
VAR_INPUT
    v : ARRAY[1..4] OF REAL;
END_VAR
VAR
    i : INT;
    sum : REAL;
END_VAR
    sum := 0.0;                                (* explicit: a FUNCTION ignores initial values *)
    FOR i := 1 TO 4 DO
        sum := sum + v[i];
    END_FOR;
    Average4 := sum / 4.0;
END_FUNCTION

PROGRAM FunctionDemo
VAR
    readings : ARRAY[1..4] OF REAL;
    window : Sample;
    avg : REAL;
END_VAR
    window.lo := 0.0;
    window.hi := 100.0;
    avg := Clamp(Average4(readings), window);    (* positional arguments *)
END_PROGRAM

```

14.14 Methods

A method is an operation of a function block. It is written inside the block, after the block's variable declarations and before its body:

```

METHOD [PUBLIC | PRIVATE | PROTECTED | INTERNAL] Name [: ReturnType]
    VAR_INPUT ... END_VAR
    VAR ... END_VAR
    statements
END_METHOD

```

- A method reads and writes the variables of the instance it runs on - the block's inputs, outputs and locals - as if they were its own.
- Its own `VAR_INPUT`, `VAR` and `VAR_TEMP` are fresh on every call (zeroed; initial values not applied). A local of the same name as a block variable hides the block variable inside the method.

- The return type is optional; if given, it must be elementary and the value is returned by assigning to the method's name.
- `VAR_IN_OUT` is rejected on a method. `VAR_OUTPUT` is accepted but cannot be read by the caller.
- Access specifiers are accepted and ignored.
- Arguments are positional, exactly as for a function (Section 14.13.2), including the warning about names.
- A method is called **by its bare name from inside the same function block** - from the block's body or from another method: `Reset();` , `x := Scaled(0.5);` . Its internal name is `Block.Method` , so two blocks may each have a method of the same name; `Block` and `Method` together must be shorter than 64 characters.

Important: A method cannot be called from outside its block. `pump1.Reset();` is rejected ("this expression has no effect"). To trigger a method from outside, give the block a command input and call the method from the block's body when the input is set.

In the IDE a method is edited in its own tab, and the function block's tab hides the `METHOD` blocks (Chapter 13).

```
FUNCTION_BLOCK Valve
VAR_INPUT
    open_cmd  : BOOL;
    reset_cmd : BOOL;
END_VAR
VAR_OUTPUT
    is_open : BOOL;
    cycles  : DINT;
END_VAR
METHOD PUBLIC ResetCount
    cycles := 0;
END_METHOD
METHOD PRIVATE Wear : REAL
VAR_INPUT
    k : REAL;
END_VAR
Wear := DINT_TO_REAL(cycles) * k;
END_METHOD
IF open_cmd AND NOT is_open THEN
    cycles := cycles + 1;
END_IF;
is_open := open_cmd;
IF reset_cmd OR Wear(0.001) > 50.0 THEN
    ResetCount();
END_IF;
END_FUNCTION_BLOCK
```

14.15 VAR_IN_OUT (by reference)

A `VAR_IN_OUT` of a function block refers to the caller's variable rather than holding a copy. Reading it reads the caller's variable; writing it writes the caller's variable, immediately.

- Only on a `FUNCTION_BLOCK` .
- Must be bound at **every** call, with `:=` ("v' is VAR_IN_OUT on 'B' and must be given a variable at every call").
- The argument must be a variable, member or array element - not an expression, not a direct address (copy the IO to a variable, pass that, and copy back), and not a local variable of a FUNCTION.

- Any type can be passed this way, including structures, arrays, strings and blocks. It is the way to hand a block a large array without copying it, or to let a block work on a structure that belongs to the caller.

```

FUNCTION_BLOCK SumArray
VAR_IN_OUT
    data : ARRAY[1..5] OF REAL;
END_VAR
VAR_OUTPUT
    total : REAL;
END_VAR
VAR
    i : INT;
END_VAR
    total := 0.0;
    FOR i := 1 TO 5 DO
        total := total + data[i];
        data[i] := 0.0;          (* clears the CALLER's array *)
    END_FOR;
END_FUNCTION_BLOCK

PROGRAM InOutDemo
VAR
    summer : SumArray;
    buffer : ARRAY[1..5] OF REAL;
    result : REAL;
END_VAR
    summer(data := buffer, total => result);
END_PROGRAM

```

14.16 Arrays

```
name : ARRAY[lo..hi {, lo..hi}] OF element_type;
```

- Up to **3 dimensions**. Bounds are integer literals and may be negative (`ARRAY[-5..5] OF DINT`); the low bound must not exceed the high bound. A `CONSTANT` cannot be used as a bound.
- The element type may be any type: elementary, `STRING[n]`, a structure, another array, or a function block (but see Section 14.12.5).
- One array may occupy at most 8 MiB.
- Index with `a[i]`; multi-dimensional arrays with `m[i, j]` or `m[i][j]`, which mean the same. Indexes are integer expressions ("an array index must be an integer").
- Every index is checked at run time.** An index outside its declared range stops the scan with a fault naming the index and the bounds (Section 14.23). A constant index out of range is not caught by the compiler.
- A whole array can be assigned from another array of the same shape - same bounds, same element type: `b := a;`. Arrays of different bounds are not assignable, even with the same number of elements.
- An array cannot have an initial value; fill it in the body (for example on the first scan).

```

TYPE Pt : STRUCT
  x : REAL;
  y : REAL;
END_STRUCT
END_TYPE

PROGRAM ArrayDemo
VAR
  a      : ARRAY[1..10] OF INT;
  b      : ARRAY[1..10] OF INT;
  grid   : ARRAY[1..3, 1..4] OF REAL;
  cube   : ARRAY[0..1, 0..1, 0..1] OF BOOL;
  offs   : ARRAY[-5..5] OF DINT;
  pts    : ARRAY[1..4] OF Pt;
  names  : ARRAY[1..5] OF STRING[20];
  i, j   : INT;
END_VAR
  FOR i := 1 TO 3 DO
    FOR j := 1 TO 4 DO
      grid[i, j] := 0.0;
    END_FOR;
  END_FOR;
  grid[2][3] := 1.5;           (* same element as grid[2, 3] *)
  cube[0, 1, 1] := TRUE;
  offs[-5] := 1;
  pts[2].x := 3.0;
  names[1] := 'Feed pump';
  a[1] := 5;
  b := a;                      (* whole-array copy *)
END_PROGRAM

```

14.17 Structures

```

TYPE Name :
STRUCT
  member : type [:= initial value];
  ...
END_STRUCT
END_TYPE

```

- A structure may have up to 128 members. Member names are unique within the structure.
- Members may be of any type, including other structures (declared earlier), arrays and strings. A structure cannot contain itself.
- Elementary members may have initial values; every variable of the type, and every array element of the type, starts with them.
- Access members with `. : door.Mode , plant.doors[2].Name`.
- A whole structure can be assigned from another of the **same type** (`a := b;`), passed by value to a function-block input or a function input, and returned through a function-block output. Two structure types with identical members are still different types.
- A structure that contains a function block (a `TON` member, say) cannot be copied whole or passed by value - the copy would share the timer's state. Copy the members, or pass it `VAR_IN_OUT`.
- `EXTENDS` (structure inheritance) and `UNION` are not supported.

```

TYPE E_DoorMode : (Auto, Manual, Locked); END_TYPE

TYPE DoorCfg :
STRUCT
    mode      : E_DoorMode := E_DoorMode.Auto;
    buzzer_sec : INT := 5;
    alarm_sec  : INT;
    name       : STRING[20];
END_STRUCT
END_TYPE

TYPE Plant :
STRUCT
    main_door : DoorCfg;
    doors     : ARRAY[1..3] OF DoorCfg;
END_STRUCT
END_TYPE

FUNCTION_BLOCK Door
VAR_INPUT  cfg : DoorCfg; END_VAR
VAR_OUTPUT secs : INT;    END_VAR
    secs := cfg.buzzer_sec + cfg.alarm_sec;
END_FUNCTION_BLOCK

PROGRAM StructDemo
VAR
    g      : DoorCfg;
    site   : Plant;
    d1     : Door;
    total  : INT;
END_VAR
    g.alarm_sec := 7;
    g.name := 'Gate';
    site.main_door := g;           (* whole-structure copy *)
    site.doors[2] := g;
    d1(cfg := g);                 (* passed by value *)
    total := d1.secs;             (* 5 + 7 *)
    IF site.doors[2].mode = Auto THEN
        site.doors[2].mode := Locked;
    END_IF;
END_PROGRAM

```

14.18 Enumerations

```

TYPE Name : (Enumerator1 [:= value], Enumerator2 [:= value], ...); END_TYPE

```

- Enumerators are numbered 0, 1, 2 ... in order; `:= value` sets a value (an integer literal, may be negative) and numbering continues from it: (`Low := 10, Mid, High := 30`) gives 10, 11, 30.
- An enumeration variable is stored as a `DINT` (4 bytes).
- An enumerator can be written bare (`Run`) or qualified by its type (`E_Mode.Run`), in expressions and initial values. In a `CASE` label only the bare form is accepted.
- **All enumerators share one namespace**, project-wide. The same enumerator name cannot appear in two enumerations ("enumerator 'Run' is already defined"), and an enumerator cannot have the name of a type.

Prefix enumerators if needed: (M_Idle, M_Run) and (V_Closed, V_Open) .

- At most 256 enumerators in the whole unit, library included.
- Enumeration values mix freely with integers: `m := 7;` is accepted, and `i := m + 1;` works. There is no range check.
- A variable with the same name as an enumerator hides the enumerator in that POU.
- A base type after the list ((A, B) INT) is not supported.

```

TYPE
  E_Mode   : (Idle, Running, Faulted := 10);
  E_Level  : (Low := 10, Mid, High := 30);
END_TYPE

PROGRAM EnumDemo
VAR
  mode     : E_Mode;
  level    : E_Level := E_Level.High;
  code     : DINT;
END_VAR
  IF mode = E_Mode.Idle THEN
    mode := Running;
  END_IF;
  CASE mode OF
    Idle:    code := 0;
    Running: code := 1;
    Faulted: code := 99;
  END_CASE;
  code := code + level;           (* enumerations are integers *)
END_PROGRAM

```

14.19 Bit access

Any bit of an integer or bit-string variable is addressed with `.n`, where `n` is a literal from 0 (least significant) to the width minus one:

- `b := w.0;` reads bit 0 as a `BOOL` .
- `Alarm.15 := TRUE;` sets bit 15; assigning `FALSE` clears it. The other bits are unchanged.
- It works on members and array elements: `s.status.4` , `a[i].2` .
- `n` must be a literal; `w.i` with a variable is not bit access. Use `SHR(w, i) AND 1` or a mask.
- A bit number beyond the type ("bit 8 is out of range for BYTE (0..7)"), or bit access on `BOOL` , `REAL` or `TIME` , is an error.


```

PROGRAM BitDemo
VAR
    status : WORD;
    alarms : DWORD;
    faulted : BOOL;
    pos : INT := 3;
    bit_n : BOOL;
END_VAR
    faulted := status.0 OR status.7;
    alarms.15 := faulted;
    IF alarms.31 THEN
        status.3 := FALSE;
    END_IF;
    bit_n := (SHR(status, pos) AND 1) <> 0;    (* bit chosen at run time *)
END_PROGRAM

```

14.20 Strings

```

name : STRING;          (* 80 characters *)
name : STRING[n];       (* n = 1 .. 254 *)

```

A string occupies $n + 1$ bytes: a length byte and n characters. Characters are bytes; there is no Unicode string type.

What you can do with a string:

Operation	Rule
Assign a literal or another string	<code>s := 'Pump';</code> , <code>s := t;</code> . A longer value is truncated to the target's capacity - it never overwrites the next variable.
Compare	<code>=</code> , <code><></code> , <code><</code> , <code>></code> , <code><=</code> , <code>>=</code> compare the content byte by byte; a prefix sorts before the longer string (<code>'AB' < 'ABC'</code>). Only a string with a string.
Structure member, array element, function-block variable	Yes.
Pass to a function block	Through a <code>VAR_IN_OUT</code> , or by assigning the member before the call: <code>blk.name := 'P1'; blk();</code> .
Read from a function block	As a member after the call: <code>s := blk.label;</code> .

What you cannot do:

- Initial values: `s : STRING := 'abc';` is rejected. Assign it in the body.
- Arithmetic or concatenation: `s + 'x'` is an error.
- String functions: `LEN` , `CONCAT` , `LEFT` , `RIGHT` , `MID` , `FIND` , `INSERT` , `DELETE` , `REPLACE` and conversions to or from numbers do not exist.
- Indexing a character (`s[1]`).
- Passing a string to a `FUNCTION` or `METHOD` input, returning one from a function, or a string argument in a function-block call (`blk(name := 'P1')`).
- Placing a string at a direct address.

Important: Do not bind a `STRING` output with `=> ( blk(label => s) )`. It compiles, but does not copy the text correctly. Read the output as a member after the call: `s := blk.label;`

```

FUNCTION_BLOCK Tagged
VAR_INPUT  tag  : STRING[16]; END_VAR
VAR_OUTPUT label : STRING[24]; is_p1 : BOOL; END_VAR
    is_p1 := tag = 'P1';
    label := tag;
END_FUNCTION_BLOCK

FUNCTION_BLOCK Stamp
VAR_IN_OUT text : STRING[24]; END_VAR
    IF text = '' THEN
        text := 'unnamed';
    END_IF;
END_FUNCTION_BLOCK

PROGRAM StringDemo
VAR
    t1      : Tagged;
    st      : Stamp;
    shown   : STRING[24];
    short   : STRING[4];
    before  : BOOL;
END_VAR
    t1.tag := 'P1';                (* string input: assign the member *)
    t1();
    shown := t1.label;             (* string output: read the member *)
    st(text := shown);             (* VAR_IN_OUT works *)
    short := 'ABCDEFGH';           (* truncated to 'ABCD' *)
    before := 'APPLE' < 'BANANA';
END_PROGRAM

```

14.21 Built-in functions

Built-in functions take **positional** arguments (`LIMIT(0, x, 100)`, not `LIMIT(MN := 0, ...)`) and are listed in full, with examples, in Appendix A. This is a summary with the points that affect how you write code.

Group	Functions	Notes
Arithmetic	<code>ABS</code> , <code>SQRT</code> , <code>LN</code> , <code>LOG</code> , <code>EXP</code> , <code>SIN</code> , <code>COS</code> , <code>TAN</code> , <code>ASIN</code> , <code>ACOS</code> , <code>ATAN</code>	Return <code>LREAL</code> , so <code>i := ABS(i)</code> ; compiles with a truncation warning. Angles in radians. <code>SQRT</code> of a negative number and <code>LN</code> / <code>LOG</code> of zero or less stop the scan (a major fault), unlike a division by zero.
Selection	<code>MIN(a, b, ...)</code> , <code>MAX(a, b, ...)</code> , <code>LIMIT(mn, x, mx)</code> , <code>SEL(g, in0, in1)</code> , <code>MUX(k, v0, v1, ...)</code>	A built-in function takes at most 8 arguments at run time (more compile, but fault when executed), so <code>MIN</code> / <code>MAX</code> compare up to 8 values and <code>MUX</code> selects among up to 7. <code>SEL</code> returns <code>in1</code> when <code>g</code> is <code>TRUE</code> . A <code>MUX</code> selector outside <code>0..n-1</code> stops the scan.

Group	Functions	Notes
Bit shift	<code>SHL(x, n)</code> , <code>SHR(x, n)</code> , <code>ROL(x, n)</code> , <code>ROR(x, n)</code>	Work on the 64-bit value. <code>SHL</code> / <code>SHR</code> give the expected result once stored in the variable's type. <code>ROL</code> / <code>ROR</code> rotate through 64 bits, not through the width of <code>x</code> : <code>ROL</code> of a <code>BYTE</code> does not bring bit 7 round to bit 0.
Conversion	<code>INT_TO_REAL</code> , <code>DINT_TO_REAL</code> , <code>REAL_TO_INT</code> , <code>REAL_TO_DINT</code> , <code>TRUNC</code> , <code>ROUND</code> , <code>BOOL_TO_INT</code> , <code>INT_TO_BOOL</code> , <code>TIME_TO_INT</code> , <code>INT_TO_TIME</code> , <code>DWORD_AS_REAL</code>	Section 14.10.
Time	<code>TIME_NOW()</code>	The scan's monotonic clock as a <code>TIME</code> : the same value for the whole scan, the same clock the timers use. Measure intervals by subtracting two readings; the absolute value has no calendar meaning.
Process image	<code>PI_IN_BYTE</code> , <code>PI_OUT_BYTE</code> , <code>PI_SET_OUT_BYTE</code>	Section 14.22.

Any other name followed by `(` that is not a function, function block instance or method in scope is "unknown function or function block".

14.22 Process-image byte access (PI_* functions)

Some modules exchange a block of bytes whose position is only known at run time - an IO-Link ISDU channel, an event area, a variable-length telegram (Chapter 12). A `%` address is fixed at compile time and cannot express this, so three functions address the process image by a computed byte offset:

Call	Result
<code>PI_IN_BYTE(off)</code>	The input byte <code>%IB[off]</code> as 0..255, or -1 if <code>off</code> is outside the input image.
<code>PI_OUT_BYTE(off)</code>	The output byte <code>%QB[off]</code> read back, or -1 if outside.
<code>PI_SET_OUT_BYTE(off, v)</code>	Writes <code>v AND 16#FF</code> to <code>%QB[off]</code> . Returns <code>TRUE</code> , or <code>FALSE</code> (and writes nothing) if <code>off</code> is outside.

They never fault: an out-of-range offset is reported through the return value, so a block can detect "no module there" and say so. `DWORD_AS_REAL` completes the set: assemble four bytes into a `DWORD` (mind the device's byte order) and reinterpret it as an IEEE 754 `REAL`.

```

PROGRAM PiDemo
VAR
    base : DINT := 16;      (* start of the module's input bytes *)
    b0, b1, b2, b3 : DINT;
    raw   : DWORD;
    value : REAL;
    ok    : BOOL;
END_VAR
    b0 := PI_IN_BYTE(base);
    b1 := PI_IN_BYTE(base + 1);
    b2 := PI_IN_BYTE(base + 2);
    b3 := PI_IN_BYTE(base + 3);
    IF b0 >= 0 AND b3 >= 0 THEN
        (* big-endian Float32 as sent by most IO-Link devices *)
        raw := SHL(b0, 24) OR SHL(b1, 16) OR SHL(b2, 8) OR b3;
        value := DWORD_AS_REAL(raw);
    END_IF;
    ok := PI_SET_OUT_BYTE(8, 16#55);
END_PROGRAM

```

14.23 Run-time behaviour and errors

The compiled program runs on a virtual machine inside the scan. Its behaviour on errors follows one rule: the controller stops only for what makes the program's own state untrustworthy.

Condition	Class	What happens
Integer division or <code>MOD</code> by zero	Minor	Result 0; counted and reported with its line; the scan continues
Real division by zero	Minor	$\pm\text{INF}$ or NaN; counted and reported; the scan continues
Integer overflow	None	The value wraps when stored; nothing is reported
Array index out of range	Major	The scan stops; the fault names the line, the index and the bounds
<code>SQRT</code> of a negative number, <code>LN</code> / <code>LOG</code> of a non-positive number	Major	The scan stops
<code>MUX</code> selector out of range	Major	The scan stops
Direct address in an expression outside the process image	Major	The scan stops (a located variable outside it is refused at download instead)
Instruction budget exceeded (200 000 instructions per scan)	Major	The scan stops: "the program has an unbounded loop"
Call depth over 32 (for example unbounded recursion)	Major	The scan stops
Expression stack over 256 entries, call stack exhausted	Major	The scan stops

Minor faults appear in the Messages panel with their source line, and their count is kept in the controller status; only the first minor fault of each scan is recorded by message. A major fault puts the controller into FAULT with the

message; outputs follow the output fault policy. Chapter 21 describes the run states, fault reset and output behaviour; Chapter 25 the diagnostics.

WARNING: A NaN or INF from a real division by zero continues into later arithmetic, comparisons and outputs. Treat a non-zero minor-fault count as a defect to fix, and guard divisors in code that drives outputs.

14.24 Compiler diagnostics

The compiler reports every independent error in one pass, up to 64 per build, each with a line and column. Errors stop the build; warnings do not. The most common:

Message (abridged)	Meaning
'x' is not declared	Misspelled or undeclared name; also a program variable used from another POU.
'x' is already declared on line n	Two declarations of one name in the same POU.
'x' shadows the global declared on line n (warning)	See Section 14.7.1.
unknown type 'X'	Typo, a type used before its TYPE is declared, or a block that is disabled or missing.
cannot assign BOOL to INT	No implicit BOOL/number conversion (Section 14.10).
assigning REAL to INT truncates the fractional part (warning)	Use REAL_TO_INT or TRUNC to say what you mean.
mixed integer and real arithmetic (warning)	The operation is done in floating point.
this expression has no effect - did you mean ':= ' ?	A statement that is neither an assignment nor a call; also = written for := , an unsupported construct (JMP , a method called from outside, an indexed instance call).
arguments to 't' must be named	Function-block calls need name := value .
F takes 2 arguments, got 1	Function and method calls pass every input, positionally.
initial values must be literal constants	Section 14.7.4.
expected END_VAR before ...	A declaration block was not closed.
CASE selector values must be integer constants	Section 14.11.2.

14.25 Limits

Item	Limit
Significant characters in a name	63
Names in one declaration line	16

Item	Limit
POUs in one download unit	256
Members per structure or function block	128
Parameters per function, function block or method	64
Array dimensions	3
Array size	8 MiB
STRING length	254 characters (default 80)
Enumerators in the unit	256
Labels in one CASE arm	16
CASE arms handled correctly	64
Call depth at run time	32
Expression stack	256 entries
Instructions per scan	200 000
Diagnostics per build	64

Appendix D lists all limits of the system.

14.26 Differences from other IEC 61131-3 implementations

Engineers coming from other IEC 61131-3 environments will find the following differences. Where there is a way to get the same result, it is given.

Feature	Zeus	Alternative
STRING initial value	Not supported	Assign in the body
String functions (LEN , CONCAT , MID , ...)	Not supported	-
STRING as FUNCTION/METHOD parameter or return value	Not supported	Function block with VAR_IN_OUT
STRING argument in a function-block call	Not supported	blk.s := 'x'; blk();
WSTRING , CHAR , DATE , TOD , DT , LTIME	Not supported	TIME_NOW() for intervals
UNION	Not supported	-
POINTER TO , REFERENCE TO , ADR , SIZEOF , ^ dereference	Not supported	VAR_IN_OUT
Sub-range types INT(0..100)	Not supported	LIMIT
Enumeration with a base type (A, B) INT	Not supported	Enumerations are DINT
Same enumerator name in two enumerations	Not supported: one flat namespace	Prefix the names

Feature	Zeus	Alternative
Qualified enumerator as a CASE label (<code>E.Run:</code>)	Not supported	<code>Run:</code>
Array and structure initial values (<code>[1, 2, 3]</code> , (<code>x := 1</code>))	Not supported	Member defaults in the TYPE; fill arrays in code
Constants as array bounds, CASE labels or initial values	Not supported	Literals
Variable <code>BY</code> step in FOR	Not supported	<code>WHILE</code> loop
Conversions named after types (<code>INT(x)</code> , <code>TO_INT</code> , <code>INT_TO_DINT</code> , ...)	Not supported	Implicit conversion, and the functions in Section 14.10
Named arguments to a FUNCTION or METHOD	Accepted but bound by position	Write arguments in declaration order
Positional arguments to a FUNCTION_BLOCK	Not supported	Named arguments
Calling a method from outside the block (<code>inst.M()</code>)	Not supported	Command input; call the method in the block body
<code>THIS^</code> , <code>SUPER^</code> , <code>EXTENDS</code> , <code>IMPLEMENTS</code> , <code>INTERFACE</code> , <code>PROPERTY</code>	Not supported	Composition: a block inside a block
Calling an element of an array of FB instances (<code>a[i](...)</code>)	Not supported	Named instances
<code>VAR_IN_OUT</code> on a FUNCTION or METHOD	Not supported	Function block
<code>VAR_OUTPUT</code> of a FUNCTION or METHOD	Not readable	Return value, or a function block
<code>VAR_EXTERNAL</code> referring to an existing global	Not needed; redeclaring is an error	Use the global directly
<code>VAR_GLOBAL</code> inside a POU	Not supported	<code>globals.st</code> or a list under <code>gv1/</code>
Qualified global access (<code>List.var</code>)	Not supported: one namespace	-
<code>VAR_TEMP</code> reset each call in a PROGRAM / FUNCTION_BLOCK	Not reset	Assign before use
<code>VAR_RETAIN</code> inside a FUNCTION_BLOCK	No effect	Declare the instance RETAIN
Initial values of FUNCTION / METHOD locals	Not applied	Assign at the top of the body
<code>JMP</code> and labels	Not supported	Structured statements
<code>AND_THEN</code> , <code>OR_ELSE</code> , <code>&</code>	Not supported	Nested IF (both operands of AND are always evaluated)
Real literal without a leading digit (<code>.5</code>)	Not supported	<code>0.5</code>
Typed literal types (<code>BOOL#1</code>)	Prefix ignored; the value is an integer	<code>TRUE</code>
Bit access with a variable bit number	Not supported	<code>SHR</code> and <code>AND</code>

Feature	Zeus	Alternative
Integer overflow	Wraps silently on store; intermediate results are 64-bit	Range-check with <code>LIMIT</code>
Pragmas <code>{...}</code>	Accepted and ignored	-
Access specifiers	Accepted and ignored	-

15. Data types (DUT)

This chapter covers user-defined data types - structures, enumerations and aliases - and the **Data Types (DUT)** objects that hold them in a project. Use it when you want one name for a record of related values (a motor, a valve, a tank, a recipe), a named set of states instead of magic numbers, or a type name that says what a value means. The language rules for `TYPE` declarations are in Chapter 14 (Sections 14.5, 14.16 to 14.18); this chapter is about working with them in the IDE and about the design patterns that make them pay off on a real machine.

15.1 What a DUT is

A DUT (data unit type) is one file, `dut/<Name>.st`, holding one `TYPE ... END_TYPE` declaration. The file name and the type name are the same: `dut/Pump_T.st` declares `TYPE Pump_T`.

Kind	What it declares	Typical use
Structure (<code>STRUCT</code>)	A record of named members of any type	One motor, one valve, one loop, one recipe
Enumeration	A set of named integer constants	Operating modes, sequence states, fault codes
Alias	A new name for an existing type	<code>T_Percent : REAL</code> , <code>T_Recipe : ARRAY[1..8] OF REAL</code>

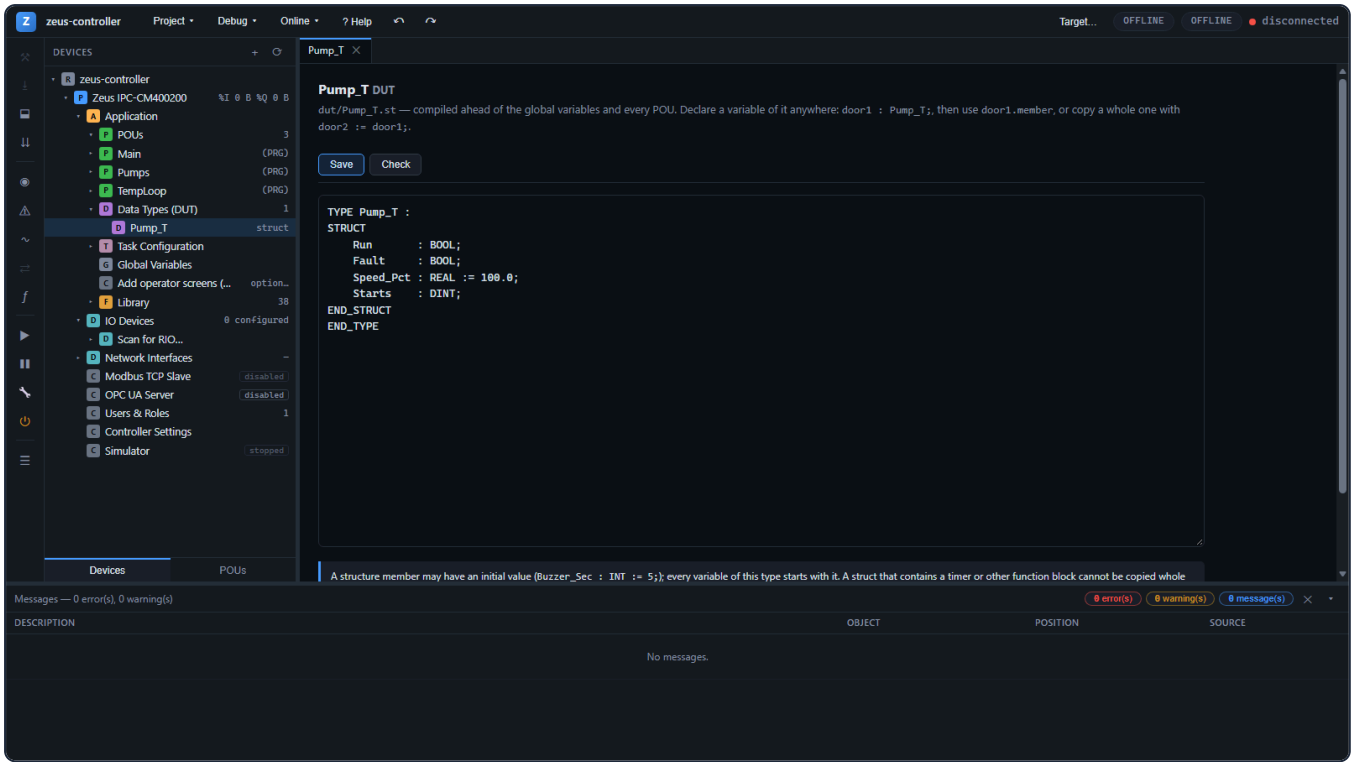
A DUT declares a **type**, not a variable. Nothing exists in the controller until you declare a variable of that type in `globals.st`, a global variable list or a POU.

DUTs are compiled **ahead of everything else** in the project - before `globals.st`, the IO aliases, the further global variable lists, the library and every POU (Section 15.4). That is why a structure declared as a DUT can be used anywhere, including in the global variables.

Note: A `TYPE` block written inside a POU file still compiles, but it is linked where that POU is linked - after `globals.st` and the library - so a global cannot be declared with it, and it exists only in the builds that include that POU. Put shared types in DUTs.

15.2 The Data Types (DUT) node

The **Data Types (DUT)** node sits under **Application**, between **POUs** and **Task Configuration**. It is always shown, even in a project with no DUTs, because "where do I put a `STRUCT`" is the question it answers.



The DUT editor with the Pump_T structure open. The tree shows Data Types (DUT) with one child, Pump_T, labelled struct.

Tree element	Shows	Actions
Data Types (DUT)	The number of DUTs on the right	Double-click (when empty) or right-click → New DUT...
Each DUT	Its name, and its kind on the right: <code>struct</code> , <code>enum</code> or <code>alias</code>	Double-click → open; right-click → Open , Rename... , Delete

The kind label is read from the file: a file containing `STRUCT` is shown as `struct` ; one whose type is a parenthesised list is `enum` ; anything else is `alias` .

15.2.1 Creating a DUT

To create a DUT:

1. Right-click **Data Types (DUT)** and choose **New DUT....** (The same command is on the **Application** right-click menu as **New DUT (data type)...**; double-clicking an empty **Data Types (DUT)** node opens it too.)
2. In the **New DUT** dialog, type the **Name**. It must start with a letter or underscore and contain only letters, digits and underscores, up to 64 characters.
3. Choose the **Type: Structure (STRUCT), Enumeration** or **Alias (TYPE X : INT)**.
4. Click **Create**.

The IDE writes `dut/<Name>.st` from a template, adds it to the tree, opens it in a DUT tab and reports "Created dut/<Name>.st".

A name already used by another DUT **or by a POU** is refused ("X' is already the name of a DUT or POU"): types and POUs share one name space, and a clash would otherwise surface as a compile error somewhere far from its cause.

The templates are:

Structure (STRUCT)	Enumeration	Alias (TYPE X : INT)
<pre> TYPE MyStruct : STRUCT END_STRUCT END_TYPE </pre>	<pre> TYPE MyStruct : (Idle := 0, Running, Fault); END_TYPE </pre>	<pre> TYPE MyStruct : INT; END_TYPE </pre>

Tip: Adopt a naming convention and keep to it. This manual uses `_T` for structures (`Motor_T`), `E_` for enumerations (`E_Mode`) and `T_` for aliases (`T_Percent`). The compiler does not care; the next engineer does.

15.2.2 The DUT tab

A DUT opens in its own tab: a heading with the type name and the word **DUT**, the file path, a one-line reminder of how to use the type, and a plain text editor.

Control	What it does
Save	Writes the file. The status text reads "saved". Ctrl+S and Save project also save an open DUT tab.
Check	Saves, then compiles the DUT together with the rest of the project and an empty throwaway program. The status reads "saved — compiles" or "saved — has errors", and errors appear in the Messages pane; click a message to jump to its line in the DUT.
Editor	Plain text with multi-step Undo/Redo (Ctrl+Z / Ctrl+Y). A modified DUT marks its tab as unsaved.

Two notes are shown under the editor:

- A structure member may have an initial value (`Buzzer_Sec : INT := 5;`); every variable of the type starts with it.
- Changing a structure that is used in a `RETAIN` variable changes the retain layout: the controller refuses the old saved values and those variables start from their initial values (Section 15.11).

Note: **Check** compiles the DUT in the context of the whole project, so an error it reports may be in a DUT this one uses. The message names the file.

15.3 Structures

15.3.1 Members and initial values

```

TYPE Pump_T :
STRUCT
    Run      : BOOL;
    Fault    : BOOL;
    Speed_Pct : REAL := 100.0;
    Starts   : DINT;
END_STRUCT
END_TYPE

```

- Up to **128 members** per structure. Member names are unique within the structure; the same member name may appear in other structures.
- A member may be of any elementary type, `STRING[n]` , an array, an enumeration, an alias or another structure.
- A member may have an initial value, written as a **literal constant** (`100.0` , `T#3s` , `TRUE` , `16#FF` , an enumerator). Every variable of the type, and every array element of the type, starts with it; members without one start at zero / FALSE / empty.
- Several members of one type may share a line: `Lo, Hi : REAL; .`

Not supported, each with the message you will see:

You write	Result
<code>Name : STRING[20] := 'Pump 1';</code> (string initial value)	"initial values must be literal constants". Assign the text in the program, once, or from the HMI.
<code>p : Pump_T := (Run := TRUE, Speed_Pct := 50.0);</code> (structure initialiser on a variable)	"initial values must be literal constants". Put the defaults on the members in the DUT instead.
<code>UNION ... END_UNION</code>	A syntax error. Use two structures, or bit access (Section 14.19).
<code>STRUCT EXTENDS Base_T</code>	Not supported. Nest the base structure as a member.

15.3.2 Nesting and arrays

A structure may contain another structure and arrays, and arrays may hold structures:

```

TYPE Range_T :
STRUCT
    Lo : REAL := 0.0;
    Hi : REAL := 100.0;
END_STRUCT
END_TYPE

TYPE Analog_T :
STRUCT
    Raw      : INT;
    Value    : REAL;
    Range    : Range_T;           (* a structure inside a structure *)
    Alarm_Hi : REAL := 90.0;
    History  : ARRAY[1..10] OF REAL; (* an array inside a structure *)
END_STRUCT
END_TYPE

TYPE T_Percent : REAL; END_TYPE
TYPE T_Recipe  : ARRAY[1..8] OF REAL; END_TYPE

PROGRAM Tanks
VAR
    Level      : Analog_T;
    Tanks      : ARRAY[1..4] OF Analog_T;
    Fill_Pct   : T_Percent;
    Recipe_A   : T_Recipe;
    Recipe_Now : T_Recipe;
    k          : INT;
END_VAR
    Level.Range.Hi := 250.0;           (* nested member *)
    Level.Value := Level.Range.Lo
                + INT_TO_REAL(Level.Raw) / 27648.0 * (Level.Range.Hi - Level.Range.Lo);
    FOR k := 10 TO 2 BY -1 DO
        Level.History[k] := Level.History[k - 1]; (* array member *)
    END_FOR;
    Level.History[1] := Level.Value;

    Tanks[2].Range := Level.Range;      (* copy a nested structure *)
    Fill_Pct := Level.Value / Level.Range.Hi * 100.0;
    Recipe_Now := Recipe_A;             (* copy a whole array *)
END_PROGRAM

```

In a project, `Range_T`, `Analog_T`, `T_Percent` and `T_Recipe` would be four DUT files and `Tanks` a POU. Members are reached with `.`, array elements with `[ ]`, in any combination: `Tanks[2].Range.Hi`, `Level.History[1]`.

15.3.3 Whole-structure copy

A structure can be assigned from another variable of the **same type** in one statement: every member, including nested structures and arrays, is copied.

```
Duty := P[i];           (* copies Run, Fault, Speed_Pct, Starts *)
```

The same applies to whole arrays of the same type and bounds (`Recipe_Now := Recipe_A;`). The rule is strict:

- Two structure types with identical members are **different types**: "cannot assign A_T to B_T - a whole struct or array can only be copied from one of the same type".

- Two arrays with different bounds are different types, even with the same number of elements (`ARRAY[1..3] OF INT` and `ARRAY[0..2] OF INT`).

Whole-structure copy is how you take a snapshot for an event log, hand a recipe from the "edit" buffer to the "active" one in a single scan, or restore defaults from a constant record.

15.3.4 Passing structures to function blocks and functions

A structure can be passed to a function block or a function in three ways:

Declared in the called POU as	Passes	Use it when
<code>VAR_INPUT m : Motor_T; (FB or FUNCTION)</code>	A copy , taken at the call	The POU only reads the record
<code>VAR_IN_OUT m : Motor_T; (FB only)</code>	A reference to the caller's variable	The POU reads and writes the record - the usual case for "one FB per device"
<code>VAR_OUTPUT o : Motor_T; (FB)</code>	The caller copies it out (<code>x := fb.o; </code>)	The FB produces a record

A FUNCTION cannot **return** a structure ("a FUNCTION must return an elementary type"). Return a BOOL or a code and use a function-block output for a record.

Note: A copy of a large structure costs execution time on every call. For a record the POU also writes, `VAR_IN_OUT` is both faster and correct; for a record it only reads, `VAR_INPUT` protects the caller from accidental writes.

15.3.5 Structures containing function blocks

A structure member may be declared as a function block type (`Tmr : TON;`). The declaration compiles and its outputs can be read (`a.Tmr.Q`), but in this version:

- A function block **inside a structure cannot be called**: `a.Tmr(IN := a.Run, PT := T#2s);` is rejected ("this expression has no effect - did you mean '=' ?"). An uncalled timer never runs.
- A structure that contains a function block **cannot be copied whole**: "cannot copy a whole `Axis_T` - it contains a function block, and a copy would share its state; copy the members instead". A function block instance has internal state (a timer's start time, an edge detector's memory); copying the bytes would give two records the same timer mid-count, which is wrong, so the compiler refuses.
- For the same reason it cannot be passed by value to a `VAR_INPUT` .

The pattern that works: keep **data** in the structure and **behaviour** - timers, counters, edge detectors - in a function block that processes the structure, with one FB instance per record. Section 15.6 shows this in full.

15.4 How DUTs are linked

When the IDE builds or downloads, it hands the compiler one unit assembled in this order (Chapter 14, Section 14.1):

1. every DUT (`dut/*.st`), in **dependency order**;
2. `globals.st` , then the generated IO alias block;
3. the further global variable lists (`gv1/*.st` , name order);

4. the library (`lib/*.st` , name order);
5. the project's function blocks and functions;
6. the program(s) being built or downloaded.

Dependency order. A type used inside another type must be declared first; the compiler does not look ahead for types. The IDE therefore sorts the DUT files so that a DUT comes after every DUT it mentions: `Analog_T` (which uses `Range_T`) is linked after `Range_T` whatever their names. DUTs that do not depend on each other keep alphabetical order. Two DUTs that use each other (a cycle) cannot be ordered; they keep name order and the compiler reports the error.

Important: The ordering is done only for DUT files. If you put several `TYPE` declarations in **one** file, or declare a type in a POU or library file, declare each type before the first type that uses it. A use before the declaration produces a misleading message on the later `TYPE` line: "type 'B_T' is already defined". It means "used before it was declared".

Function blocks and functions have no such restriction: a structure member, a global or a POU may use a function block type declared later in the unit.

15.5 Enumerations

15.5.1 Declaring an enumeration

```
TYPE E_MotorState :
(
    MS_Stopped := 0,
    MS_Starting := 1,
    MS_Running := 2,
    MS_Stopping := 3,
    MS_Faulted := 9
);
END_TYPE
```

- Enumerators are numbered 0, 1, 2 ... in order unless given `:= value`; numbering continues from an explicit value.
- A variable of an enumeration type is stored as a `DINT` (4 bytes) and shows its numeric value online.
- A base type after the list (`(A, B) INT`) is not supported ("expected a type name in the TYPE block").

Giving **explicit values** is worth the typing whenever the number leaves the controller - to an HMI, to OPC UA, to a SCADA alarm text table or into a batch record. Explicit values survive somebody inserting a new state in the middle of the list; implicit ones silently renumber every state after it.

15.5.2 Using enumerators

An enumerator may be written bare (`MS_Running`) or qualified with its type (`E_MotorState.MS_Running`), in expressions and in initial values:

```
State : E_MotorState := E_MotorState.MS_Stopped;
IF State = MS_Faulted THEN ...
```

In a `CASE` label only the **bare** form is accepted; `E_MotorState.MS_Running:` is a syntax error.

15.5.3 One flat name space

All enumerators in the whole unit - your DUTs, the library, every POU - share **one** name space:

- The same enumerator name cannot appear in two enumerations. `TYPE E_Valve : (Closed, Open, Off);` next to `TYPE E_Mode : (Off, Auto, Manual);` fails with "enumerator 'Off' is already defined, or too many enumerators".
- The qualified form `E_Mode.Off` does **not** create a separate name; it is accepted for readability and for pasted code, and refers to the same constant as bare `Off`.
- Enumeration types are not checked against each other: assigning an enumerator of `E_Valve` to a variable of type `E_Mode`, or an integer to an enumeration, compiles. Values are not range-checked.
- A variable with the same name as an enumerator hides the enumerator in that POU. `Run : BOOL;` next to an enumerator `Run` makes `e := Run;` a type error. Use the qualified form, or better, avoid the clash.
- At most **256 enumerators** in the unit, library included.

Practical rule: prefix every enumerator with a short tag of its type - `MS_` for motor states, `VS_` for valve states, `M_` for modes. It costs a few characters and makes clashes impossible.

15.6 Worked example: a motor record, an array of motors and one FB per motor

This example controls three conveyor motors. Each motor's commands, feedback, state and counters live in one `Motor_T` record; the three records are a global array so an operator screen, OPC UA or the EtherNet/IP tag server can read and write them by name; one function block holds the logic and runs once per motor. It uses every technique in this chapter.

DUT `E_MotorState` - the states (Section 15.5.1, shown above).

DUT `Motor_T`:

```

TYPE Motor_T :
STRUCT
  (* Commands - written by the operator screen or the sequence *)
  Cmd_Start   : BOOL;
  Cmd_Stop    : BOOL;
  Cmd_Reset   : BOOL;
  (* Field signals - copied in by the program every scan *)
  Fb_Running  : BOOL;          (* auxiliary contact of the contactor *)
  Overload    : BOOL;          (* TRUE = thermal trip *)
  (* Results *)
  Out_Run     : BOOL;          (* to the contactor coil *)
  State       : E_MotorState;
  Starts      : DINT;
  Run_Seconds : DINT;
  (* Settings - each motor starts with these *)
  Fb_Timeout  : TIME := T#3s;
  Enabled     : BOOL := TRUE;
END_STRUCT
END_TYPE

```

Global Variables (`globals.st`):


```
VAR_GLOBAL
    Motors : ARRAY[1..3] OF Motor_T;    (* the screen reads and writes these *)
END_VAR
```

Function block `FB_MotorCtl` and **function** `MotorAvailable` (two POU's):

```

FUNCTION_BLOCK FB_MotorCtl
VAR_IN_OUT
    M : Motor_T;                                (* the motor this instance looks after *)
END_VAR
VAR
    FbTimer : TON;                                (* start command without feedback *)
    SecPulse : TON;
    StartEdge : R_TRIG;
END_VAR
    StartEdge(CLK := M.Cmd_Start);

    CASE M.State OF
    MS_Stopped:
        M.Out_Run := FALSE;
        IF StartEdge.Q AND M.Enabled AND NOT M.Overload THEN
            M.State := MS_Starting;
            M.Starts := M.Starts + 1;
        END_IF;
    MS_Starting:
        M.Out_Run := TRUE;
        IF M.Fb_Running THEN M.State := MS_Running; END_IF;
    MS_Running:
        M.Out_Run := TRUE;
        IF M.Cmd_Stop THEN M.State := MS_Stopping; END_IF;
    MS_Stopping:
        M.Out_Run := FALSE;
        IF NOT M.Fb_Running THEN M.State := MS_Stopped; END_IF;
    MS_Faulted:
        M.Out_Run := FALSE;
        IF M.Cmd_Reset AND NOT M.Overload THEN M.State := MS_Stopped; END_IF;
    END_CASE;

    (* Faults that apply in any state *)
    FbTimer(IN := M.Out_Run AND NOT M.Fb_Running, PT := M.Fb_Timeout);
    IF (FbTimer.Q OR M.Overload) AND M.State <> MS_Faulted THEN
        M.State := MS_Faulted;
    END_IF;

    (* Running hours, in seconds *)
    SecPulse(IN := M.Fb_Running AND NOT SecPulse.Q, PT := T#1s);
    IF SecPulse.Q THEN M.Run_Seconds := M.Run_Seconds + 1; END_IF;

    (* Commands are pulses from the screen: consume them *)
    M.Cmd_Stop := FALSE;
    M.Cmd_Reset := FALSE;
END_FUNCTION_BLOCK

FUNCTION MotorAvailable : BOOL
VAR_INPUT
    M : Motor_T;                                (* passed by value: a copy *)
END_VAR
    MotorAvailable := M.Enabled AND NOT M.Overload
                    AND M.State <> E_MotorState.MS_Faulted;
END_FUNCTION

```

Program Conveyors :

```

PROGRAM Conveyors
VAR
    (* Field IO *)
    K1_Aux    AT %IX2.0 : BOOL;
    K2_Aux    AT %IX2.1 : BOOL;
    K3_Aux    AT %IX2.2 : BOOL;
    F1_Trip   AT %IX2.3 : BOOL;
    F2_Trip   AT %IX2.4 : BOOL;
    F3_Trip   AT %IX2.5 : BOOL;
    K1        AT %QX2.0 : BOOL;
    K2        AT %QX2.1 : BOOL;
    K3        AT %QX2.2 : BOOL;

    (* One controller instance per motor *)
    Ctl1, Ctl2, Ctl3 : FB_MotorCtl;

    Snapshot    : Motor_T;           (* a copy for the event log *)
    Available   : INT;
    i           : INT;
END_VAR

(* Inputs into the structures *)
Motors[1].Fb_Running := K1_Aux; Motors[1].Overload := F1_Trip;
Motors[2].Fb_Running := K2_Aux; Motors[2].Overload := F2_Trip;
Motors[3].Fb_Running := K3_Aux; Motors[3].Overload := F3_Trip;

(* The same logic, three times, each on its own record *)
Ctl1(M := Motors[1]);
Ctl2(M := Motors[2]);
Ctl3(M := Motors[3]);

(* Structures out to the outputs *)
K1 := Motors[1].Out_Run;
K2 := Motors[2].Out_Run;
K3 := Motors[3].Out_Run;

(* A loop over the array, and a FUNCTION taking a whole structure *)
Available := 0;
FOR i := 1 TO 3 DO
    IF MotorAvailable(M := Motors[i]) THEN
        Available := Available + 1;
    END_IF;
END_FOR;

(* Whole-structure copy: every member in one assignment *)
IF Motors[2].State = MS_Faulted THEN
    Snapshot := Motors[2];
END_IF;
END_PROGRAM

```

Points to notice:

- **One FB instance per record.** Ctl1, Ctl2 and Ctl3 each have their own FbTimer, SecPulse and StartEdge. That is why the timers are in the FB and not in Motor_T (Section 15.3.5).
- **VAR_IN_OUT** makes M the caller's Motors[n] itself: the FB's writes to M.State, M.Out_Run and M.Starts land directly in the global array, where the screen sees them. Every call must bind it.
- **The instances are called explicitly**, not in a loop. An array of function block instances cannot be called by index in this version (Ctl[i](...) is rejected), so a machine with many identical devices has one call line per

device. A loop over the **data** is fine, as the `MotorAvailable` loop shows.

- **Commands are consumed.** The screen writes `Cmd_Stop := TRUE`; the FB acts on it and clears it in the same scan, so a command cannot be acted on twice. `Cmd_Start` is edge-detected instead, which suits a push-button-style HMI that holds the bit while pressed.
- **The IO stays in the program.** `Motor_T` has no `AT` addresses: the record describes a motor, not where it is wired. Moving motor 2 to another card changes two lines in `Conveyors`, not the type.

WARNING: A command member such as `Cmd_Start` can be written by the HMI, OPC UA, the EtherNet/IP tag server and the IDE. Decide which of them may start a motor and restrict the others (Chapters 29, 30 and 34). A structure makes every member writable by name.

15.7 Aliases

An alias gives a new name to an existing type:

```
TYPE T_Percent : REAL; END_TYPE
TYPE T_Recipe : ARRAY[1..8] OF REAL; END_TYPE
```

- An alias **is** its base type. A `T_Percent` is a `REAL` and mixes with any `REAL`; the compiler does not check units.
- An alias of an array or a `STRING[n]` gives one name to the bounds, so two variables declared with it can be copied whole (Section 15.3.3) and a change of size is made in one place.
- The value of an alias is documentation: `Fill : T_Percent;` says what the number means where `Fill : REAL;` does not.

15.8 Pasting DUT text from other IEC 61131-3 environments

A DUT copied out of another IEC 61131-3 project can usually be pasted into a DUT tab as it is:

- **Pragmas** in braces - `{attribute 'qualified_only'}`, `{attribute 'strict'}` and the like - are accepted and **ignored**. In particular, `qualified_only` does not make the enumerators qualified-only: they remain one flat name space (Section 15.5.3), so bare names still work and clashes are still clashes.
- `(* *)` and `//` comments are kept.
- Several types in one `TYPE` block are accepted; split them into one DUT per type so the IDE can order them.

This, pasted as is, compiles:

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_Mode :
(
    Off      := 0,
    Manual   := 1,
    Auto     := 2
);
END_TYPE

PROGRAM ModeDemo
VAR
    Mode      : E_Mode := E_Mode.Off;
    Auto_Sel  AT %IX3.0 : BOOL;
    Man_Sel   AT %IX3.1 : BOOL;
    Lamp      : INT;
END_VAR
    IF Auto_Sel THEN
        Mode := E_Mode.Auto;
    ELSIF Man_Sel THEN
        Mode := E_Mode.Manual;
    ELSE
        Mode := E_Mode.Off;
    END_IF;

    CASE Mode OF
        Off:    Lamp := 0;
        Manual: Lamp := 1;
        Auto:   Lamp := 2;
    END_CASE;
END_PROGRAM
```

What to check after pasting:

In the pasted text	What to do
The same enumerator name in two enumerations	Rename one set (prefix them).
E_X.Value in a CASE label	Write the bare enumerator.
An enumeration with a base type, (A, B) INT	Remove the base type.
A STRING member with an initial value	Remove the initial value; assign it in the program.
UNION, EXTENDS, POINTER TO, REFERENCE TO	Not supported; restructure.
A member of function block type that the original program calls	Move the FB into a function block that processes the structure (Section 15.3.5).

Press **Check** after pasting; every problem is reported with its line in the DUT.

15.9 Renaming and deleting a DUT

To rename a DUT:

1. Right-click it under **Data Types (DUT)** and choose **Rename...**
2. Type the new name and confirm.

The IDE first writes a backup of the project (`backups/<time>-rename-dut-<old name>`), then renames the file **and the TYPE line inside it**. Uses of the old name elsewhere are **not** rewritten. Instead, the **Messages** pane lists every file that still mentions the old name ("Renamed X to Y. Still using the old name: globals.st, Conveyors.st — change them to Y."). Change those declarations, then build.

Note: The type name is not part of the retain layout signature (Section 15.11). Renaming a DUT that a retained variable uses, and updating that declaration to the new name, keeps the retained values - provided the members themselves are unchanged.

To delete a DUT:

1. Right-click it and choose **Delete**.
2. Confirm "Delete the data type ...? Any variable declared with it stops compiling. A backup of the project is written first."

The backup is written to `backups/<time>-delete-dut-<name>`. Recover a deleted DUT by copying its file back from there into `dut/`.

15.10 Viewing structures online

When the controller is connected and running the program, structured variables appear in the online variable tables (the **Global Variables** tab's live values and each POU's **Variables** tab; Chapter 22) as a tree:

- A structure variable has a **(+)**; opening it lists its members, each with type, live value, prepared value and address columns.
- An array shows one child per element, labelled `[1]`, `[2]` ...; an array of structures nests: `Motors` → `[2]` → `State`.
- Every member can be written, prepared, and (if it is `AT`-addressed) forced individually, exactly like a plain variable. Its full name is the path: `Motors[2].Cmd_Start`.
- An enumeration member shows its **number**. Keep the enumeration DUT open beside the table, or give the states explicit values you know by heart.
- Large arrays are listed up to their first **100** elements in the tree; any element can still be read and written by its full name (watch list, trace, OPC UA).

Hovering over a structured variable in an editor shows its type and where it is declared (Chapter 13).

15.11 Retain

A structure, an array of structures or an enumeration can be retained like any other variable (`VAR_GLOBAL RETAIN`, or `VAR RETAIN` in a PROGRAM; Chapter 14, Section 14.7 and Chapter 33).

The retain store is protected by a **layout signature** hashed over every retained symbol's name, type, position and - for a structure - its full member layout: member order, types, array bounds and string lengths. So:

- Adding, removing, reordering or retyping a member of a structure used in a retained variable changes the signature. At the next start the controller **refuses** to restore the old image, records why, and every retained variable starts from its initial values.
- This is deliberate. The alternative is restoring bytes into the wrong members: a batch counter becomes a setpoint, silently.

Important: Finish the structures that are retained **before commissioning**. When a retained structure must change on a running plant, record the retained values first (Save writes `data/saved.data.csv` ; Chapter 5), download, then restore the values that still apply (Chapter 33).

15.12 Limits

Item	Limit
Members per structure	128
Enumerators in one unit, all enumerations together	256
Array dimensions	3
Size of one array (including an array member)	8 MB
Array elements shown in the online tree	first 100 (all are addressable by name)
Name of a DUT / type	64 characters; letters, digits, underscore; must not start with a digit
<code>STRING</code> length	254 characters (default 80)
Structure nesting	No fixed limit; a structure cannot contain itself

15.13 Summary of what is not supported

Feature	Status in this version
<code>STRING</code> initial values (on members or variables)	Not supported - assign in the program
Structure initialiser (<code>a := 1, b := 2</code>) on a variable	Not supported - use member initial values in the DUT
The same enumerator in two enumerations	Not supported - one flat name space
Enumeration base type (<code>(A, B) INT</code>)	Not supported
<code>UNION</code> , <code>EXTENDS</code> on a structure	Not supported
Calling a function block that is a structure member	Not supported - put the FB in a function block that processes the structure
A FUNCTION returning a structure	Not supported - use a function-block output
<code>STRING</code> parameters to FUNCTIONS (and to function-block inputs in a call)	Not supported - pass the text through a structure member or set the FB input as <code>inst.s := text</code> ; before the call

16. Ladder

This chapter describes the graphical Ladder Diagram (LD) editor: how to create a ladder program or function block, draw rungs with contacts, compare boxes, function-block boxes and coils, declare the variables, read the Structured Text the diagram compiles to, and watch power flow on a running controller. It is built around one example - a conveyor with start/stop, overload, speed-up and jam detection - which is shown both as the drawing and as the ST it generates. Use Ladder for interlocks, permissives and motor starters that an electrician must be able to read; use Structured Text (Chapter 14) for arithmetic, loops and state machines, and SFC (Chapter 17) for sequences.

16.1 How a ladder POU works

The controller runs only Structured Text. A ladder POU is therefore stored as an ordinary `.st` file with two parts:

1. A leading comment marker, `(*@LADDER <base64 data>@*)`, that holds the diagram - rungs, lines, elements, coils, comments and the declarations - in encoded form.
2. The Structured Text generated from that diagram: a normal `PROGRAM` (or `FUNCTION_BLOCK`) with its `VAR` block and one group of statements per rung.

```
(*@LADDER eyJ2IjoxLCJraw5kIjoiUFJHIi...@*)
(* Generated from the Ladder editor - open this POU to edit the diagram.      *)
(* Editing the text directly is fine, but the Ladder view rewrites it on save. *)
PROGRAM Conveyor
VAR
    ...
END_VAR
    (* Rung 1: ... *)
    Motor := (Start_PB OR Motor) AND NOT Stop_PB AND NOT Overload AND NOT Jam_Alarm;
    ...
END_PROGRAM
```

Because the file is valid ST, everything else in the IDE works on it unchanged: build, download, online change, upload, rename, where-used, the archive, version control. The compiler never sees the drawing; it sees the generated ST. What you see in **Show ST** is exactly what runs.

When the IDE opens a POU whose file carries the marker, it shows the Ladder editor instead of the text editor. Saving regenerates the whole file - marker, declarations and body - from the diagram.

Important: Edit a ladder POU **in the Ladder editor**, including its declarations (**Declarations...**, Section 16.8). The declarations are held inside the marker as well as in the text. A change made only in the ST text - an extra variable, a fixed typo - compiles and runs, and is then silently overwritten the next time someone opens the diagram and saves.

16.2 Creating a ladder POU

To create a ladder program:

1. Right-click **Application** (or **POUs**, or a POU folder) and choose **New POU...**
2. Enter the **Name**.

3. Under **Type**, choose **Program — Ladder**. For a ladder function block choose **Function Block — Ladder**.
4. Optionally choose a **Folder**.
5. Click **Create**.

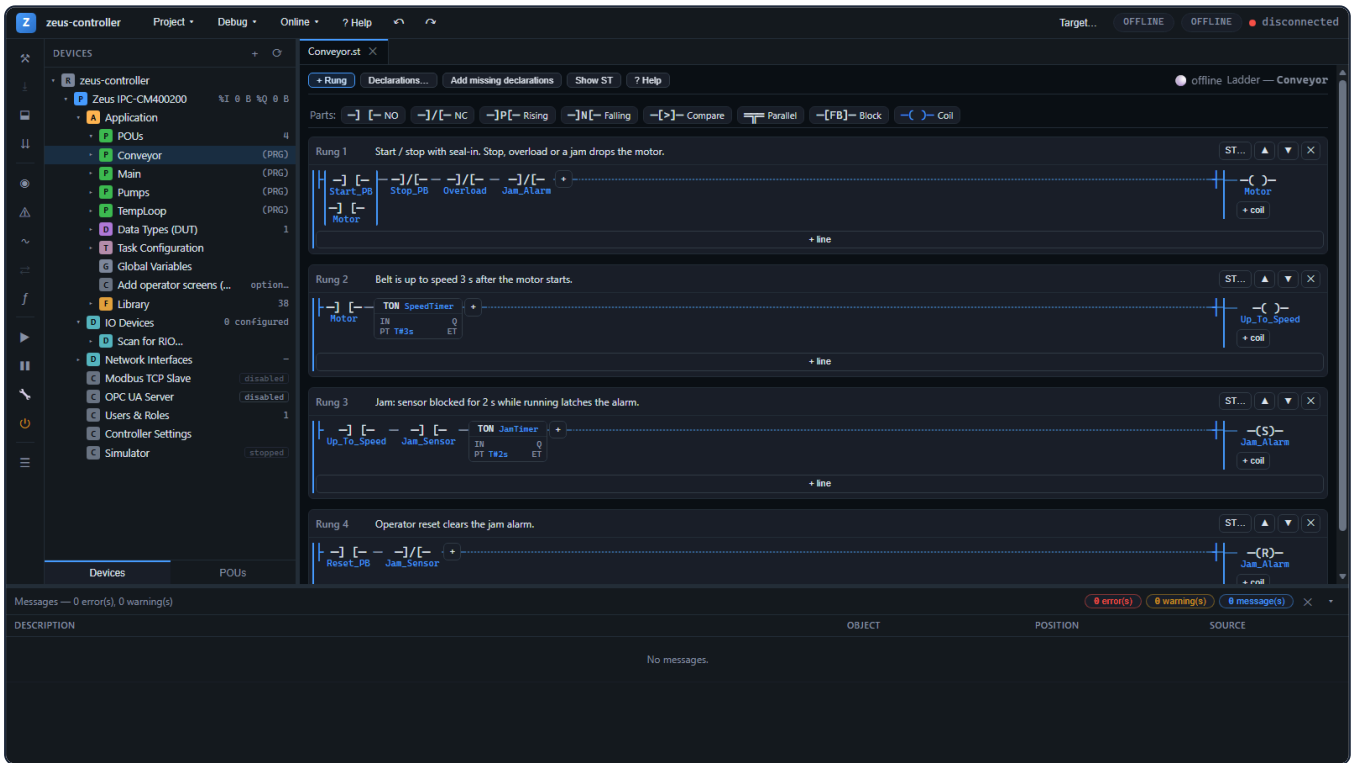
The new POU opens in the Ladder editor with one empty rung.

Type	Starting declarations	Generated header
Program — Ladder	Start, Stop, Motor : BOOL; (a placeholder you replace)	PROGRAM <Name> with one VAR block
Function Block — Ladder	VAR_INPUT IN : BOOL; END_VAR , VAR_OUTPUT Q : BOOL; END_VAR , VAR END_VAR	FUNCTION_BLOCK <Name> ; the declarations are the block's interface

A ladder function block is used like any other function block: declare an instance and call it from ST, or drop it on another ladder rung as a **Block** (Section 16.6.4). There is no ladder FUNCTION in the **New POU** dialog; write functions in ST.

A new ladder POU is a PROGRAM that nothing calls yet. Add it to the task call list (Chapter 18) so a download includes it.

16.3 The editor window



The Conveyor ladder program in the editor, offline: four rungs, the toolbar and the Parts palette.

From top to bottom:

Toolbar

Button	What it does
+ Rung	Adds an empty rung at the bottom.

Button	What it does
Declarations...	Edits the POU's declarations (Section 16.8).
Add missing declarations	Declares every name the diagram uses that is not yet declared (Section 16.8.2).
Show ST	Shows or hides the generated Structured Text of the body, below the rungs. It updates as you edit.
? Help	Opens the Ladder editor help page in a new window.
Live status (right)	<code>offline</code> , or when online <code>online</code> • <code>N symbols</code> • <code>X/Y matched</code> (Section 16.11).
Ladder — <name>	The POU being edited.

Parts palette - one chip per element type: **NO**, **NC**, **Rising**, **Falling**, **Compare**, **Parallel**, **Block**, **Coil**. Drag a chip onto a rung line to drop it at that position, or click a chip to add it to the **last** rung. Either way the element editor opens pre-set to that part, so you only pick the variable.

Rungs - each rung has a header and a body:

Rung header	Purpose
Rung n	The rung number. An ST badge marks a raw ST rung (Section 16.10).
Comment box	Free text. It becomes the comment line <code>(* Rung n: <text> *)</code> in the generated ST.
ST...	Shows and edits this rung as Structured Text (Section 16.10).
▲ / ▼	Moves the rung up or down. Rung order is execution order.
×	Deletes the rung, without confirmation.

The rung body holds one or more **lines** between the left rail (┆) and the right rail (┆), each with its coils to the right. Under the lines, **+ line** adds another line to the rung.

16.4 Rungs and lines

A **line** is a series path, read left to right: every element on it must conduct for power to reach the right rail. Elements on a line are ANDed.

A rung can have several lines. Each **coil belongs to one line**: it is energised by that line alone. When you add a coil you choose **Driven by**:

Driven by	Generated power expression
Line 1 only, Line 2 only ...	That line's series expression
All lines (OR)	The lines ORed: <code>(line1) OR (line2)</code> . Such coils are drawn on a separate row labelled any line ▶ under the lines.

Per-line coils let one rung hold related outputs that share a comment - a set on one line and a reset on the next, for example - while "All lines (OR)" gives the classic parallel rung.

Within a line, OR logic is drawn with a **Parallel** element (Section 16.6.3): a box of two or more paths, any of which conducts. That is how a seal-in contact is drawn.

Action	How
Add an element to a line	The + button at the end of the line, or drag a palette chip onto the line
Add a coil driven by a line	+ coil to the right of that line
Add a line	+ line under the lines
Delete a line	✕ at the end of the line (not shown on the first line). Its coils are deleted with it.
Move an element	Drag it to another position on the same line, onto another line, or into a path of a Parallel element
Edit an element or coil	Click it
Other element commands	Right-click it (Section 16.7)

A rung with no elements on a line conducts unconditionally (an empty line is `TRUE`). A rung with no coil is legal: it still runs its function blocks, and the ST shows `(* no coil on this rung *)`.

16.5 Execution order

Rungs execute top to bottom, once per scan, as the ST statements they generate. Within a rung, the function-block calls and edge detectors on the lines are executed first, in line order, then the coils are assigned in the order they were added.

A coil written on rung 1 and read on rung 3 gives rung 3 this scan's value. A coil read on an **earlier** rung sees the previous scan's value. This is the same rule as any IEC ladder, and the generated ST makes it explicit.

16.6 Parts

16.6.1 Contacts

Part	Symbol	Conducts when	Generated ST
NO - normally open	–] [–	the variable is TRUE	<code>Var</code>
NC - normally closed	–]/[–	the variable is FALSE	<code>NOT Var</code>
Rising - rising edge (P)	–]P[–	for one scan when the variable goes FALSE → TRUE	an <code>R_TRIG</code> call, then <code>inst.Q</code>
Falling - falling edge (N)	–]N[–	for one scan when the variable goes TRUE → FALSE	an <code>F_TRIG</code> call, then <code>inst.Q</code>

A contact takes a BOOL: a local, a global, an `AT`-addressed input, a function-block output such as `SpeedTimer.Q`, a structure member or a bit of an integer (`Alarm_Word.3`).

An edge contact compiles to an edge-detector instance with a generated name such as `_edgeb4x1k`. The name is stored with the element so it is stable across edits; **Add missing declarations** declares it (`_edgeb4x1k : R_TRIG;`). Copying and pasting an edge contact gives the copy its own detector, so two copies never share one edge memory.

Note: Physical stop buttons and safety chains are normally wired **normally closed**, so the input is TRUE when healthy. The program then reads them with an **NO** contact (conduct while healthy). In the conveyor example `Stop_PB` is a normally open button, so it is read with **NC**. Match the contact to the wiring, and write the wiring in the variable's comment.

16.6.2 Compare

The **Compare** box, `-[ a > b ]-`, conducts while the comparison is true. Operators: `>`, `<`, `>=`, `<=`, `=`, `<>`. Each side is a numeric variable, a function-block output such as `JamTimer.ET` or `Counter.CV`, or a literal (`20000`, `90.0`, `T#5s`).

The generated ST is the comparison in parentheses: `(Level_Pct > 90.0)`.

Tip: `=` on a REAL is rarely true. Compare REALs with `>=` / `<=`.

16.6.3 Parallel

The **Parallel** part inserts an OR group into a line. Dropping it opens the **Parallel branch (OR)** editor with two paths:

- Each **Path** is a series (AND) of contacts. Each cell has a kind selector (NO, NC, Rising, Falling, Compare) and its variable.
- **+ contact** adds a contact in series on that path; **X** removes one.
- **+ parallel path** adds another path; **X path** removes one (a group keeps at least one).
- **Delete branch** removes the whole group.

On **OK**, blank contacts and blank paths are dropped. A group left with one path is flattened back into ordinary contacts on the line; a group left empty is removed. A Parallel element cannot contain another Parallel element or a function block.

The group conducts when any path conducts, and sits in series with the rest of the line: `(Start_PB OR Motor) AND NOT Stop_PB`.

16.6.4 Block (function block box)

The **Block** part places a function block on a line, drawn as a box with its type on top, the instance name, its input pins down the left and output pins down the right.

The **Add element / Edit element** dialog for a block:

Field	Meaning
Standard block	A catalogue: the IEC standard blocks and the data blocks (below), or Custom / library block...
Block type	The type name: <code>TON</code> , <code>CTU</code> , <code>SCALE</code> , <code>MOTOR_DOL</code> , a project FB ...
Instance	The instance variable, e.g. <code>SpeedTimer</code> . Every box needs its own.
Input pin (power)	The input the rung power drives (<code>IN</code> for a timer, <code>CU</code> for a counter)
Output pin (power)	The output that continues the rung (<code>Q</code>)

Field	Meaning
Inputs	Other input pins, each wired to a variable or a value (<code>PT := T#3s</code>). + input pin adds one, × removes one.
Outputs	Output pins sent to variables (<code>ET → Elapsed</code>). + output pin adds one.
+ New FB / FC...	Opens New POU to create your own block without leaving the diagram

Choosing a catalogue entry fills in the type, the power pins and the default pins. Typing a **project** function block's name in **Block type** loads its pins from its declaration (`VAR_INPUT` and `VAR_IN_OUT` as inputs, `VAR_OUTPUT` as outputs). For a **library** block that is not in the catalogue (`MOTOR_DOL` , `VALVE_2POS` ...) type the pins by hand; the library dialog (Chapter 19) lists them.

Boolean (IEC) blocks - the rung power drives the input pin and the output pin continues the rung:

Block	Purpose	Power in → out	Other inputs (defaults)	Outputs offered
TON	On-delay timer	IN → Q	PT (T#5s)	ET
TOF	Off-delay timer	IN → Q	PT (T#5s)	ET
TP	Pulse timer	IN → Q	PT (T#1s)	ET
CTU	Count up	CU → Q	R (FALSE), PV (10)	CV
CTD	Count down	CD → Q	LD (FALSE), PV (10)	CV
CTUD	Count up/down	CU → QU	CD , R , LD (FALSE), PV (10)	QD , CV
R_TRIG	Rising-edge detect	CLK → Q	-	-
F_TRIG	Falling-edge detect	CLK → Q	-	-
RS	Reset-dominant latch	S → Q	R1 (FALSE)	-
SR	Set-dominant latch	S1 → Q	R (FALSE)	-

Generated ST: `SpeedTimer(IN := <power to the left>, PT := T#3s);` , outputs assigned (`Elapsed := SpeedTimer.ET;`), and the rung continues as `SpeedTimer.Q` . Blocks chain: one block's output pin powers the next block on the same line.

Data blocks (from the library, Chapter 19) - the box is **enabled** by the rung power: the block is called only while the power to its left is TRUE, its inputs are the parameters, its outputs go to variables, and the power passes through unchanged:

Block	Purpose	Parameters (defaults)	Outputs offered
SCALE	Raw counts → engineering units	raw , raw_min (0), raw_max (27648), eu_min (0.0), eu_max (100.0)	eu , under , over
PID	Closed-loop control	pv , sp , kp (1.0), ti_s (10.0), td_s (0.0), out_min (0.0), out_max (100.0)	out , err , sat , integral
ALARM_AN	HH/H/L/LL alarm	pv , hh (95), h (90), l (10), ll (5), hyst (1)	hh_alm , h_alm , l_alm , ll_alm , any
FILTER_PT1	First-order low-pass	x , tau_ms (1000), dt_ms (10)	y

Block	Purpose	Parameters (defaults)	Outputs offered
RATE_LIMIT	Slew limiter	x, up_per_s, dn_per_s (10.0), dt_ms (10)	y, limited
RAMP	Ramp to a target	target, rate_per_s (1.0), dt_ms (10)	out, ramping, at_target
HYSTERESIS	On/off with deadband	pv, on_at (80), off_at (70)	q
DEADBAND	Ignore small changes	pv, band (1.0)	out, moved
PWM	Pulse-width output	duty, period_ms (1000), dt_ms (10)	q, on_ms
TOTALIZER	Integrate a rate	rate, rate_base (60), dt_ms (10), run (TRUE), reset (FALSE)	total
DERIV	Rate of change	pv, filter_s (1.0), dt_ms (10)	rate
AVERAGE_N	Moving average	pv, n (10)	out, ready

The catalogue fills each parameter with a placeholder (raw , pv ...); replace every one with your variable or value. A parameter left empty is not passed, and the block uses its own default. dt_ms must equal the task interval (Chapter 18).

Generated ST for a data block with power to its left:

```
IF <power> THEN
  LevelScale(raw := Level_Raw, raw_min := 0, ...);
  Level_Pct := LevelScale.eu;
END_IF;
```

and without a condition (first on the line) simply the call and the assignments.

Important: A data block behind a contact is **not called** while the contact is open. A filter, ramp, PID or totaliser then stops integrating and resumes later from where it stopped. Put a data block that must run every scan first on its own rung.

16.6.5 Coils

Coil	Symbol	Generated ST
Coil - follows the line	-()-	Var := <power>;
Set (latch)	-(S)-	IF <power> THEN Var := TRUE; END_IF;
Reset (unlatch)	-(R)-	IF <power> THEN Var := FALSE; END_IF;

The **Add coil / Edit coil** dialog has **Kind**, **Driven by** (Section 16.4) and **Variable**.

- A plain coil **writes its variable every scan**, TRUE or FALSE. Writing the same variable from two plain coils (or from a coil and ST elsewhere) means the last one executed wins - the classic double-coil fault.
- Set and Reset coils write only while their rung conducts. A Set on one rung and a Reset on a later rung of the same scan: the Reset wins when both conduct, because it runs last.
- A latched variable keeps its value across STOP/RUN only if it is declared **RETAIN** (Chapter 33).

16.7 Selecting, copying and editing

Selecting. Click an element or coil to select it (a blue outline). **Ctrl+click** adds or removes one; **Shift+click** selects every element between the last one clicked and this one on the same line. **Drag a box** on empty space in the diagram to select every element and coil it touches (hold **Ctrl** to add to the selection). Click **Rung N** to select a whole rung, **Ctrl/Shift+click** further rung numbers for several, **Ctrl+A** for all rungs. Click empty space or press **Esc** to clear. **Double-click** an element or coil (or press **Enter**) to edit it.

Keys (click in the diagram first):

Key	Does
Ctrl+C / Ctrl+X / Ctrl+V	Copy / cut / paste the selection
Ctrl+D	Duplicate the selection
Del	Delete the selection
Ctrl+Z / Ctrl+Y (or ↶ ↷ in the top bar)	Undo / redo, up to 200 steps
Ctrl+B	Wrap the selected side-by-side elements in a parallel branch
Ctrl+F	Find / replace a variable
Insert	Add an empty rung below the selected one
← →	Select the previous / next element on the line
Ctrl + mouse wheel, or – 100% +	Zoom the diagram

Pasting. Whole rungs paste as new rungs below the selected rung. Elements and coils paste after the selected element on its line, or at the end of line 1 of the selected rung; coils go to the rung of the paste and are driven by that line. The clipboard is shared by every ladder POU, so rungs and elements copy from one program to another. A pasted rising/falling contact gets its own edge detector.

Find / Replace... searches every contact, compare, block (instance, inputs, outputs), coil and ST rung for a variable — as a whole name (`Motor1` , `Motor1.Q`) or part of one, ignoring case — and selects every match; **Replace all** renames them in one step that **Ctrl+Z** undoes.

Dragging a selection. With several elements selected, drag any one of them: all of them move, in order, to the drop point (coils stay with their rung).

Math / move boxes. The **Math / Move** part is a box that runs while the rung has power up to it and passes that power on: **MOV** (Dest := Source), **ADD**, **SUB**, **MUL**, **DIV**, **MOD** (Dest := A op B) and **CPT** (Dest := any ST expression). **Limit (LIM)** passes power while *low* ≤ *test* ≤ *high*.

Branches inside branches. In the branch editor, set a cell's kind to **Branch inside** to put a parallel branch within a path, up to three levels; **Ctrl+B** also wraps a selection that already contains a branch.

A rung as text. + Rung as text... (toolbar) and **Edit rung as text...** (rung menu) take one text line per rung line with the common mnemonics: `XIC` , `XIO` , `ONS / OSF` , `EQU` `NEQ` `GRT` `GEQ` `LES` `LEQ` , `LIM` , `TON` `TOF` `TP` `CTU` `CTD` (instance, preset), `MOV` , `ADD` `SUB` `MUL` `DIV` `MOD` , `CPT` dest "expression" , `BST` ... `NXB` ... `BND` for a branch (may nest), and `OTE` / `OTL` / `OTU` outputs — for example `XIC Start BST XIC Motor NXB XIC Jog BND XIO Stop OTE Motor` . **Copy rung as text** puts the selected rungs on the clipboard in the same form.

Verify and errors on rungs. **Verify** (or F7) builds the POU; every rung that holds an error or warning is outlined and gets a **X/Δ** badge whose tooltip lists the messages. Clicking a message in the **Messages** panel selects its rung.

Descriptions. The comment written after a declaration — `Start : BOOL; (* start button *)` or `// start button`, in the POU or in the global variables — is shown above the operand. **Descriptions** on the toolbar turns them on and off.

Viewing large programs. Go to... (Ctrl+G) jumps to a rung number; **Fold all** or the ▼ on a rung folds it to a one-line text summary; **Print / PDF...** opens the diagram as a light printable page (choose *Save as PDF* in the print dialog).

Cross reference. Cross reference... on an element or coil, and **Find in all programs** in Find / Replace, list every declaration, write and read of the variable in the whole project.

Online edit. While online and logged in, **Download changes (online)** sends this program as an online change — the controller keeps running — and **Discard changes** throws away the edits since the last save.

An ST program as ladder. In any Structured Text POU, **View as ladder** (in the editor's action bar) draws the program as a ladder diagram in a read-only tab, with live values and power flow while online. Every statement with an exact ladder form becomes a rung — a boolean assignment (contacts, compares, branches and a coil), `IF c THEN x := TRUE/FALSE` (set / reset coil), `IF c THEN assignments [ELSE ...]` (one rung per assignment, the ELSE ones on NOT c), a move or arithmetic (MOV, ADD... or CPT), and block calls (a timer takes its IN from the rung; any other block is an always-enabled box). Everything else — CASE, loops, ELSIF, an IF whose body changes its own condition — stays as an ST rung, so the drawing always does exactly what the program does. The banner says how many statements were drawn and how many kept as ST. The program itself is never changed; **Convert to ladder POU...** writes a NEW ladder POU (default `<name>_LD`) next to it, which runs only when you put it in the task's call list.

`x = TRUE` and `x = FALSE` in a condition are drawn as a normally-open and a normally-closed contact, `IF c THEN x := TRUE; ELSE x := FALSE; END_IF` as a single coil on `c`, and `IF c THEN x := y` as a MOV box after the condition.

Function blocks and methods. **View as ladder** works in a function block's tab and in a METHOD tab too. A block's view draws its body (its methods are drawn from their own tabs); a method's view draws that method. A block's variables live in each instance, so the banner has an **Instance** list: the live values and power flow are the chosen instance's, and a value written from the view goes to that instance. **Convert to ladder POU...** is offered only for a POU drawn whole (not for a method, or a block that has methods).

Right-click commands

On	Command	Does
Element	Edit... / Edit branch...	Opens the element (or Parallel) editor
Function block box	<type> — standard block	Says the block is built into the runtime
Function block box	Open / create <type> definition	Opens the project FB of that name, or offers to create it
Contact, compare	Set value...	Online: write or force the variable (Section 16.11)
Element	Cut, Copy, Paste after, Duplicate, Delete	On the selection when the element is part of it, otherwise on this element

On	Command	Does
Element	Toggle NO / NC (rising / falling)	Swaps the selected contacts' kind
Element	Branch around selection	As Ctrl+B
Element	Select whole rung	Selects the rung the element is on
Coil	Edit..., Set value..., Cut, Copy, Paste, Delete	As for elements
Coil	Change to () → (S) → (R)	Steps the selected coils through coil, set and reset
Rung (its number, comment bar or empty space)	Add rung above / below, Cut / Copy rung, Paste rung(s) below (or Paste into this rung for copied elements), Duplicate, Edit rung as ST..., Move up / down, Undo / Redo, Delete rung	Rung commands, on every selected rung

16.8 Declarations

16.8.1 Declarations... dialog

Declarations... opens the POU's declarations as text:

- For a **program** the dialog is **Declarations (VAR ... END_VAR)** and holds the **inside** of the single `VAR` block - one declaration per line, `AT` addresses and comments allowed.
- For a **function block** the dialog is **Interface & declarations** and holds the full blocks: `VAR_INPUT ... END_VAR`, `VAR_OUTPUT ... END_VAR`, `VAR ... END_VAR`. The inputs and outputs are the block's pins.

Click **Save** to apply; the generated ST updates immediately, and the POU is marked unsaved.

Every variable a rung uses must be declared - in this dialog, in `globals.st` or a global variable list (Chapter 19), or as an IO alias (Chapter 11). A program may use globals by name without declaring them here.

16.8.2 Add missing declarations

Add missing declarations scans the diagram and appends a declaration for every name that is not already in the POU's declarations:

- a function-block instance, with its type (`SpeedTimer : TON;` , `_edgeb4x1k : R_TRIG;`);
- every other name - contact, coil, compare operand, block output variable - as `BOOL`.

It is a convenience, not a type checker. After using it:

- Open **Declarations...** and correct every variable that is not a `BOOL` - a compare operand or a block output such as `Level_Pct` must be `REAL`, a counter value `INT`, and so on.
- Delete any line it added for a global.** The command only looks at this POU's declarations; a global you used by name is declared again locally, and the local declaration **hides the global** (the compiler warns that it shadows the global). The program would then work on its own copy while the HMI and the IDE show the global.
- Add `AT` addresses to the physical IO.

Names in raw ST rungs (Section 16.10) are not scanned; declare those by hand.

16.8.3 Picking variables

Every variable field in the element and coil dialogs has two controls: a **pick a declared variable** list and an **or type one** box. The list offers only what fits the element, grouped:

Group	Contents
This POU	The POU's own declarations of a fitting type
Global	Globals from <code>globals.st</code> (including the IO aliases declared there) of a fitting type
Block outputs	Outputs of the instances in scope: <code>SpeedTimer.Q</code> , <code>JamTimer.ET</code> , <code>PartCounter.CV</code> , and the members of project FB instances

A contact or coil lists BOOLs; a compare lists numeric variables and TIME. An empty group says why ("3 declared, none are BOOL", "none declared"). A name that is not declared yet can always be typed; **Add missing declarations** then writes it.

Note: Variables in further global variable lists (`gv1/*.st`) are not offered in the picker. Type their names; they compile normally.

16.9 Worked example: a conveyor

The conveyor has a start button, a stop button, a motor overload contact, a jam sensor at the discharge and an operator reset. Requirements:

1. Start/stop with seal-in. Stop, the overload or a latched jam alarm drops the motor.
2. The belt is "up to speed" 3 s after the motor starts.
3. A jam - the sensor blocked for 2 s while the belt is up to speed - latches the jam alarm (which stops the motor through rung 1).
4. The operator reset clears the jam alarm, but only once the sensor is clear.

Declarations (Declarations...):

```

Start_PB    AT %IX0.0 : BOOL;    (* green push button *)
Stop_PB     AT %IX0.1 : BOOL;    (* red, normally open *)
Overload    AT %IX0.2 : BOOL;    (* motor thermal trip *)
Jam_Sensor  AT %IX0.3 : BOOL;
Motor       AT %QX0.0 : BOOL;
Jam_Alarm   AT %QX0.1 : BOOL;
Reset_PB    AT %IX0.4 : BOOL;
Up_To_Speed : BOOL;
SpeedTimer  : TON;
JamTimer    : TON;

```

The rungs (as in the screenshot in Section 16.3):

Rung	Comment	Line	Coil
1	Start / stop with seal-in. Stop, overload or a jam drops the motor.	Parallel (NO <code>Start_PB</code> NO <code>Motor</code>), NC <code>Stop_PB</code> , NC <code>Overload</code> , NC <code>Jam_Alarm</code>	Coil <code>Motor</code>

Rung	Comment	Line	Coil
2	Belt is up to speed 3 s after the motor starts.	NO Motor , Block TON SpeedTimer with PT := T#3s	Coil Up_To_Speed
3	Jam: sensor blocked for 2 s while running latches the alarm.	NO Up_To_Speed , NO Jam_Sensor , Block TON JamTimer with PT := T#2s	Set Jam_Alarm
4	Operator reset clears the jam alarm.	NO Reset_PB , NC Jam_Sensor	Reset Jam_Alarm

To build it:

1. Create **Program — Ladder** named Conveyor (Section 16.2).
2. Click **Declarations...**, replace the placeholder with the declarations above, click **Save**.
3. Rung 1: type the comment. Drag **Parallel** onto the line; in the **Parallel branch (OR)** editor set Path 1 to NO Start_PB and Path 2 to NO Motor ; **OK**. Drag **NC** after it three times for Stop_PB , Overload , Jam_Alarm . Click **+ coil**, Kind **Coil — follows the line**, Driven by **Line 1 only**, Variable Motor .
4. **+ Rung** for rung 2: NO Motor , then **Block**: Standard block **TON — on-delay timer**, Instance SpeedTimer , input pin PT := T#3s . Coil Up_To_Speed .
5. Rung 3: NO Up_To_Speed , NO Jam_Sensor , Block TON JamTimer with PT := T#2s . **+ coil**, Kind **Set (latch)**, Jam_Alarm .
6. Rung 4: NO Reset_PB , NC Jam_Sensor . **+ coil**, Kind **Reset (unlatch)**, Jam_Alarm .
7. Click **Show ST** and compare with the listing below. Save (Ctrl+S) and **Build** (Chapter 20).

The generated file (the marker shortened):

```

(*@LADDER eyJ2IjoxLCJraW5kIjoiUFJHIiwibmFtZSI6IkNvbnZleW9yIi...@*)
(* Generated from the Ladder editor - open this POU to edit the diagram. *)
(* Editing the text directly is fine, but the Ladder view rewrites it on save. *)
PROGRAM Conveyor
VAR
    Start_PB    AT %IX0.0 : BOOL;    (* green push button *)
    Stop_PB     AT %IX0.1 : BOOL;    (* red, normally open *)
    Overload    AT %IX0.2 : BOOL;    (* motor thermal trip *)
    Jam_Sensor  AT %IX0.3 : BOOL;
    Motor       AT %QX0.0 : BOOL;
    Jam_Alarm   AT %QX0.1 : BOOL;
    Reset_PB    AT %IX0.4 : BOOL;
    Up_To_Speed : BOOL;
    SpeedTimer  : TON;
    JamTimer    : TON;
END_VAR

(* Rung 1: Start / stop with seal-in. Stop, overload or a jam drops the motor. *)
Motor := (Start_PB OR Motor) AND NOT Stop_PB AND NOT Overload AND NOT Jam_Alarm;

(* Rung 2: Belt is up to speed 3 s after the motor starts. *)
SpeedTimer(IN := Motor, PT := T#3s);
Up_To_Speed := SpeedTimer.Q;

(* Rung 3: Jam: sensor blocked for 2 s while running latches the alarm. *)
JamTimer(IN := Up_To_Speed AND Jam_Sensor, PT := T#2s);
IF JamTimer.Q THEN Jam_Alarm := TRUE; END_IF;

(* Rung 4: Operator reset clears the jam alarm. *)
IF Reset_PB AND NOT Jam_Sensor THEN Jam_Alarm := FALSE; END_IF;

END_PROGRAM

```

What the ST makes visible:

- **Rung 1** is one boolean expression; the Parallel group became the parenthesised OR.
- **Rung 2**: the timer's `IN` pin is powered by everything to its left (`Motor`) and its `Q` powers the coil.
- **Rung 3**: two contacts in series become `Up_To_Speed AND Jam_Sensor` on the timer input; the Set coil is an `IF ... THEN ... := TRUE`.
- **Rung 4** runs after rung 3 in the same scan, so while the reset button is held with the sensor clear the Reset wins. With the sensor still blocked the alarm cannot be cleared - the NC contact enforces "fix the jam first".
- `Jam_Alarm` stops the motor in rung 1 on the **next** scan (rung 1 reads it before rung 3 sets it). One scan of delay is normal ladder behaviour.

WARNING: This example is for illustration. A stop button and an overload in a PLC program are **not** an emergency stop. Emergency stopping and guard interlocks must be implemented in a safety-rated circuit or safety controller, independent of this program.

16.10 Raw ST rungs and editing a rung as ST

Some logic is awkward to draw - a `CASE`, a loop, arithmetic. A rung can hold Structured Text instead of a diagram.

To edit a rung as ST:

1. Click the rung's **ST...** button, or double-click the rung background (not an element).
2. The **Rung n — Structured Text** dialog shows the ST this rung generates. Edit it.
3. Click **Save as ST**.

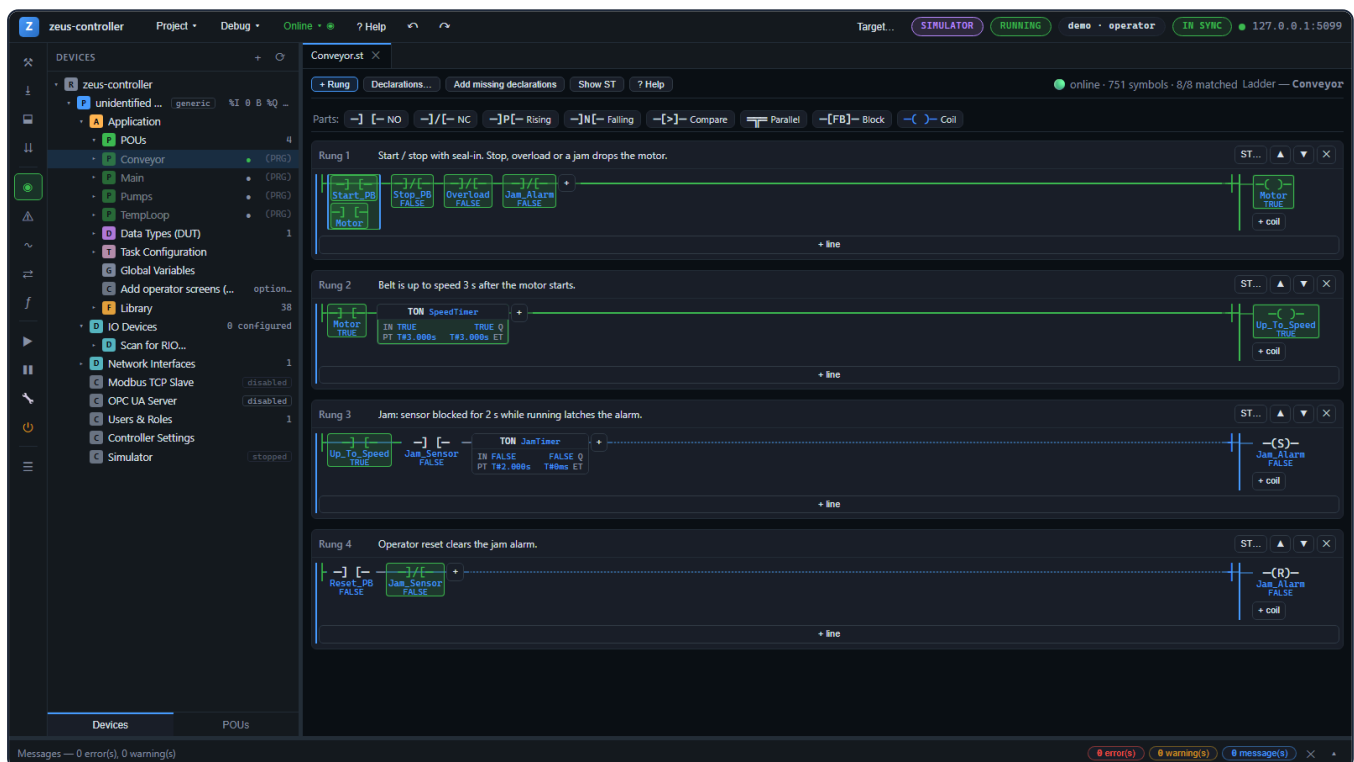
The rung becomes a **raw ST rung**: it compiles to exactly your text, it shows the text instead of the diagram, and its header carries an **ST** badge. Double-click it (or **ST...**) to edit again. **Back to graphical** restores the drawn rung and discards the ST; **Cancel** changes nothing.

Rules for raw ST rungs:

- Statements only; no **VAR** blocks. Declare variables with **Declarations...**
- **Add missing declarations** does not read raw ST.
- The rung shows no power flow online. Watch its variables in the **Variables** tab.
- Keep them short. A diagram that is mostly raw ST rungs should be an ST POU.

16.11 Online: power flow and live values

When the IDE is online and the controller is running this program, the diagram animates:



The same Conveyor program online: energised paths and elements in green, live values under each operand, timer boxes with live PT and ET.

- The **energised path** is drawn green: from the left rail along each line up to the first open element, and on to the coil when the line conducts.
- Each contact and coil shows its **live value** (**TRUE** / **FALSE**) under the variable name; a conducting element is highlighted.
- A function-block box shows its live pins - for a timer **IN** , **PT** , **Q** and **ET** (**T#3.000s** , **T#0ms**).
- The live status in the toolbar reads **online · N symbols · X/Y matched** : N symbols in the controller's program, of which X of the Y variables this diagram uses were found.

If the status says `0/... matched`, the controller is running a different program - download this one. A partial match usually means a variable was renamed or added since the last download.

Power flow is computed in the IDE from the live values of the operands, using the same logic as the generated ST. It reflects the values sampled at the last poll, not every scan: a pulse shorter than the poll interval (an edge contact, a `TP` of a few milliseconds) may never be seen lit. Use the trace (Chapter 23) for fast signals.

Writing and forcing from the diagram. Right-click a contact, compare or coil and choose **Set value....** The same write/force dialog as the variables pages opens (Chapter 22): a `%`-addressed variable can be forced, a plain variable is written.

WARNING: Setting a value or forcing an output changes a running machine immediately. Make sure it is safe for people and plant before you do it, and release every force afterwards.

16.12 A second example: counting, comparing, data blocks and shared coils

This program shows the remaining parts: a data block on its own rung, a compare, a counter with a value output, an edge contact with set/reset coils on separate lines, and a coil driven by two lines ORed.

Rung	Line(s)	Coil(s)
1	Block <code>SCALE</code> <code>LevelScale</code> (<code>raw := Level_Raw , ...</code>), output <code>eu → Level_Pct</code>	none
2	Compare <code>Level_Pct > 90.0</code>	Coil <code>High_Level</code>
3	NO <code>Part_Eye</code> , Block <code>CTU</code> <code>PartCounter</code> (<code>R := Reset_PB , PV := 12</code>), output <code>CV → Parts_In_Box</code>	Coil <code>Box_Full</code>
4	Line 1: Rising <code>Box_Full</code> ; Line 2: NO <code>Ack_PB</code>	Line 1: Set <code>Horn</code> ; Line 2: Reset <code>Horn</code>
5	Line 1: NO <code>Manual_Run</code> ; Line 2: NO <code>Auto_Mode</code> , NO <code>Auto_Demand</code>	Coil <code>Pump</code> , Driven by All lines (OR)

Generated ST (header comments and marker omitted; `SCALE` is a library block, Chapter 19):

```

PROGRAM Packing
VAR
    Level_Raw      AT %IW4      : INT;
    Part_Eye       AT %IX0.5    : BOOL;
    Reset_PB       AT %IX0.6    : BOOL;
    Ack_PB         AT %IX0.7    : BOOL;
    Manual_Run     AT %IX1.0    : BOOL;
    Auto_Mode      AT %IX1.1    : BOOL;
    Box_Full       AT %QX0.2    : BOOL;
    Horn           AT %QX0.3    : BOOL;
    Pump           AT %QX0.4    : BOOL;
    Auto_Demand    : BOOL;
    High_Level     : BOOL;
    Level_Pct      : REAL;
    Parts_In_Box   : INT;
    LevelScale     : SCALE;
    PartCounter    : CTU;
    _edgeb4x1k     : R_TRIG;
END_VAR

    (* Rung 1: Tank Level in percent. *)
    LevelScale(raw := Level_Raw, raw_min := 0, raw_max := 27648, eu_min := 0.0, eu_max := 100.0);
    Level_Pct := LevelScale.eu;
    (* no coil on this rung *)

    (* Rung 2: High Level above 90 %. *)
    High_Level := (Level_Pct > 90.0);

    (* Rung 3: Count parts; the box is full at 12. *)
    PartCounter(CU := Part_Eye, R := Reset_PB, PV := 12);
    Parts_In_Box := PartCounter.CV;
    Box_Full := PartCounter.Q;

    (* Rung 4: Horn once when the box becomes full; Ack silences it. *)
    _edgeb4x1k(CLK := Box_Full);
    IF _edgeb4x1k.Q THEN Horn := TRUE; END_IF;
    IF Ack_PB THEN Horn := FALSE; END_IF;

    (* Rung 5: Pump runs in manual, or in auto on demand. *)
    Pump := (Manual_Run) OR (Auto_Mode AND Auto_Demand);

END_PROGRAM

```

Notes:

- In rung 4 the edge detector is called while the lines are evaluated, before either coil; then the Set and the Reset follow in the order they were added.
- **CTU** counts rising edges of **CU** itself, so a plain NO contact is right in front of a counter; a Rising contact there would be redundant.
- After **Add missing declarations** on this diagram, **Level_Pct** and **Parts_In_Box** would have been declared **BOOL** - they were corrected to **REAL** and **INT** by hand (Section 16.8.2).

16.13 A ladder function block

A **Function Block** — **Ladder** POU draws the logic of one device once and is instantiated per device. Its declarations are its interface:

```
(* Generated from the Ladder editor - open this POU to edit the diagram. *)
(* Editing the text directly is fine, but the Ladder view rewrites it on save. *)

FUNCTION_BLOCK Starter
VAR_INPUT
    Start_Cmd : BOOL;
    Stop_Cmd  : BOOL;
    Trip      : BOOL;
END_VAR
VAR_OUTPUT
    Run       : BOOL;
    Running_5s : BOOL;
END_VAR
VAR
    RunTimer : TON;
END_VAR

(* Rung 1: Seal-in. *)
Run := (Start_Cmd OR Run) AND NOT Stop_Cmd AND NOT Trip;

(* Rung 2: Has been running for 5 s. *)
RunTimer(IN := Run, PT := T#5s);
Running_5s := RunTimer.Q;

END_FUNCTION_BLOCK

PROGRAM Line
VAR
    Fan_Start AT %IX4.0 : BOOL;
    Fan_Stop  AT %IX4.1 : BOOL;
    Fan_Trip  AT %IX4.2 : BOOL;
    Fan       AT %QX4.0 : BOOL;
    FanStarter : Starter;
END_VAR
FanStarter(Start_Cmd := Fan_Start, Stop_Cmd := Fan_Stop, Trip := Fan_Trip);
Fan := FanStarter.Run;
END_PROGRAM
```

(Line is an ordinary ST program; in a ladder program you would place Starter as a **Block**, with Block type Starter, Input pin Start_Cmd, Output pin Run, and the other inputs as pins - they are loaded automatically from the declaration.)

Online, a ladder function block used more than once shows an **Instance** picker so you can watch one particular instance (Chapter 22).

16.14 Tips and limitations

- Press **Show ST** while you learn; every construct maps to one or two lines of ST.
- Avoid program-level names that are also members of standard or library blocks (start, stop, reset, enable); prefer Start_PB, Stop_Cmd.

- Use each coil variable once. Combine conditions on lines of one rung instead of writing the same output from two rungs.
- Use distinct instance names for every timer, counter and block; two boxes with one instance share one state.
- A Parallel element cannot hold a block or another Parallel element. Use two lines driving an **All lines (OR)** coil instead.
- The diagram has no graphical jumps, labels or returns; use a raw ST rung if you really need one.
- The multi-step Undo/Redo of the text editors does not cover the graphical diagram, and deleting a rung or element is not confirmed. To back out of a mistake, close the POU without saving (its last saved version is kept), or save often and use version control.
- The Ladder editor help (**? Help**) summarises this chapter on one page.

17. Sequential Function Charts

This chapter describes Sequential Function Charts (SFC) in Zeus: the text syntax of steps, transitions and actions, the action qualifiers, the step flags, parallel and alternative branches, exactly how a chart executes in each scan, and the **Chart** view that draws the sequence, lights up the active step and builds the chart for you. It ends with a complete, compile-checked mixing sequence. Use SFC for anything that happens in order - a fill-heat-mix-drain cycle, a machine start-up sequence, a batch phase. Continuous control and interlocks belong in ST or Ladder beside the chart.

17.1 How SFC is implemented

A chart is **text in an ST POU**: `INITIAL_STEP`, `STEP`, `TRANSITION` and `ACTION` blocks written after the declarations. The compiler **lowers** the chart into ordinary Structured Text before it compiles it. Nothing downstream knows that SFC exists: no special opcodes, no second execution engine. That has practical consequences:

- The per-scan instruction budget, breakpoints, single-stepping, forcing and `RETAIN` all apply to a chart exactly as to any ST.
- Every step becomes a variable, so its state is readable by the variables pages, the trace, OPC UA, the EtherNet/IP tag server and your own logic.
- The **Chart** view is drawn from the same text the compiler reads. What is drawn is what runs.

17.2 Creating a chart

To create a chart:

1. Right-click **Application** (or **POUs**) and choose **New POU...**
2. Enter the **Name** and choose **Program — SFC (sequence)** as the **Type**.
3. Click **Create**.

The new POU contains an empty `VAR` block and the opening step:

```
PROGRAM Fill
VAR
END_VAR
(* A sequence: steps, transitions between them, and actions that run
   while a step is active. See Help > SFC. *)
INITIAL_STEP Start:
END_STEP
END_PROGRAM
```

Any POU whose body starts with `INITIAL_STEP` is a chart, whichever template it came from. The tree then shows a **Chart** node under the POU, and the POU's right-click menu offers **Open Chart**.

Program — S88 phase (batch) creates a PROGRAM with a phase state machine already wired to a chart; it is described in Chapter 28.

A chart may be the body of a `PROGRAM` or a `FUNCTION_BLOCK`. A `FUNCTION` cannot be a chart ("a FUNCTION cannot be a chart: it keeps no state between calls, and a step that cannot stay active is not a step").

17.3 Syntax

The body of a chart POU is, in this order:

1. optional **statements** that run every scan (Section 17.7);
2. the **chart**: steps, actions and transitions, starting with `INITIAL_STEP`.

Nothing may follow the chart except the closing `END_PROGRAM` / `END_FUNCTION_BLOCK`.

17.3.1 Steps

```
INITIAL_STEP Name:
    <statements and action associations>
END_STEP

STEP Name:
    <statements and action associations>
END_STEP
```

- Exactly **one** `INITIAL_STEP` per chart. A chart without one is refused ("this chart has no `INITIAL_STEP` - nothing would ever be active, so the sequence could not start"); a second is refused too.
- Step names are identifiers, unique in the chart, and must not clash with a variable of the POU (each step **is** a variable).
- Statements written in a step run on every scan while the step is active.
- A step may be empty.
- Steps must be written before the transitions that name them.
- A step that no transition enters is reported: "no transition enters step 'X' - it can never become active".

17.3.2 Actions and action associations

An action is a named block of statements, written once and associated with one or more steps:

```
ACTION Name:
    <statements>
END_ACTION
```

Inside a step, an **association** runs it with a qualifier:

```
ACTION Name(Q);
```

The keyword `ACTION` marks the association, so it can never be confused with a function call. `ACTION Name;` without a qualifier means `N`.

17.3.3 Transitions

```
TRANSITION FROM Source TO Target := <BOOL expression>;
END_TRANSITION
```

- `Source` and `Target` are step names. A list in parentheses on either side makes a parallel branch (Section 17.5): `FROM (A, B) TO C`, `FROM A TO (B, C)`.

- The condition is any BOOL expression: variables, comparisons, function-block outputs, step flags (`Filling.T > T#30s`).
- An optional name after `TRANSITION` is accepted for readability: `TRANSITION FillDone FROM Filling TO Heating := ...`.
- The semicolon after the condition is optional; `END_TRANSITION` is required.

A transition condition is an expression, not a block of statements: it cannot call a function block. Call the block in the statements before the chart and use its output in the condition.

17.4 Action qualifiers

Qualifier	Effect
N	The action runs on every scan the step is active. The default.
P	The action runs once , on the first scan the step is active.
S	Set: with a BOOL variable name, sets it TRUE while the step is active; it stays TRUE after the step is left.
R	Reset: with a BOOL variable name, sets it FALSE while the step is active.

`S` and `R` may name a plain BOOL variable instead of an action - `ACTION Agitator(S)`; - which is how a chart latches an output across several steps and releases it later. Any other qualifier on a name that is not an action is an error: "'X' is not an action in this chart, and only the S and R qualifiers may name a variable directly".

Important: When `S` or `R` names an **action** that exists in the chart, the action's body simply runs while the step is active, as with `N` - there is no stored action. Use `S` and `R` with a BOOL variable name, and make sure no `ACTION` block of the same name exists. The Chart view's **Action...** command always writes an `ACTION` body (Section 17.8.2); for an `S` or `R` association on a variable, delete that generated body.

Timed qualifiers are refused, not approximated: `L`, `D`, `SD`, `DS` and `SL` each give

```
the timed qualifier D is not implemented. Use N and test the step's own
timer instead: IF S0.T > T#5s THEN ...
```

Each timed qualifier needs its own timer with defined behaviour across a step exit, and one that is nearly right is worse than none, because the chart still looks correct on screen. The step timer does the same job in a form anyone can read:

Instead of	Write in the step
<code>ACTION Horn(D);</code> with T#5s (delayed)	<code>Horn := Filling.T >= T#5s;</code>
<code>ACTION Horn(L);</code> with T#5s (limited)	<code>Horn := Filling.T < T#5s;</code>

Any other qualifier is refused with "'X' is not an action qualifier - expected N (while active), P (once on entry), S (set) or R (reset)".

Note: An **N** action (or a statement in a step) **does not undo what it wrote** when the step is left: the statements simply stop running and every variable keeps its last value. IEC charts reset the Boolean of an **N** action on exit; Zeus does not. Write outputs explicitly in the steps that must switch them off (as the example in Section 17.9 does), or drive them from step flags: `Inlet_Valve := Filling.X AND Level < Target;` in the statements before the chart.

17.5 Step flags

Each step is a variable of the built-in structure `SFC_STEP`:

Flag	Type	Meaning
<code>Step.X</code>	BOOL	TRUE while the step is active
<code>Step.T</code>	TIME	How long the step has been active in its current activation; <code>T#0ms</code> when it is not active
<code>Step.T0</code>	TIME	When the step was entered (monotonic clock; internal)
<code>Step.XP</code>	BOOL	Whether the step was active in the previous scan (internal; used by <code>P</code>)

`X` and `T` are ordinary symbols: watch them in the variables table, plot them in the trace, read them over OPC UA, interlock on them. A step that is not active reads `T#0ms`, never the time it spent there last time, so an interlock such as `Heating.T > T#10m` cannot fire from a stale value.

The chart also declares hidden variables (`__sfc_started`, `__sfc_fire0` ...). Do not use them.

17.6 Branches

17.6.1 Alternative branch (choice)

Two or more transitions leaving the **same** step form an alternative branch. When several conditions are TRUE in the same scan, **the transition written first in the file wins**. The chart is never in two places at once.

```
TRANSITION FROM Filling TO Abort := Aborted;          END_TRANSITION
TRANSITION FROM Filling TO Heating := Level >= Target;  END_TRANSITION
```

Written in this order, an abort takes priority over "full". The order in the file is the priority - put the safety exit first.

17.6.2 Parallel branch (simultaneous)

A list of steps on the right of a transition activates them all at once; a list on the left waits for all of them:

```
TRANSITION FROM Filling TO (Heating, Mixing) := Level >= Target;          END_TRANSITION
TRANSITION FROM (Heating, Mixing) TO Draining := Hot AND Mixing.T > T#2m;  END_TRANSITION
```

- After the divergence, `Heating` and `Mixing` are both active and each runs its own statements and actions.
- The convergence fires only when **every** source step is active **and** the condition is TRUE. If one arm has more steps than the other, the join waits for the last step of each arm.
- Up to **8** steps on either side of one transition.

17.6.3 Loops (jumps)

A transition to an earlier step closes a loop - that is how a sequence returns to its idle state or repeats a stage. There is no separate jump element; `TRANSITION FROM Complete TO Idle := ...` is the jump.

17.7 How a chart runs in each scan

The lowered code runs in this order, every scan:

1. **Statements before the chart.** Anything written between the declarations and `INITIAL_STEP` runs every scan, whatever step is active.
2. **First scan only:** the initial step is activated.
3. **Step times:** for every active step, `T` is updated; every inactive step reads `T#0ms`.
4. **Actions:** for each step in the order written, if it is active: its own statements, then its associated actions in the order associated (`N` every scan, `P` only on the first active scan, `S / R` on variables).
5. **Remember** which steps are active (`XP`), for `P` actions.
6. **Transitions, in two passes.** First, every transition's condition is evaluated against the state as it is now (all its source steps active AND its condition). Then the transitions that fired are applied in file order: source steps deactivated, target steps activated with `T` reset. A transition whose sources were already deactivated by an earlier transition in this pass does not apply - that is what makes "first written wins" work.

Consequences worth knowing:

- **One transition per branch per scan.** A chart cannot race through five steps in one cycle because each happened to be ready; every step it passes through is active for at least one full scan and is visible.
- **A step entered in this scan runs its actions in the next.** Its `P` action runs on that next scan, before the transitions of that scan are evaluated.
- Transitions see the variables as the actions of **this** scan left them.
- The whole chart is inside the POU's body, so a chart in a program that the task does not call (Chapter 18) does not run at all, and a chart in a function block runs only when the instance is called.
- A chart in a function block keeps separate state per instance: two instances of the same chart can be in different steps.

17.8 The Chart view

Double-click the **Chart** node under a chart POU (or right-click the POU → **Open Chart**). The tab is titled `<POU> . Chart`.

Area	Shows
Heading	<code><POU> – Chart</code> , and "N step(s), M transition(s)"
Status	Online: active: <code><step names></code> or "no step of this chart is active". Offline: a note that the controller is not reporting this program's symbols.
Toolbar	Selected: <code><step></code> and the building commands (Section 17.8.2)
Drawing	Steps as boxes (the initial step marked), transitions with their conditions, the actions of each step with their qualifiers. The active steps are filled green with their elapsed time.
Actions	A table of the chart's <code>ACTION</code> blocks and the start of their bodies

The drawing re-reads the POU twice a second - the open editor's text if the POU is open, otherwise the saved file - and, when online, the controller's step flags. It stops polling when its tab is closed.

17.8.1 Navigating

- **Click** a step to select it; the toolbar commands act on the selected step. With nothing selected, the last step is selected.
- **Double-click** a step to open the POU's editor at that step's line.

17.8.2 Building the chart from the view

The toolbar commands edit the **text** of the POU - they splice text into it - and redraw. Nothing is saved: each edit lands in the POU's editor as an unsaved change (the toolbar says "edits <POU> — unsaved until you press Ctrl+S"), where it can be read, corrected or undone (Ctrl+Z in the editor).

Command	Writes
+ Step	A new step (Step1 , Step2 ...) after the selected one, with a transition from the selected step to it, condition TRUE
 Alternative...	Asks for a step name and a condition; adds the step and a second transition out of the selected step, appended so it has lower priority than the existing ones
 Parallel...	Asks for the two branch steps, the join step and the split and join conditions; adds all three steps and both transitions as one unit
 Jump...	Asks for an existing target step and a condition; adds a transition back to it (no new step)
 Action...	Asks for an action name and the qualifier (N — while the step is active, P — once, on entry, S — set a BOOL and leave it set, R — reset it); adds the association to the selected step and an empty ACTION body unless one of that name exists
 Rename...	Renames the step everywhere: its declaration, every transition that names it, and every Step.X / Step.T in the logic
 Delete	Removes the step and every transition that mentions it, after a confirmation that lists those transitions and any steps that would become unreachable. Not available on the initial step.

Every new transition is written with the condition **TRUE**, which fires on the next scan. Replace each **TRUE** with the real condition before you download.

Tip: Use the view to lay down the skeleton quickly, then write conditions and step statements in the editor. Comments, blank lines and ordering you wrote are never moved by the view.

17.8.3 Limitations of the view

- A chart inside a **FUNCTION_BLOCK** compiles and runs, but the view cannot light up its steps: a step inside an instance is a two-level name (Door1.Opening.X) that the view does not resolve. Watch the instance's step flags in the variables table instead, or use a **PROGRAM** where you want the animation.
- The live highlighting reads the step flags by step name. Give steps names that are unique across the programs you download together, or two charts may light each other's steps.
- The view samples at the poll rate; a step active for a single scan may not be seen. Use the trace on Step.X for that.

17.9 Worked example: a mixing sequence

A mixing vessel is filled to a target level, then heated to a target temperature **while** the agitator runs for a minimum time, then drained. A stop (the normally closed stop button opening) at any point during fill, heat or mix aborts: inputs close, heater off, agitator off, vessel drained. The batch counter increments once per batch.

Sequence:

```

      Idle
      | Start_PB AND Stop_PB
      Filling -----+
      | Level >= Fill_Target          | Aborted (first)
      +-----+-----+               |
      Heating      Mixing              |
      +-----+-----+ -----+ Aborted (first)
      | Temp >= Heat_Target AND       |
      | Mixing.T >= Mix_Time          | Abort
      Draining      | Level < 200 AND Stop_PB
      | Level < 200
      Complete      |
      | Complete.T > 10 s
      Idle <-----+
  
```

The POU:


```

PROGRAM Mixer
VAR
    (* Operator and process signals *)
    Start_PB      AT %IX1.0 : BOOL;
    Stop_PB       AT %IX1.1 : BOOL;      (* normally closed: FALSE = stop *)
    Level         AT %IW10  : INT;      (* 0..27648 = 0..100 % *)
    Temp          AT %IW12  : INT;      (* 0.1 degC per count *)

    (* Recipe *)
    Fill_Target   : INT := 18000;
    Heat_Target   : INT := 650;          (* 65.0 degC *)
    Mix_Time      : TIME := T#2m;

    (* Outputs *)
    Inlet_Valve   AT %QX1.0 : BOOL;
    Agitator      AT %QX1.1 : BOOL;
    Heater        AT %QX1.2 : BOOL;
    Drain_Valve   AT %QX1.3 : BOOL;
    Done_Lamp     AT %QX1.4 : BOOL;

    (* Bookkeeping *)
    Batch_Count   : DINT;
    Aborted       : BOOL;
    Stop_Edge     : F_TRIG;
END_VAR

    (* Statements before the chart run every scan, whatever step is active. *)
    Stop_Edge(CLK := Stop_PB);
    IF Stop_Edge.Q THEN Aborted := TRUE; END_IF;
    Done_Lamp := Complete.X;

INITIAL_STEP Idle:
    Inlet_Valve := FALSE;
    Heater      := FALSE;
    Drain_Valve := FALSE;
END_STEP

STEP Filling:
    ACTION OpenInlet(N);      (* every scan while Filling *)
    ACTION CountBatch(P);    (* once, on entry *)
END_STEP

STEP Heating:
    Inlet_Valve := FALSE;
    Heater := Temp < Heat_Target;
END_STEP

STEP Mixing:
    ACTION Agitator(S);      (* latch the agitator output ON *)
END_STEP

STEP Draining:
    ACTION Agitator(R);      (* unlatch it *)
    Heater := FALSE;
    Drain_Valve := TRUE;
END_STEP

STEP Complete:
    Drain_Valve := FALSE;

```

```

END_STEP

STEP Abort:
    ACTION Agitator(R);
    Inlet_Valve := FALSE;
    Heater      := FALSE;
    Drain_Valve := TRUE;
END_STEP

ACTION OpenInlet:
    Inlet_Valve := Level < Fill_Target;
END_ACTION

ACTION CountBatch:
    Batch_Count := Batch_Count + 1;
    Aborted := FALSE;
END_ACTION

(* Start *)
TRANSITION FROM Idle TO Filling := Start_PB AND Stop_PB;
END_TRANSITION

(* Out of Filling: abort is written first, so it wins over "full" *)
TRANSITION FROM Filling TO Abort := Aborted;
END_TRANSITION
TRANSITION FROM Filling TO (Heating, Mixing) := Level >= Fill_Target;
END_TRANSITION

(* Out of the parallel branch: again, abort first *)
TRANSITION FROM (Heating, Mixing) TO Abort := Aborted;
END_TRANSITION
TRANSITION FROM (Heating, Mixing) TO Draining :=
    Temp >= Heat_Target AND Mixing.T >= Mix_Time;
END_TRANSITION

TRANSITION FROM Draining TO Complete := Level < 200;
END_TRANSITION
TRANSITION FROM Complete TO Idle := Complete.T > T#10s;
END_TRANSITION
TRANSITION FROM Abort TO Idle := Level < 200 AND Stop_PB;
END_TRANSITION
END_PROGRAM

```

How it behaves, scan by scan:

- **Start.** With `Stop_PB` healthy (TRUE) and `Start_PB` pressed, `Idle` → `Filling` fires. On the next scan `Filling`'s actions run: `CountBatch` (P) increments the counter and clears `Aborted`, `OpenInlet` (N) opens the inlet while below target. Because the P action runs before that scan's transitions, a stale `Aborted` from an earlier batch cannot abort the new one.
- **Parallel stage.** When the level reaches the target, `OpenInlet` has already closed the valve in the same scan, and `Heating` and `Mixing` become active together. `Heating` closes the inlet explicitly and runs the heater on a simple on/off; `Mixing` latches `Agitator` with `S`.
- **Join.** `Draining` needs both arms active, the temperature reached and at least `Mix_Time` in `Mixing` (`Mixing.T` counts from the start of the parallel stage).
- **Outputs off.** `Draining` releases the agitator with `R` and switches the heater off explicitly, because a step's statements do not undo themselves (Section 17.4).

- **Abort.** The F_TRIG before the chart catches the stop button opening in any step. The abort transitions are written **before** the normal ones out of `Filling` and out of the parallel stage, so they win. `Abort` returns to `Idle` only when the vessel is empty **and** the stop button is healthy again.
- **Lamp.** `Done_Lamp := Complete.X;` uses a step flag in the statements before the chart; it shows the previous scan's state, one scan late, which is irrelevant for a lamp.

WARNING: A chart step is not a safety function. The abort path here is operational. Heater over-temperature cut-outs, agitator guards and emergency stops must be independent of the program.

Note: In this version the compiler reports "action 'OpenInlet' is never associated with a step" (and the same for `CountBatch`) for this chart although both are associated and run. The warning is issued for **every** named action, used or not, and can be ignored when the Chart view shows the action on its step. It will be corrected in a later release.

17.10 Watching and debugging a chart

- Open the **Chart** view online to see the active steps and their times.
- In the POU's **Variables** tab each step appears as a structure: open it to see `X`, `T`, `T0` and `XP`.
- Force or write a transition's **input variables** to walk the chart on the bench. Do not write `Step.X` directly: activating a step by hand bypasses the transitions and can leave two steps of one branch active.
- Breakpoints can be set on step statements and actions (Chapter 24); the chart stops mid-scan like any ST.
- The trace (Chapter 23) can record `Step.X` for every step: the result is a timing diagram of the sequence.
- To restart a chart from `INITIAL_STEP`, download it again (a cold start of the program), or build a reset transition from each step to `Idle` on a reset command. There is no separate chart-reset command.

17.11 Limits

Item	Limit
Steps per chart	64
Transitions per chart	96
Actions per chart	64
Steps on either side of one transition	8
Action associations per step	8
Charts per POU	1
Where a chart may be	Body of a PROGRAM or FUNCTION_BLOCK, after any statements; not in a FUNCTION or METHOD

17.12 Summary of differences from IEC 61131-3 graphical SFC

Topic	Zeus
Representation	Text in the POU; the Chart view draws and edits that text
Qualifiers	N , P , S , R . L , D , SD , DS , SL refused - use Step.T
S / R on an action	Runs the action like N ; use them on BOOL variables
N on exit	Variables keep their last value; nothing is reset automatically
Alternative priority	The transition written first in the file
Transitions per scan	One per branch; a step entered this scan acts on the next
Step flags	.X , .T (plus internal .T0 , .XP)
Transition conditions	BOOL expressions; no transition actions or sub-charts
Macro steps, jumps as symbols	Not provided; a transition to an earlier step is the jump

18. Task configuration

This chapter describes how the controller schedules your programs: the one real-time scan task (**MainTask**), its interval, priority and CPU core, the program call list that decides which PROGRAMs a download contains and in which order they run, tasks that run some programs at a slower rate, the service threads, scan-time monitoring and what happens when a scan overruns. Read it before you download a project with more than one PROGRAM, and before you commission a machine whose timing matters.

18.1 The execution model in one page

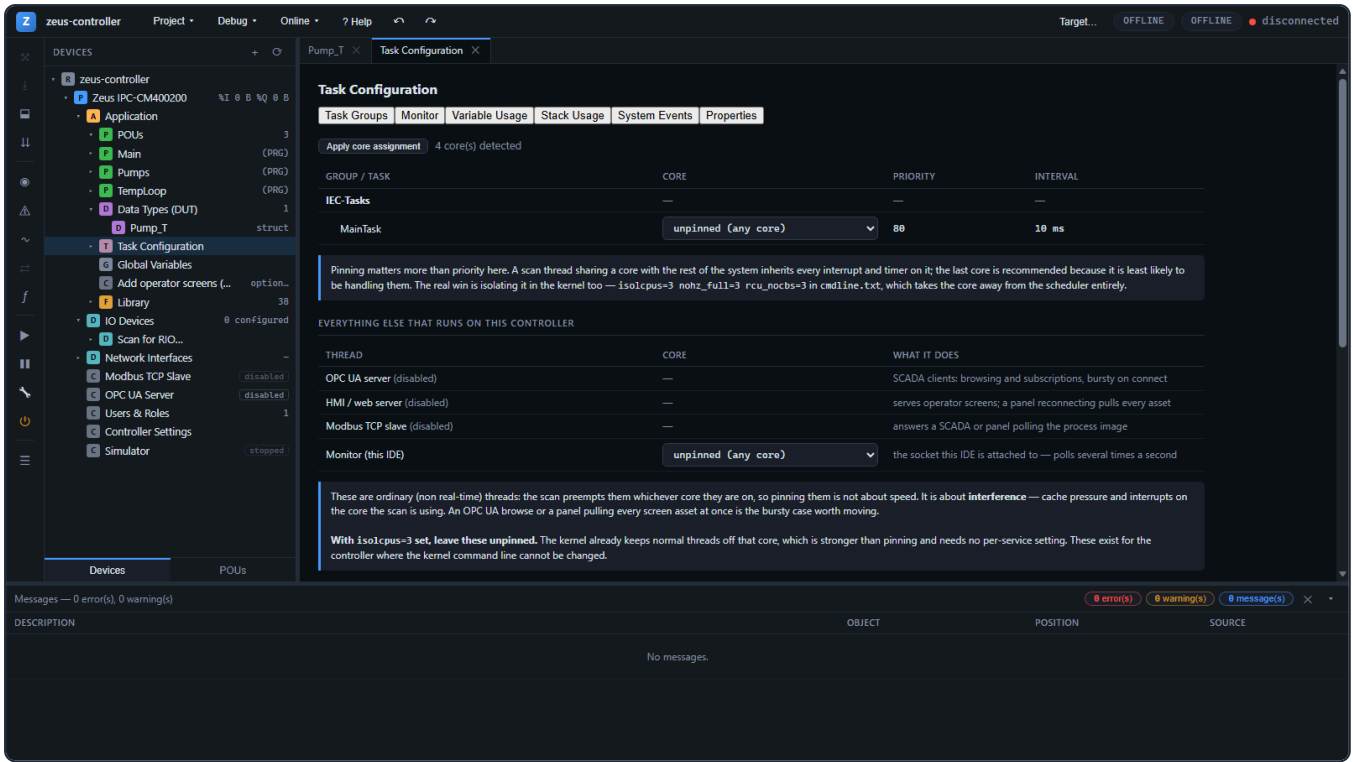
- The controller has **one** real-time scan thread, **MainTask**. It wakes at a fixed interval (default **10 ms**) on an absolute deadline, so the period does not drift.
- Each scan: read inputs → apply input forces → run the programs → apply output forces → write outputs → publish values → statistics (Chapter 21 gives the detail). The input image is frozen while the programs run.
- **Every PROGRAM in the call list runs in every scan, in call-list order**, unless it has been given a slower rate. They share the globals and the same input snapshot; each keeps its own variables between scans.
- A slower "task" is a **rate of the same clock**, not a second thread: a program in a 100 ms task runs inside every tenth MainTask scan. Two programs can therefore never see a half-updated view of each other, and nothing on the cyclic path takes a lock. The price is that a slow program's execution time lands on the scan that runs it.
- The whole scan shares one **instruction budget** (200 000 VM instructions). A program that exceeds it - a runaway loop - faults the cycle instead of hanging the core (Chapter 21).

The configuration lives in `controller.json` under "task" (Appendix H). Changes to the task configuration take effect when the runtime restarts or on **Download all**; the IDE says so when you apply them.

18.2 Where the task settings are

The **Task Configuration** node sits under **Application**:

Tree node	Opens
Task Configuration	The Task Configuration tab: task groups, cores, tasks and programs, monitoring
Task Configuration → MainTask (shows its interval, e.g. 10 ms)	The MainTask tab: interval, priority, overrun policy, watchdog settings and the POU calls list



The Task Configuration tab, Task Groups page: MainTask with its core, priority and interval, and the service threads that run on the controller.

The **Task Configuration** tab has six pages, selected by the buttons at the top:

Page	Contents
Task Groups	MainTask and its core; the service threads and their cores; the Tasks and Programs editors (Section 18.5)
Monitor	Live scan statistics (Section 18.8)
Variable Usage	How many symbols the running program has, by storage: <code>address</code> (<code>%I/%Q/%M</code>), <code>memory</code> (program and global data), <code>instance</code> (function-block instances)
Stack Usage	The fixed per-scan limits (Section 18.10)
System Events	The scan engine's state and each IO device's state
Properties	A summary of the task settings and the controller's real-time posture, and the per-program period table (Section 18.6.3)

18.3 MainTask settings

Open **Task Configuration** → **MainTask**. The **Configuration** section:

Field	Range / values	Default	Meaning
Priority (0..99)	0-99	80	Linux <code>SCHED_FIFO</code> real-time priority of the scan thread. 80 is above most kernel threads and below the network soft-interrupt threads of a real-time kernel. Asynchronous IO driver threads default to one below (79), so the scan always preempts its own IO.

Field	Range / values	Default	Meaning
Type	Cyclic	Cyclic	Only cyclic tasks exist. There are no event or free-wheeling tasks.
Interval (ms)	≥ 0.25 , steps of 0.1	10	The scan period. The controller refuses a period below the board's safe minimum: 500 μs on a CM4 (the IPC-CM400200), 250 μ s on a CM5.
CPU affinity	-1, 0 ... cores-1	-1	The core the scan thread is pinned to; -1 lets the runtime choose (it picks the isolated core when the kernel has one).
Overrun policy	skip , catchup , fault	skip	What happens when a scan does not finish within its period (Section 18.9).
Lock memory (mlockall)	on/off	on	Locks the runtime's memory into RAM so a page fault can never stall a scan. Leave it on.

The **Watchdog** section:

Field	Default	Meaning
Enable	off	Saved to the configuration (<code>watchdog_enable</code>).
Time (ms)	0	Saved to the configuration (<code>watchdog_ms</code>).
Sensitivity (overruns)	10	How many consecutive overruns are tolerated before the overrun policy acts (<code>overrun_fault_limit</code>). It only has an effect with the <code>fault</code> policy.

Important: In this version the runtime does not act on **Enable** and **Time (ms)**: they are stored in `controller.json` but no scan watchdog is armed from them. Scan-time supervision is provided by the overrun policy and **Sensitivity** (Section 18.9), and loss of communication with the IO is handled by each fieldbus device's own watchdog (Chapters 7 to 10). Do not rely on the **Watchdog** fields for a safety or validation argument.

Click **Apply** at the bottom of the tab to save the settings ("Task configuration saved").

18.4 The program call list

The **POU calls** section of the MainTask tab is the task's call list: the PROGRAMs the task runs, in order.

Control	Does
Program selector + Add call	Adds the selected PROGRAM to the end of the list. A program already in the list is refused.
Move up / Move down	Moves the selected call; the list order is the execution order.
Remove call	Removes the selected call.
Open POU	Opens the selected program. Double-clicking a row does the same.
Table	# , POU , Status (runs 1st , runs #2 ...). Click a row to select it.

Each change to the list is saved immediately ("Call list updated").

The list follows the POU's: renaming or moving a program (Chapter 13) renames its entry in place, keeping its position, and deleting a program removes its entry.

What the call list controls:

1. **What a download contains.** A download **bundles every PROGRAM in the call list** - the open editor's text for the program being downloaded, the saved files for the others - into one unit, together with the DUTs, globals, library and function blocks (Chapter 20). A PROGRAM that is not in the call list is not downloaded, and does not run.
2. **Execution order.** The programs run top to bottom, every scan. Order matters when one program computes a value another uses in the same scan: put the producer first.
3. **An empty list** means "the program being downloaded runs on its own": the unit then holds that one program. The table says "No calls. The downloaded program runs on its own."

The program you download is always included: if it is not in the list yet it is added for that download and runs first. Disabled POU's are skipped.

Note: Every PROGRAM in the unit runs. A PROGRAM placed in a library file (lib/) would be linked into every build and run on every controller - keep programs out of the library.

Tip: Keep the call list explicit on every project with more than one program. It is the one place a reviewer can see what the controller runs.

The note under the list states that "several tasks with different intervals are not supported yet". That refers to separate real-time threads; programs at different rates of MainTask **are** supported, as the next section describes.

18.5 Tasks and task groups

On the **Task Groups** page of the **Task Configuration** tab:

The Group / Task table shows the group **IEC-Tasks** with **MainTask** - its **Core** selector, **Priority** and **Interval** - and one row for each additional task, marked "in the MainTask thread" with its interval and offset.

The Tasks editor (heading **Tasks**) defines slower tasks:

Column	Meaning
Task	The task name - an identifier, unique, not MainTask
Interval	Its period in ms; a whole multiple of MainTask's interval
Offset	In scans: which scan of its period the task runs on (0 ... multiple-1)
Programs	One chip per PROGRAM in the project; a highlighted chip runs in this task. Click a chip to move that program into this task, click again to move it back to MainTask.
x	Deletes the task; its programs go back to MainTask.

MainTask's own row lists the programs that run every scan.

To add a task:

1. Click **+ Add task**. The **Add task** dialog opens.
2. Enter the **Name** (suggested Task1 , Task2 ...), the **Interval (ms)** and the **Offset (scans)**.

3. Under **POUs to run in this task**, tick the PROGRAMs to move into it. Each line shows the program's current task, and "not in the call list yet, will be added" where applicable. Function blocks and functions are listed separately - they are called from a PROGRAM, not by a task. + **New PROGRAM...** opens **New POU**.
4. Click **Add task**.
5. Click **Apply tasks** under the tables. **Revert** discards unsaved changes.

The Programs table lists every enabled PROGRAM with a **Task** selector (move it between tasks) and **Running as**, which is read back from the controller: `every scan`, `every 10 scans (100 ms)`, or "not in the running program". The last choice in the **Task** selector, - **not running** -, takes the program out of MainTask's call list when you press **Apply tasks** (choosing a task again puts it back); a program the call list leaves out shows as - **not running** - when the page opens. At least one program has to run. A change to the call list alone is applied by the next download without stopping the controller (Section 20.4.2).

Apply tasks checks the configuration with the controller's own rules before it saves - an interval that is not a whole multiple of MainTask's, or an offset not less than the number of scans per run, is refused with the reason - then:

- writes a `task.programs` entry (name, period, offset) for each program outside MainTask;
- writes the task names to `task.groups` (only the IDE uses them; the controller knows periods, not names);
- makes sure every program given a task is in the call list. If the call list was empty, it is first filled with every enabled PROGRAM, because once any program has its own rate the list must say what runs.

The result is saved to the project with "Tasks saved — restart the runtime (or Download all) to apply".

18.6 Programs at different rates

18.6.1 How it works

A program with a period of $N \times$ the MainTask interval runs on one scan in N , on the scan where `scan number MOD N = offset`. On the other scans it is skipped. A skipped program loses nothing: its variables, timers and function-block instances are exactly as it left them, and its IEC timers still measure real time.

18.6.2 Rules

Rule	Why
The period must be a whole multiple of the MainTask interval	25 ms on a 10 ms scan is refused, naming 20 ms and 30 ms. Rounding would run the program at a rate nobody chose.
At most 16 programs may have their own period	<code>task.programs</code> holds up to 16 entries
The offset is taken modulo the multiple	An offset of 12 on a 10-scan period behaves as 2
A period entry naming a program that is not in the downloaded unit	The controller reports it, and the IDE shows "A program named in the task configuration is not in the running program. It is scheduling nothing — check for a rename." A POU renamed with Rename... takes its entry with it; this row applies when the name in the <code>PROGRAM</code> header was changed by hand.

Stagger slow programs with offsets. Two 1-second programs with offset 0 both land on the same scan; that scan carries both and shows up as jitter nobody can explain. Give them different offsets.

Function blocks that integrate or differentiate over time (`PID` , `RAMP` , `TOTALIZER` , `FILTER_PT1` ...) take `dt_ms` . In a slow program, `dt_ms` is **that program's period**, not the MainTask interval.

18.6.3 The Properties page

The **Properties** page shows the same schedule as a table, **Programs in this task**: each PROGRAM with a **Period** (ms, stepping by the task interval), an **Offset** and **Running as**. **Apply task periods** saves it ("Task periods saved — restart the runtime to apply"). Only programs whose period or offset differs from "every scan" are written. Both editors write the same `task.programs` list; use whichever you prefer, and do not leave unapplied changes in the other.

18.6.4 Example

A fast control program at 10 ms, a totaliser and a reporting program at 1 s, staggered.

Global Variables:

```
VAR_GLOBAL
    Flow_Raw    AT %IW20 : INT;      (* flowmeter, 0..27648 = 0..120 m3/h *)
    Flow_m3h    : REAL;
    Pump_Run    AT %QX5.0 : BOOL;
END_VAR
VAR_GLOBAL RETAIN
    Total_m3    : LREAL;            (* survives a power cut *)
END_VAR
```

Programs (three POU; `SCALE` is a library block, Chapter 19):

```

PROGRAM FastControl
VAR
    FlowScale : SCALE;
END_VAR
(* Every 10 ms: scale the flow; the pump interlock lives here. *)
FlowScale(raw := Flow_Raw, raw_max := 27648, eu_min := 0.0, eu_max := 120.0);
Flow_m3h := FlowScale.eu;
END_PROGRAM

PROGRAM Totaliser
(* Runs every 1000 ms (period set in Task Configuration).
   m3/h integrated over one second is m3/h / 3600 per run. *)
IF Pump_Run THEN
    Total_m3 := Total_m3 + Flow_m3h / 3600.0;
END_IF;
END_PROGRAM

PROGRAM Reporting
VAR
    Hourly_Avg : REAL;
    Sum        : REAL;
    Samples    : DINT;
END_VAR
(* Runs every 1000 ms, offset so it never shares a scan with Totaliser. *)
Sum := Sum + Flow_m3h;
Samples := Samples + 1;
IF Samples >= 3600 THEN
    Hourly_Avg := Sum / DINT_TO_REAL(Samples);
    Sum := 0.0;
    Samples := 0;
END_IF;
END_PROGRAM

```

Configuration:

1. MainTask: **Interval (ms)** 10. **POU calls:** FastControl.st, Totaliser.st, Reporting.st, in that order.
2. Task Groups → + **Add task:** Name Slow_1s_A, Interval 1000, Offset 3, tick Totaliser. **Add task.**
3. + **Add task:** Name Slow_1s_B, Interval 1000, Offset 7, tick Reporting. **Add task.**
4. **Apply tasks**, then **Download all**.

The resulting task section of controller.json:

```

"task": {
  "period_ms": 10,
  "rt_priority": 80,
  "overrun_policy": "skip",
  "overrun_fault_limit": 10,
  "lock_memory": true,
  "calls": ["FastControl.st", "Totaliser.st", "Reporting.st"],
  "programs": [
    { "name": "Totaliser", "period_ms": 1000, "offset": 3 },
    { "name": "Reporting", "period_ms": 1000, "offset": 7 }
  ],
  "groups": [
    { "name": "Slow_1s_A", "period_ms": 1000, "offset": 3 },
    { "name": "Slow_1s_B", "period_ms": 1000, "offset": 7 }
  ]
}

```

Online, the **Programs** table then reads every scan for FastControl and every 100 scans (1000 ms) for the other two.

18.7 Cores and service threads

The scan is single-threaded: every PROGRAM runs in sequence on one core. More cores do not make a scan faster; they keep other work **off** the scan's core.

On the **Task Groups** page:

- **MainTask** → **Core: unpinned (any core)** or **Core 0 ... Core 3**; the last core is marked — **recommended**, because it is the least likely to be handling interrupts. The line above the table shows how many cores were detected on the connected board.
- **Everything else that runs on this controller** lists the other long-lived threads with a **Core** selector each (only for enabled services):

Thread	What it does
IO driver — <code><device></code> (one per asynchronous IO device)	The fieldbus cyclic exchange on its own thread - the other time-sensitive thread
OPC UA server	SCADA clients: browsing and subscriptions, bursty on connect
HMI / web server	Serves operator screens; a panel reconnecting pulls every asset
Modbus TCP slave	Answers a SCADA or panel polling the process image
Monitor (this IDE)	The socket the IDE is attached to; polls several times a second

Apply core assignment saves the scan core and every service core together ("Core assignment saved — restart the runtime to apply").

The service threads are ordinary (non-real-time) threads; the scan preempts them wherever they run. Pinning them is about **interference** - cache pressure and interrupts on the scan's core - not speed.

The best setting is kernel isolation. With `isolcpus=3 nohz_full=3 rcu_nocbs=3` on the kernel command line (`cmdline.txt`) and MainTask on core 3, Linux keeps every ordinary thread off that core by itself - stronger than

pinning, and the services can stay **unpinned**. The per-service cores are for a controller whose kernel command line cannot be changed. The one thread worth pinning either way is an asynchronous IO driver.

18.8 Scan-time monitoring

Online, **Task Configuration** → **Monitor** shows, updated live:

Field	Meaning
Running for	Time since the last runtime restart (cycles × period)
Cycles	Scans since the runtime started
Overruns	Scans that did not finish within the period (red when not zero)
Jitter avg / Jitter max	Wake-up latency: actual wake time minus the intended deadline. Max is cumulative since start.
Exec last	Duration of the last scan's work (input read to output write)

Overruns should be **zero**. Jitter is what decides whether a control loop is deterministic, and it is the number to show a customer - measured on a controller prepared as Section 18.11 describes. The full jitter histogram and per-phase timings are in the diagnostics (Chapter 25).

Keep **Exec last** well under the interval - as a working rule below 50 % of it at the worst point of the machine cycle - so there is room for a slow program's scan, an online change and a burst of communication.

18.9 Overruns and the overrun policy

An **overrun** is a scan whose work did not finish before the next deadline. Every overrun increments the counter and writes a record with the overshoot in microseconds and the scan number. What happens next is the **Overrun policy**:

Policy	Behaviour	Use for
skip — resync, keep phase (control loops)	Missed deadlines are dropped; the next scan starts at the next deadline on the original time grid	Control loops (default). Phase is preserved, which is what a PID needs.
catchup — run back to back (logging)	The deadline stays in the past; scans run back to back until caught up	Data collection where every sample matters more than when it is taken
fault — stop after N (validated systems)	Like skip , but after Sensitivity consecutive overruns the runtime enters FAULT	Validated systems: a controller silently missing deadlines is worse than one that stops and says so

In **FAULT** the program stops and the outputs go to the per-device fault policy (clear or hold last; Chapter 21). An explicit reset is required.

Budget faults are different. A scan that exceeds the instruction budget (200 000 instructions for all programs together) is aborted at that point regardless of the overrun policy. See Chapter 21.

18.10 Fixed limits (Stack Usage page)

The VM has no call stack to overflow in the C sense: locals, function-block instances and the operand stack are sized at compile time and allocated once, because the cyclic path never allocates memory. What is bounded is work per scan:

Limit	Value	Protects against
Instruction budget	200 000 per scan	A runaway <code>WHILE</code> faults the cycle instead of wedging the core
Operand stack	256	Expression nesting depth
Call depth	32	A function block or function calling itself
Function frames	64 KB	FUNCTION locals, per call
Programs with their own period	16	
POUs in one download unit	256 (library, FBs, methods, functions and programs together)	

18.11 Real-time posture

The determinism of the scan depends more on how the controller's Linux is prepared than on anything in the program. Four things matter:

1. A **PREEMPT_RT** kernel.
2. **Core isolation**: `isolcpus=3 nohz_full=3 rcu_nocbs=3` in `cmdline.txt`, with MainTask on core 3.
3. The **performance** CPU frequency governor (a frequency change is a latency spike).
4. Deep CPU idle states blocked - the runtime does this itself at start-up.

The standard controller image does **not** ship with a PREEMPT_RT kernel. Check what the connected controller has:

- **Task Configuration** → **Properties**: the **PREEMPT_RT** row reads `yes` (green) or `no` (amber). On a kernel without it, the **Task Groups** page adds "This kernel is not PREEMPT_RT, so pinning will help but the tail stays wide."
- On the controller, `zeus-plcd --board` reports the kernel type, the isolated cores and the governor, and counts a warning for each deviation.

What each step is worth, measured on an IPC-CM400200 at a 10 ms interval (maximum wake-up jitter, cumulative):

Posture	Jitter avg	Jitter max	Overruns
Standard kernel, <code>ondemand</code> governor, no isolation	13 μ s	~100 μ s	0
+ <code>performance</code> governor + isolated core 3	7 μ s	71 μ s	0
+ PREEMPT_RT kernel	7 μ s	57 μ s	0

At 10 ms on an isolated core the standard kernel is already adequate for ordinary machine control. PREEMPT_RT earns its place at short intervals (towards the 500 μ s floor), under heavy interrupt load, and with EtherCAT

distributed clocks (Chapter 9). Isolation and the governor cost nothing but a reboot; apply them on every controller.

Important: A jitter figure from a controller without this preparation is a measurement of Linux scheduling, not of the controller. Do not quote one to a customer or put it in a validation report.

If the runtime cannot obtain real-time priority (for example in a development environment), it starts anyway, logs "the runtime will start in non-deterministic mode" and runs at normal priority. Timing figures from the simulator on a PC (Chapter 26) are not representative of the controller either.

18.12 Checklist

- [] Every PROGRAM that must run is in the **POU calls** list, in the right order.
- [] The interval is the slowest that meets the process (10 ms is typical), and **Exec last** is well under it.
- [] Slow programs have their own task with staggered offsets; their `dt_ms` inputs match their period.
- [] MainTask is on the isolated core; the kernel command line has `isolcpus`; the governor is `performance`.
- [] The overrun policy is chosen deliberately (`fault` for validated systems) and **Overruns** is zero after a full machine cycle.
- [] Tasks were applied and the runtime restarted or **Download all** done; the **Programs** table shows the expected "Running as".

19. Global variables and libraries

This chapter covers the two things every POU in a project shares: the **global variables** (`globals.st` and further global variable lists under `gv1/`) and the **function-block library** (`lib/`). It explains where each is declared, how they are linked into every build, why retained globals belong in `globals.st` , how duplicate names are caught, how to insert a library block into a POU, how library versions are locked for a validated system, and how to write and maintain your own library of reusable function blocks, including blocks with methods.

19.1 How shared declarations reach every POU

A Zeus application is compiled as one unit (Chapter 14, Section 14.1). For every build and download the IDE assembles, in this order:

#	Part	Files
1	Data types	<code>dut/*.st</code> , in dependency order (Chapter 15)
2	Global variables	<code>globals.st</code>
3	IO aliases	Generated from the aliases on the IO channels (Chapter 11)
4	Further global variable lists	<code>gv1/*.st</code> , in name order
5	Library	<code>lib/*.st</code> (or the default library), in file-name order
6	Project function blocks and functions	The enabled FB and FC POU's
7	Programs	The PROGRAMs of the task call list (Chapter 18)

Everything is then **one name space**. A global declared in any list is visible, by its bare name, in every POU; a library block is a type every POU can instantiate. Nothing needs to be imported or re-declared.

When the compiler reports an error inside a shared file, the message names that file and line (`globals.st` , `gv1/Utilities.st` , `lib/20_control.st`), and clicking it in **Messages** opens the right editor.

19.2 Global Variables (`globals.st`)

Open **Application** → **Global Variables**. The tab shows `globals.st` in an editor, with:

Control	Does
Save	Writes <code>globals.st</code>
Check	Saves and compiles the project, reporting errors in Messages
Insert template	Inserts the <code>VAR_GLOBAL</code> / commented <code>VAR_GLOBAL RETAIN</code> scaffold
Rename variable... (F2)	Renames the global under the caret in its declaration and in every POU that uses it
Live values on the controller	Below the editor: the online table of every global, with write, prepare and force (Chapter 22)

An empty `globals.st` opens with the template already in the editor (not saved until you save it):


```

VAR_GLOBAL
  (* Project-wide variables. Visible to every POU. *)

END_VAR

(* Values that must survive a restart. Uncomment to use.
...
VAR_GLOBAL RETAIN
  batch_total : DINT;
END_VAR
*)

```

A typical `globals.st` :

```

VAR_GLOBAL
  (* Project-wide variables. Visible to every POU. *)
  Plant_Running : BOOL;
  E_Stop_OK     : BOOL;
END_VAR

VAR_GLOBAL RETAIN
  Batch_Number  : DINT;
  Pump1_Hours   : DINT;
END_VAR

```

Rules:

- `globals.st` holds **only** `VAR_GLOBAL` blocks. A POU written here would be prepended to every other POU and collide with itself.
- Globals may be of any type: elementary, arrays, DUT types (Chapter 15), and function-block types from the library or the project (`Line_Pid : PID` ;). A function-block instance declared as a global must still be **called**, and by one program only - two callers would run it twice per scan.
- `AT` addresses are allowed (`Estop_Chain AT %IX0.7 : BOOL` ;). Prefer aliases on the IO channels (Chapter 11), which the IDE binds for you.
- `VAR_GLOBAL CONSTANT` declares project-wide constants.
- After editing, **Download** for the change to reach the controller. Chapter 20 explains how a download with changed globals treats the values already in the controller.

Important: Do not declare a local variable with the same name as a global. The program uses the local, while the monitor, the variable tables, the ladder display, forces and writes from the IDE resolve the name to the **global** - the two disagree on screen and nothing looks wrong. The compiler warns ("x' shadows the global declared on line N"); **Where used** (Shift+F12) shows both declarations.

Tip: Avoid very short global names (`x` , `q` , `p` , `i`). Library blocks have locals of such names; each one then produces a shadowing warning, and the warnings bury the one that matters. A prefix or a descriptive name (`Line_Pv`) avoids it.

19.3 Further global variable lists (`gv1/`)

A large project organises its globals into several lists - one per area of the plant, one per interface.

To create a global variable list:

1. Right-click **Application** and choose **New global variable list...**
2. Enter the **Name** (default `Utilities`); letters, digits and underscore, starting with a letter or underscore.
3. Click **Create**.

The IDE writes `gv1/<Name>.st` with an empty `VAR_GLOBAL` block and an explanatory comment, and opens it. The **Global Variables** node now has children: **globals.st** (marked `default`) and each list. Right-click a list for **Open**, **Rename...** and **Delete** (both write a project backup first).

A list's tab has **Save**, **Check** and **Rename variable...** (F2), like `globals.st`. **Check** saves the list and compiles an empty program against the project; the status reads "saved — compiles" or "saved — has errors (see Messages)", with the diagnostics in **Messages**. The rename dialog states its scope before it changes anything - read it, and build the project after a rename in a list to confirm every use was updated.

```
VAR_GLOBAL
(* Utilities - compressed air and cooling water *)
Air_Pressure_Bar : REAL;
CW_Supply_Temp   : REAL;
Air_Low          : BOOL;
END_VAR
```

Lists organise; they do not scope. There is one symbol name space:

- There is no `Utilities.Air_Low` qualification. Use the bare name.
- A name declared in two lists - or in a list and `globals.st` - is **one duplicate symbol**, not two variables.

19.3.1 Duplicate detection

Duplicates are reported in two places:

- **On every save** of a list, the tab shows "Declared in more than one list:" with each duplicate name and **both** files (`E_Stop_OK` - `globals.st` and `gv1/Safety.st`), while both files are still in front of you.
- **On build**, the compiler refuses the second declaration: "'E_Stop_OK' is already declared on line 4".

The check reads the `VAR_GLOBAL` blocks of `globals.st` and every list; commented-out declarations are ignored.

Note: The Ladder editor's variable picker offers the globals of `globals.st` only; globals in further lists compile normally but must be typed by name there (Chapter 16).

19.4 RETAIN belongs in `globals.st`

`VAR_GLOBAL RETAIN` is valid in any list, but keep retained globals in `globals.st`. The reason is the retain layout signature (Chapter 33):

- The controller protects retained data with a signature hashed over every retained symbol's **name, type and position** in the retain image.
- The global lists are linked in **name order** after `globals.st`.

- Creating a new list whose name sorts before an existing list that holds retained values - or renaming a list - moves every retained symbol after it. The signature changes, the controller **refuses** the restore at the next start, and **every** retained value resets to its initial value.

Creating a list is something people do casually; losing a batch counter or a totaliser is not. With all retained globals in `globals.st`, their order changes only when somebody edits that file, which is when they are thinking about it.

The IDE supports this rule rather than enforcing it:

- A list that declares `VAR_GLOBAL RETAIN` shows "This list declares RETAINED values." with the explanation, on every save.
- Renaming such a list warns that it "has moved in the load order — the controller will refuse the next restore and reset retained values to zero".
- The `globals.st` tab warns that adding, removing, reordering or retyping any `RETAIN` declaration changes the layout and resets retained values. Add new retained globals **at the end** of the block, and finish the retained declarations before commissioning.

`VAR RETAIN` inside a PROGRAM is also retained (Chapter 14, Section 14.7), and the same layout rule applies to it; `RETAIN` inside a FUNCTION_BLOCK has no effect - retain the instance instead.

19.5 The function-block library

19.5.1 Where the library comes from

Situation	Library used
The project has a <code>lib/</code> folder with at least one <code>.st</code> file	The project's own <code>lib/*.st</code>
No <code>lib/</code> , or an empty one	The default library shipped with the IDE (<code>stdlib/</code>)

A **new project** gets its own copy of the default library in `lib/` when it is created, so it is self-contained: the project folder holds everything the controller runs, and a block can be modified in one project without changing any other. Blocks deleted from a project's `lib/` are never restored automatically.

The default library files:

File	Contents
<code>10_signal.st</code>	<code>SCALE</code> , <code>FILTER_PT1</code> , <code>RATE_LIMIT</code> , <code>ALARM_AN</code> , <code>TOTALIZER</code> , <code>AVERAGE_N</code> , <code>MEDIAN3</code> , <code>MINMAX_HOLD</code> , <code>DERIV</code> , <code>LOOKUP16</code>
<code>20_control.st</code>	<code>PID</code> , <code>RAMP</code> , <code>HYSTERESIS</code> , <code>PWM</code> , <code>DEADBAND</code> , <code>SPLIT_RANGE</code> , <code>THREE_POINT</code> , <code>RATIO</code>
<code>22_autotune.st</code>	<code>PID_AutoTune</code>
<code>30_equipment.st</code>	<code>MOTOR_DOL</code> , <code>VALVE_2POS</code> , <code>DEBOUNCE</code> , <code>BLINK</code> , <code>RUNTIME_HRS</code> , <code>PUMP_ALT2</code> , <code>TRIP_FIRST_OUT</code> , <code>SOFT_START</code>
<code>40_hvac.st</code>	<code>TempHumidityCoolReheat</code> , <code>AhuTempHumidity</code>
<code>50_s88.st</code>	<code>S88_PHASE</code> , <code>S88_MODE</code> , <code>S88_RESOURCE</code> , <code>S88_UNIT</code> , <code>S88_CONTEXT</code> (Chapter 28)
<code>70_iolink.st</code>	IO-Link helpers and <code>IOL_ISDU</code> , <code>IOL_ReadParam</code> , <code>IOL_WriteParam</code> (Chapter 12)

Appendix C documents every block. The IEC standard blocks (`TON`, `TOF`, `TP`, `CTU`, `CTD`, `CTUD`, `R_TRIG`, `F_TRIG`, `RS`, `SR`) are built into the runtime, not into the library (Appendix B).

Library files are linked in **file-name order** - hence the numeric prefixes. Function blocks may use blocks from a later file; a `TYPE` declared in a library file must come before the first type that uses it (Chapter 15, Section 15.4).

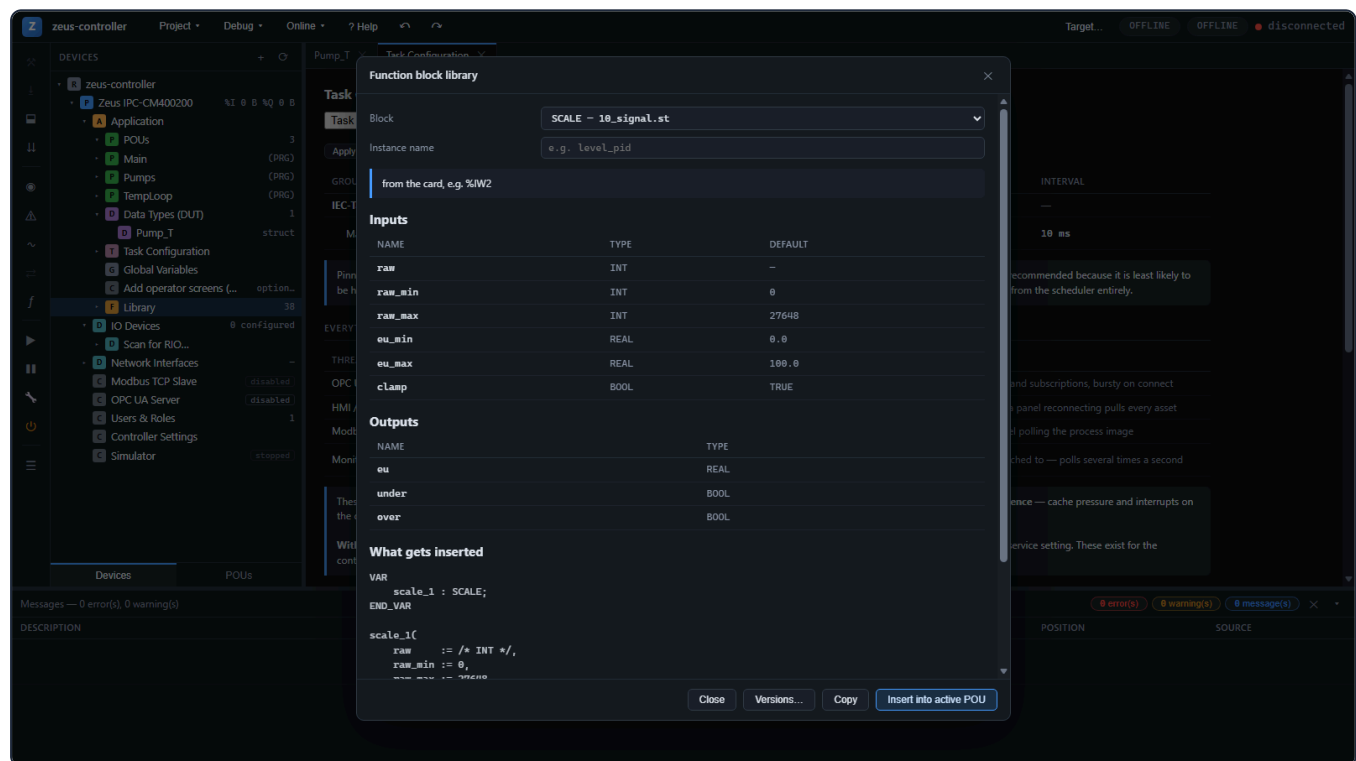
An unused library block costs compile time, not controller memory: it declares a type and generates no code until something instantiates it.

19.5.2 The Library node

Application → **Library** shows the number of blocks (with `default` when the default library is in use) and one child per function block, with its interface size (`6 in` · `3 out`). Double-click the node, a block, or the **f** button in the left rail to open the **Function block library** dialog.

Note: The Library node and dialog list the library's `FUNCTION_BLOCK`s. Functions in a library file (such as the IO-Link helpers `IOL_IN_INT` ...) are compiled and callable, but not listed.

19.5.3 Inserting a block from the library



The Function block library dialog with `SCALE` selected: its inputs with types and defaults, its outputs, and the declaration and call that will be inserted.

To insert a library block into a POU:

1. Open the POU and place the caret where the declaration and call should go.
2. Open the **Function block library** dialog (Library node, or **f** in the rail).
3. Choose the **Block** (listed as `NAME - file`).
4. Type the **Instance name** (for example `level_scale`). Without one, `<block>_1` is used.
5. Check the **Inputs** (name, type, default), **Outputs** (name, type) and **What gets inserted**.
6. Click **Insert into active POU**, or **Copy** to put the text on the clipboard.

What is inserted, at the caret:

```
VAR
    level_scale : SCALE;
END_VAR

level_scale(
    raw      := /* INT */,
    raw_min  := 0,
    raw_max  := 27648,
    eu_min   := 0.0,
    eu_max   := 100.0,
    clamp    := TRUE);
```

Move the `VAR` lines into the POU's declaration pane (a ST POU shows declarations and body separately; Chapter 13), replace every `/* type */` placeholder with a variable, and delete the inputs you leave at their default. Read outputs as `level_scale.eu`.

Both parts are needed: an instance that is never called runs nothing, and a call without a declared instance does not compile.

The note at the top of the dialog is the first comment inside the block's source (Section 19.6.2).

In Ladder, place library blocks with the **Block** part (Chapter 16, Section 16.6.4).

19.5.4 Library versions and the lock

A validated system has to be able to answer "which version of the PID ran this batch?", and a folder of `.st` files cannot. So:

- A library file may declare its name and version in a header comment within its first 40 lines:

```
(* @library zeus-control @version 1.4.0 *)
```

- The project can be **locked** to its library: `library.lock.json` records, per file, the declared name and version, the SHA-256 of the content (line endings normalised) and the size, with who locked it, when and why.

The hash is what makes the version trustworthy. A version number is a claim; an edited file that kept its number is exactly the case worth catching.

To view and lock the library:

1. Open the **Function block library** dialog and click **Versions....** The **Library versions** dialog opens.
2. Read the state of each file:

State	Meaning
unchanged	Same content as when locked
CHANGED	The content differs from the lock, whether or not the version number moved. The code that will be downloaded is not the code that was validated.
added	A file that was not there when the lock was taken
REMOVED	A file in the lock that is no longer there - programs using its blocks will not compile ("unknown type 'PID'")
not locked	No lock has been taken yet

The table columns are **File**, **Library** (the declared name), **Version** (now), **Locked at** (the version recorded in the lock) and **State**. When anything is CHANGED, added or REMOVED, the dialog says: "The library is not what this project was locked to."

3. Enter **Why are you locking this?** (for example "validated for the 2026-09 batch").
4. Click **Lock this library** (or **Lock again**).

Locking is deliberate: it records a decision, with a name and a reason. The lock never updates itself when a file changes.

Adding default files. When the shipped default library contains files the project does not have - a newer IDE release that added the IO-Link blocks, for example - the **Library versions** dialog says so and offers **Add to this project**. It copies only the missing files; files already in `lib/` are never overwritten, because a silently replaced PID is a retuned loop on a real machine. On a locked project the new files then show as **added** until you lock again.

Important: A newer library file may call runtime functions that an older controller runtime does not have. Update the controller runtime to the IDE's release before a project uses newly added library files (Chapter 33).

19.6 Writing your own library

Put a block in the **library** when it is used by several projects or several machines and should change only under control: site standards, vendor device blocks, validated algorithms. Put it in the **project** (a Function Block POU) when it belongs to this machine.

19.6.1 Creating a library file

The IDE has no editor for library files; they are plain text files in the project's `lib/` folder.

To add a library file:

1. Open the project folder (Chapter 5) and its `lib/` subfolder.
2. Create a file with a name that sorts after the files it depends on, for example `80_site.st`. Use UTF-8 or ASCII text.
3. Write the header comment, then the blocks (Section 19.6.3).
4. Build any POU. The compiler reads `lib/` afresh on every build and download, so errors appear at once, reported against `lib/80_site.st`.
5. Restart the IDE (or reopen the project) to see the new blocks under **Library** and in the dialog; the list is read when the project loads.
6. Open **Versions...** and lock the library again, with a note.

To share it with other projects, copy the file into their `lib/` folders and lock each.

Note: A project that still uses the default library (no `lib/` of its own) gets its own `lib/` only when you add one. Creating `lib/` with just your file would **replace** the default library for that project. Copy the default files in first (**Versions...** → **Add to this project**, or copy them from the IDE's `stdlib` folder).

19.6.2 Conventions that make a block usable

- **Header:** `(* @library <name> @version <x.y.z> *)` in the first 40 lines. Bump the version on every change.

- **One-line summary** as the first comment **inside** the block, directly after the `FUNCTION_BLOCK` line. The library dialog shows it above the pin tables. (A comment above the `FUNCTION_BLOCK` line is for readers of the file; the dialog does not show it.)
- **A comment on every input and output**, on the same line as its declaration.
- **Defaults** (`:= value`) on every input that has a sensible one. The dialog shows them, and the caller need not pass them.
- **One `VAR_INPUT` and one `VAR_OUTPUT` block.** The dialog lists the first block of each kind.
- **No globals, no `AT` addresses** inside a library block: everything through the interface, so the block works on any machine.
- **`dt_ms` as an input** for anything that integrates or differentiates, never a hard-coded scan time (Chapter 18).
- **Unique names.** Block names share one name space with every POU and DUT; prefix site blocks if there is any risk of a clash.

19.6.3 Example: a duty/standby pump block with methods

```
lib/80_site.st :
```

```

(* @Library acme-site @version 1.2.0 *)
(* Site standard blocks for ACME plant.
 *
 * Owner: controls engineering. Change only through the change procedure;
 * every change bumps @version and is re-locked in each project.
 *)

(* Duty/standby pair with automatic changeover on a fault and a weekly swap. *)
FUNCTION_BLOCK DUTY_STANDBY
(* Duty/standby pump pair: changeover on a fault, periodic duty swap. *)
VAR_INPUT
    demand      : BOOL;                (* run one pump                *)
    a_fault      : BOOL;                (* pump A tripped              *)
    b_fault      : BOOL;                (* pump B tripped              *)
    swap_now     : BOOL;                (* rising edge swaps duty      *)
    swap_after   : TIME := T#168h;      (* automatic swap, 0 = never    *)
END_VAR
VAR_OUTPUT
    run_a        : BOOL;
    run_b        : BOOL;
    duty_is_b    : BOOL;
    no_pump      : BOOL;                (* demand but both faulted     *)
END_VAR
VAR
    swap_edge    : R_TRIG;
    duty_timer    : TON;
END_VAR

METHOD Swap
    (* Change duty. Called from the body only. *)
    duty_is_b := NOT duty_is_b;
END_METHOD

METHOD Pick : BOOL
    (* TRUE = run B this scan. *)
    IF duty_is_b THEN
        Pick := NOT b_fault OR a_fault;
    ELSE
        Pick := a_fault AND NOT b_fault;
    END_IF;
END_METHOD

    swap_edge(CLK := swap_now);
    duty_timer(IN := demand AND swap_after > T#0s AND NOT duty_timer.Q,
               PT := swap_after);
    IF swap_edge.Q OR duty_timer.Q THEN
        Swap();
    END_IF;

    no_pump := demand AND a_fault AND b_fault;
    run_b := demand AND NOT no_pump AND Pick();
    run_a := demand AND NOT no_pump AND NOT run_b;
END_FUNCTION_BLOCK

```

A program that uses it, with `globals.st` and `gvl/Utilities.st` as in Sections 19.2 and 19.3:


```

PROGRAM CoolingWater
VAR
    Demand_Sw  AT %IX6.0 : BOOL;
    P1_Trip    AT %IX6.1 : BOOL;
    P2_Trip    AT %IX6.2 : BOOL;
    Swap_PB    AT %IX6.3 : BOOL;
    P1         AT %QX6.0 : BOOL;
    P2         AT %QX6.1 : BOOL;
    Pumps      : DUTY_STANDBY;
END_VAR
    Pumps(demand := Demand_Sw AND E_Stop_OK,
          a_fault := P1_Trip,
          b_fault := P2_Trip,
          swap_now := Swap_PB);
    P1 := Pumps.run_a;
    P2 := Pumps.run_b;
    Air_Low := Air_Pressure_Bar < 5.5;
END_PROGRAM

```

E_Stop_OK comes from `globals.st`, Air_Low and Air_Pressure_Bar from `gv1/Utilities.st`, DUTY_STANDBY from `lib/80_site.st` - all by bare name.

19.7 Methods

A function block may contain **methods**: named sub-routines that work on the block's own variables (Chapter 14, Section 14.14 has the language rules).

To add a method to a function block POU:

1. Right-click the function block in the tree and choose **Add method...**
2. Enter the **Method name** (default `Operation`); it must be an identifier not already used by a method of this block.
3. The IDE inserts the method skeleton after the declarations and any existing methods, saves the block and opens the method in **its own tab**.

The function block's own tab hides its `METHOD` blocks, the way the method list of a function block is shown separately in other IEC environments; each method is edited only in its tab, reached from the tree under the block. Adding, renaming and deleting a method saves the block. In a library file, which has no IDE editor, write the methods directly in the text, as in Section 19.6.3.

What methods are for, in this version:

- A method is called **by its bare name from inside the same block** - from the body or from another method:
`Swap();`, `run_b := ... Pick();`.
- A method **cannot be called from outside** the block: `Pumps.Swap();` is rejected ("this expression has no effect - did you mean '=' ?"). To trigger a method from a program, give the block a command input (as `swap_now` above) and call the method from the body when the input is set.
- A method reads and writes the instance's inputs, outputs and locals directly. Its own `VAR_INPUT` / `VAR` are fresh on every call, and its declared initial values are not applied.
- A return value, if declared, must be elementary; `VAR_IN_OUT` is not allowed on a method.

Methods therefore structure a large block's body into named, readable pieces; they are not an external interface.

19.8 Summary

Question	Answer
Where do I declare a variable every POU can use?	<code>globals.st</code> , or a list under <code>gv1/</code>
Can two lists declare the same name?	No - one name space; reported on save and on build
Where do retained globals go?	<code>globals.st</code> , new ones at the end of the block
Where do shared function blocks go?	<code>lib/</code> for site or vendor standards; a Function Block POU for this machine
How do I prove which library version ran?	Versions... → Lock this library ; <code>library.lock.json</code> holds a SHA-256 per file
How do I get newly shipped library files into an old project?	Versions... → Add to this project
Can I call a method from a program?	No - through a command input of the block

ZEUS PLC IDE · OPERATION MANUAL

Part V — Running and debugging

20. Build, download and online change

This chapter explains how a project becomes the program a controller runs: what the compiler checks, what is linked into the single unit that is sent, the four download commands and what each one does to a running machine, how online change carries the machine's data across a download, and how to prove afterwards what the controller is running. Read it before the first download to a real machine, and again before the first download to a machine that is in production.

20.1 Overview

Zeus has one compiler. The IDE does not contain a second, simplified one: **Build** runs the same Structured Text compiler that the controller runs (`zeus-check.exe` in the Windows package, `zeus-plcd --check` on Linux), and on **Download program** the controller compiles the source again itself before it replaces anything. A program that does not compile on the controller is refused and the running program is not touched.

What travels to the controller is source text, not machine code. The controller stores that source as its *boot application* and records its SHA-256. This is what makes it possible to ask a controller, at any time, what it is running (section 20.9).

Stage	Where	What happens
Check	IDE (declaration editors)	Saves the file and compiles its declarations with an empty test program
Build	IDE	Compiles the active POU together with everything it depends on
Build All	IDE	Builds every enabled POU, one at a time, the same way
Download program	Controller	Compiles and binds the whole application, then swaps it in (online change)
Download hardware	Controller	Replaces the controller configuration; the runtime restarts

20.2 Check, Build and Build All

20.2.1 Check

The declaration-only editors — **Global Variables**, a global variable list (GVL) and a data type (DUT) — have a **Check** button next to **Save**. A file that holds only declarations is not a compilation unit on its own, so **Check**:

1. saves the file, then
2. compiles the project's declarations together with an empty test program (`PROGRAM __gv1_check` or `PROGRAM __dut_check`), and
3. shows the result next to the button (**saved — compiles** or **saved — has errors**) and lists any diagnostics in the Messages panel.

Use **Check** while you are writing types and globals, before any program uses them.

Note: The **Check** button of a global variable list (a GVL under **Global Variables**, not `globals.st` itself) works like the DUT tab's: it saves the list and compiles an empty program against the whole project. Its errors are listed in the Messages panel against the GVL file and line.

20.2.2 Build (F7)

Debug → **Build (F7)**, or the  button on the action rail) compiles the **active POU** — the editor tab in front. The POU is linked with everything it can refer to (section 20.3) and compiled from scratch; there is no incremental cache.

- Diagnostics appear in the Messages panel with file, line and column. The parser reports every independent mistake in one pass, not only the first.
- An error in `globals.st`, a GVL, a DUT, a library block or the generated IO-alias block is reported against **that** file and line, not against the POU you built. Click the message to open the file at the line (Chapter 25).
- Lines with errors and warnings are marked in the editor: the line is tinted (red for an error, amber for a warning) and its number in the gutter is shown in bold colour.
- When the active tab is a METHOD tab, the method is spliced into its function block and the whole block is compiled; messages refer to the method's own line numbers.
- A successful build ends with a note such as `ok - 10 symbol(s), 468 byte(s) of code, 248 byte(s) of data`.
- If the active tab is not a POU (a device page, Task Configuration, ...), **Build** does nothing.

If the compiler cannot run at all (for example the compiler binary is missing), the message says **The build did not complete** and gives the reason. That is a tool problem, not a program problem.

20.2.3 Build All (Ctrl+F7)

Debug → **Build All (Ctrl+F7)** builds every POU in the project, each one linked exactly as **Build** links it. The toast summarises the result, for example `Build All: 7 ok, 1 failed, 2 skipped`.

A POU can be left out with the POU's context menu entry **Disable**; the tree then shows it as *(disabled)*, and **Build All** lists it as **disabled - skipped** so that nothing is silently missing. A disabled POU is also left out of every download: a disabled function block or function is not linked, and a disabled program is not taken from the task call list (Section 13.6.5).

Build All compiles several POUs at the same time (one fewer than the PC's cores, at most six; two when the compiler runs on the controller), so it takes a few seconds even on a project of a few dozen POUs. An open POU with unsaved edits is built as it stands in the editor.

Important: While a **Variables** page or the **Global Variables** live table is the visible view, **F7**, **Ctrl+F7** and **Alt+F7** act on that table instead (Force, Write, Release — Chapter 22) and do not build. Build from a POU editor tab, or use the **Debug** menu.

20.2.4 Clean and Clean all

Command	Effect
Debug → Clean	Clears the Messages panel. Nothing else is cached, so there is nothing else to discard.
Debug → Clean all	Clears the Messages panel and runs Build All .

Neither command touches the data on the controller.

20.3 What is linked into one unit

The controller runs one compiled unit. The IDE server assembles it from the project files in a fixed order, so that everything is declared before it is used:

Order	Part	Source in the project
1	Data types	every file under <code>dut/</code> (STRUCT, ENUM, alias types)
2	Global variables	<code>globals.st</code> , with IO aliases bound to their channel addresses (Chapter 11)
3	IO aliases	a generated <code>VAR_GLOBAL</code> block for aliases that are not declared in <code>globals.st</code>
4	Global variable lists	every file under <code>gv1/</code>
5	Library	every block under <code>lib/</code> (Chapter 19)
6	Project blocks	every enabled FUNCTION_BLOCK and FUNCTION in <code>pou/</code> , ordered so a block comes after the blocks it uses
7	Programs	the program(s) being built or downloaded — see below

Parts 1 to 6 are called the *prelude*. The IDE keeps a map of which prelude lines came from which file, which is how a compiler message on line 612 of the unit is shown as line 7 of `gv1/Recipes.st`.

What goes in part 7 depends on the command:

Command	Programs in the unit
Build	the active POU only (from the editor buffer, saved or not)
Download program	every PROGRAM in MainTask's call list, in call-list order (the bundle)

For a download, the program open in the editor contributes its **editor buffer**; the other programs of the call list are read from disk. Line endings are normalised to LF before the unit is hashed, so a file saved with CRLF and the same file open in the editor are the same program.

20.3.1 The 256-POU limit

A unit may contain at most **256 POUs**. Every PROGRAM, FUNCTION_BLOCK, FUNCTION and METHOD in the unit counts — including every block in `lib/`, whether or not the program uses it. A unit that exceeds the limit fails with `too many POUs in one file (max 256)`.

If you reach the limit, remove library blocks the project does not use (Chapter 19) or exclude unused POUs from the build.

20.3.2 Example: an application of two programs

`globals.st` :

```
VAR_GLOBAL
    g_TankLevel   : REAL;      (* % full, written by Fill *)
    g_FillCount   : DINT;      (* completed fills *)
END_VAR
```

pou/Fill.st :

```
PROGRAM Fill
VAR
    Inlet_Open : BOOL;
    Level_Hi   : BOOL;
    Done_Edge  : R_TRIG;
END_VAR
IF Inlet_Open THEN
    g_TankLevel := g_TankLevel + 0.5;
END_IF;
Level_Hi := g_TankLevel >= 95.0;
Inlet_Open := NOT Level_Hi;
Done_Edge(CLK := Level_Hi);
IF Done_Edge.Q THEN
    g_FillCount := g_FillCount + 1;
    g_TankLevel := 0.0;
END_IF;
END_PROGRAM
```

pou/Report.st :

```
PROGRAM Report
VAR
    LastCount : DINT;
    Changed   : BOOL;
END_VAR
Changed := g_FillCount <> LastCount;
LastCount := g_FillCount;
END_PROGRAM
```

With MainTask's call list set to `Fill.st` , `Report.st` (Chapter 18), a download from either tab sends one unit: `globals.st` , then `Fill` , then `Report` . Every scan the controller runs `Fill` and then `Report` , and both see the same `g_FillCount` .

20.4 The download commands

All transfers are in **Online** → **Download & Upload** (the group expands in place). The most used ones are also on the action rail at the left edge of the window.

Command	Key / rail	What is sent	Effect on the machine	Role
Download program	F8, ↓	The application (section 20.3) and its file map	Scan paused for about 50 ms while the program is swapped; IO keeps exchanging	engineer

Command	Key / rail	What is sent	Effect on the machine	Role
Download hardware	🔲	The controller configuration: IO devices, rack, process image, task, users, services	⚠️ STOPS THE CONTROL LOOP. The runtime restarts; outputs go to their fail-safe state for a few seconds	engineer
Download all	⇓	Hardware, then the program	⚠️ STOPS THE CONTROL LOOP (hardware step)	engineer
Download screens	—	The operator screens (Chapter 32)	None on the control loop	engineer

The same program download is also offered in three other places:

- the POU context menu in the tree: **Download to controller** (F8);
- the **Download this POU** button at the bottom of a POU's **Variables** page;
- the Ladder editor, through **F8** while a ladder POU is the active tab.

Every one of them calls the same function: they differ only in which POU they start from, and section 20.5 explains what that changes.

WARNING: Download sends to whichever target is connected. Check the target badge in the header (for example **SIMULATOR**, or the controller's name) before every download. The Target... dialog selects it (Chapter 6).

20.4.1 Download program (F8)

To download the program:

1. Save your work (**Ctrl+S**). The download saves the POU it starts from if that tab has unsaved edits, and saves `globals.st` if it is open and changed, but it does not save the rest of the project. Other programs of the call list are taken from disk.
2. Make sure you are logged in with the **engineer** role (Chapter 6). If the controller has accounts and you are not logged in, the login dialog opens first.
3. Press **F8**, or choose **Online** → **Download & Upload** → **Download program**.
4. Read the confirmation — *This REPLACES the running program...* — and confirm.
5. Read the Messages panel. A successful download adds two notes:

```
Downloaded Main: 212 symbol(s), 14880 B code, 3912 B data. Controller is RUNNING.
Online change - variable state carried over: 210 kept, 2 reset, 1 dropped (gone: OldFlag).
```

The second line is the transfer report (section 20.6). It is a warning, not a note, when anything was dropped.

What the controller does, in order:

1. Compiles the source. On any error it answers `the program did not compile`, lists the diagnostics, and changes nothing.
2. Binds the program to the process image. A `%I`, `%Q` or `%M` address outside the configured image is refused with one diagnostic per address, and nothing changes. This happens when the program is written for IO that has not been sent with **Download hardware** yet.
3. If program signing is enforced (Chapter 34), checks the signature and version.

- 4. Pauses the scan, copies the old program's data into the new one (online change), re-applies the per-program task rates, swaps the programs and resumes the scan.
- 5. Writes the source to its boot application, together with the file map, so that the same program comes back after a restart (section 20.8).
- 6. Records the download in its audit log: accepted or rejected, program name, symbol count, code size, and whether it was an online change or a cold start.

WARNING: After an accepted download the controller is **RUNNING**. The download resumes the scan even if the controller was PAUSED, and a download clears a FAULT state. A program that compiles starts controlling the machine as soon as it is swapped in. If the machine must not start, keep it safe by other means before you download.

The IDE translates the controller's diagnostics back to project files the same way a build does, so a rejected download is clicked through exactly like a failed build.

20.4.2 Download hardware

Download hardware sends the project's controller configuration — the IO device tree, the rack and process-image sizes, the task settings, the accounts, and the settings of the OPC UA server, the EtherNet/IP tag server, the Modbus TCP slave and the operator-screen server.

Without a stop where possible. The IDE first compares the project's configuration with the one the controller is running, section by section:

What changed	How it is applied
OPC UA server, operator-screen (web) server, Modbus TCP slave, EtherNet/IP tag server	Without stopping: the runtime rebuilds only those services; the scan, the program and the fieldbus keep running. OPC UA / Modbus / screen clients reconnect.
The task's call list, disabled POU's, screen settings kept by the IDE	Without stopping: the runtime does not use them; the controller simply adopts the new file.
IO devices, rack, process image, task timing, monitor, accounts and security, alarms, log, clock, LEDs, redundancy	A restart : the runtime reads these only at start.

When nothing in the last row changed, the hardware download asks nothing and the toast reads "Settings applied — the controller kept running". Otherwise the confirmation below warns of the restart, and the Messages panel says which sections required it. A controller running a runtime older than 0.1.0.9 cannot apply settings while running; it is restarted as before until its runtime is deployed.

When a restart is needed:

To download the hardware configuration:

- 1. Put the machine in a state in which a stop of a few seconds is safe.
- 2. Choose **Online** → **Download & Upload** → **Download hardware** (or the  rail button).
- 3. Read the confirmation. For a controller it names the number of enabled IO devices and says that the control loop stops and outputs go to their fail-safe state; for the simulator it says that no hardware is touched.
- 4. Confirm. The Messages panel lists each step (configuration written, service restarted, and so on) with **ok** or the reason it failed.

- Wait for the connection indicator in the header to come back. The IDE logs the session back in automatically after the restart.

A **program** download (F8) on its own never stops the controller - it is an online change (Section 20.6). When the controller's settings also differ from the project, **Download program** sends them with it: without a question when they apply while running, otherwise after asking ("OK - settings AND program, the runtime restarts; Cancel - the program only").

What a hardware download does **not** do:

- It does not change the runtime software itself. Installing a new runtime version is **Target... → Deploy** (Chapter 6).
- It does not send a new program. The runtime restarts onto its **boot application**, that is, the program that was last downloaded.

A hardware download to the **simulator** rebuilds the simulator from the project configuration and restarts it (Chapter 26).

20.4.3 Download all

Download all runs **Download hardware** and then **Download program**, in that order and never the other way round: the hardware step restarts the runtime, which would discard a program sent before it.

- Choose **Online → Download & Upload → Download all** (or ⇓).
- Confirm the single dialog, which counts the enabled IO devices and names the active POU.
- The IDE sends the hardware, waits up to 20 seconds for the controller to come back and the session to be logged in again, then downloads the program from the active POU (section 20.5).

If the program step fails or is cancelled, the Messages panel says so explicitly: *The hardware was downloaded, but the PROGRAM was not. The controller restarted onto its previous program - download the program again.* The header then shows **PROGRAM CHANGED**.

Use **Download all** for the first download to a new controller and after any change to the IO devices, the rack or the task.

20.4.4 Download screens

Download screens copies the operator screens (HMI) to the controller so that its own web server can serve them. It does not affect the control loop. See Chapter 32.

20.5 Which program is downloaded

A download always sends **the application**, not a single file. Which program the download starts from decides only details:

Where you start	What is sent
A PROGRAM that is in MainTask's call list	The whole call list, with this program's editor buffer
A PROGRAM that is not in the call list (and the call list is not empty)	You are asked first — see below
A FUNCTION_BLOCK or FUNCTION tab	The whole call list; the block is part of the prelude
A METHOD tab	The method is saved into its block, then as for the block

Where you start	What is sent
No POU tab in front (a device page, the welcome page)	The call list, starting from its first program
A project with an empty call list	The active program, or the first enabled PROGRAM in the project

When you download from a PROGRAM that the task does not call, the IDE stops with the dialog **<name> is not called by any task**, because a program nobody calls would be compiled, sent and never run:

Button	Result
Add to MainTask and download	Appends the program to the call list (it then runs every scan), saves the project configuration, and downloads
Download the call list	Downloads the application without this program
Cancel	Nothing is sent

Note: Changing the call list with **Add to MainTask and download** changes the project configuration. The controller runs the new list at once, because the bundle it receives contains the program; the task settings themselves (period, priority) still need **Download hardware**.

The tree shows the result. Every program the controller is executing carries a **RUNNING** badge; enabled programs it is not executing are dimmed.

20.6 Online change

A download keeps the machine's data. Online change is always on when you download from the IDE: there is no command in the IDE that downloads with a cold start.

20.6.1 What is kept, reset and dropped

The controller compares every variable in static storage — program variables, globals, function-block instances and their members — between the old program and the new one:

Outcome	Condition	Result
Kept	Same name, same type; for a structure or array also the same total size and member count	The running value is copied into the new program
Reset	New name, or the type changed (INT to REAL, INT to WORD, a structure whose size or member count changed)	Starts at its declared initial value
Dropped	The old program had it; the new one does not	Gone; the first few names are listed in the report

Matching is by **name and type, never by address**. Offsets shift as soon as a declaration is added anywhere, and copying by offset would move one variable's value into another.

Standard function blocks are handled specially:

- A timer or counter instance whose name and block type survive keeps its internal state: a TON that had counted 40 s of a 90 s delay continues from 40 s.

- Its **preset** (PT for timers, PV for counters) is taken from the **new** program text, so shortening a delay takes effect on the next scan.
- An SFC chart stays in the step it was in, because the step flags are ordinary variables that are carried over (Chapter 17).

Variables of a FUNCTION (its locals and VAR_TEMP) are not carried: they exist only while the function runs.

20.6.2 Example

Version 1, running:

```
PROGRAM Mixer
VAR
    Run      : BOOL;
    Speed_Sp : INT := 1200;
    MixTimer : TON;
    Batches  : DINT;
    OldFlag  : BOOL;
END_VAR
MixTimer(IN := Run, PT := T#90s);
IF MixTimer.Q THEN
    Batches := Batches + 1;
    Run := FALSE;
END_IF;
END_PROGRAM
```

Version 2, downloaded while a batch is 40 s into mixing:

```
PROGRAM Mixer
VAR
    Run      : BOOL;
    Speed_Sp : REAL := 1200.0; (* was INT: type changed -> reset *)
    MixTimer : TON;
    Batches  : DINT;
    Agitate  : BOOL;          (* new -> starts at its initial value *)
END_VAR
(* OldFlag deleted -> dropped, named in the report *)
MixTimer(IN := Run, PT := T#60s); (* new preset takes effect at once *)
Agitate := Run AND NOT MixTimer.Q;
IF MixTimer.Q THEN
    Batches := Batches + 1;
    Run := FALSE;
END_IF;
END_PROGRAM
```

Result:

Variable	Outcome	Value after the download
Run	kept	TRUE — the batch continues
Batches	kept	unchanged count
MixTimer	kept	elapsed time 40 s, preset now 60 s: finishes 20 s later
Speed_Sp	reset	1200.0 (initial value), whatever the operator had set

Variable	Outcome	Value after the download
Agitate	reset (new)	FALSE, then computed on the first scan
OldFlag	dropped	listed in the report

The report reads, for example, `Online change - variable state carried over: 4 kept, 2 reset, 1 dropped (gone: OldFlag).`

Important: Read the transfer report after every download to a running machine. It is the only place that says what the machine kept. A setpoint that was reset because its type changed runs at its initial value from that scan on.

20.6.3 Retained variables and the retain file

RETAIN variables are carried by online change like any other variable. Their copy on disk — the retain file — follows the new program: its layout signature covers the name, type and position of every RETAIN variable **and the size of the program's data area**, so almost any download that adds or removes a declaration changes it.

- The retain saver writes the new layout on its next save (every 30 s by default, `process_image.retain_interval_ms`) and always on a planned stop: **Restart runtime, Download hardware, Shut down controller**, and the power-fail save of the controller carrier.
- If the controller loses power **between** the download and that first save, the retain file on disk still has the old layout. At the next start the runtime refuses to restore it and every retained value starts at its initial value; the controller log records `retain: RESTORE REFUSED - the program's retained variables changed since this image was written`.

So: after a download that changes declarations on a machine whose retained values matter, wait at least the retain interval before you remove power, or restart the runtime deliberately. See Chapter 33 for the retain mechanism in full.

20.7 Upload from controller

Online → **Download & Upload** → **Upload** (also **Upload from controller...** in a POU's context menu) reads back the program the controller is running. Any role may upload.

To upload:

1. Choose **Upload**. A confirmation shows the program name, size, when and from which address it was downloaded, and the start of its SHA-256.
2. Confirm. The IDE saves:
 - the linked unit as a read-only snapshot, `uploads/uploaded_<program>.st`;
 - the controller's current variable values as `data/<time>-upload_<program>.data.csv` (section 23.6).
3. A dialog shows the uploaded text. If the program carries a file map, the dialog offers **Replace this project...**

The snapshot is kept out of `pou/` on purpose: it is the whole linked unit — globals, library, blocks and programs in one file — and inside `pou/` it would be linked into every build and duplicate every definition.

20.7.1 The manifest (file map)

Every download from the IDE sends, with the source, a small file map (the manifest): the program name, the number of prelude lines, and the list of files the prelude was built from with their line ranges. The controller stores it opaquely next to its boot application and returns it on upload. Without it an upload could only hand back one large text; with it, the IDE can cut the text back into the files it came from.

A program downloaded by an IDE version that did not send a map uploads with the note *This program has no file map* and can only be kept as a snapshot. Download it once from the current IDE and the next upload can rebuild the project.

20.7.2 Replace this project from the controller

Replace this project... rebuilds the open project from what the controller runs.

1. In the upload dialog, choose **Replace this project...**
2. The confirmation lists every file that will be overwritten and says whether `controller.json` (hardware and settings) is included. From a real controller the configuration it reports is included; from the **simulator** it is not, because the simulator runs a transformed configuration (every device replaced by the sim driver) that is not the project's.
3. Confirm. The whole project is first copied to `backups/<time>-before-upload/` (POUs, library, DUTs, GVLs, `globals.st`, `controller.json`, `data/`); the Messages panel names the backup directory.

Paths in a manifest are confined to the project directory. A manifest that tries to write outside it is refused as a whole.

20.8 The boot application

The controller keeps the last accepted program as its boot application (`/var/lib/zeus-plc/boot.st` by default), with the file map and, when used, the program signature beside it. At every start the runtime compiles and binds the boot application and runs it; the scan starts in RUNNING.

If the boot application does not compile, does not fit the configured process image, or fails its signature check, the runtime **still starts** — with no program. IO keeps exchanging, the monitor stays reachable, and the controller log explains why, so that a corrected program can be downloaded. A bad download can therefore never take a controller off the network.

Note: The simulator does not keep a boot application between starts: after **Start / Restart** of the simulator, or a hardware download to it, download the program again (Chapter 26).

20.9 Is the controller running this project?

20.9.1 The sync badge: IN SYNC / PROGRAM CHANGED

The header shows, all the time, whether the controller is running what is open in the IDE. It is a state, not a command:

Badge	Meaning
IN SYNC	The controller runs exactly this program (the whole call-list bundle) and this hardware configuration

Badge	Meaning
PROGRAM CHANGED	The program on screen — including unsaved edits — is not the one the controller runs
HARDWARE CHANGED	The controller runs a different configuration from this project
PROGRAM + HARDWARE CHANGED	Both
⚠ NOT CHECKED — LOG IN	The controller has accounts and this IDE is not logged in, so nothing could be compared. Click to log in
⚠ NOT CHECKED	The comparison could not be made; hover for the reason. Not the same as a match
<i>(no badge)</i>	No controller is connected

How it works:

- **Program:** the IDE links the bundle exactly as a download would and asks the controller to compare its SHA-256 with the source it is running. The comparison happens on the controller. When the editor has unsaved edits, the buffer is compared; otherwise the saved file.
- **Hardware:** the IDE hashes the configuration it would send and compares it with the hash of the configuration the runtime loaded.
- It is re-checked about 0.7 s after you stop typing, after every download and save, on connecting, and every 30 s as a backstop (not while the IDE window is hidden).

Clicking the badge opens the comparison (section 20.9.3).

Important: Live values are always values of the program the **controller** is running. While the badge says **PROGRAM CHANGED**, the values beside a variable you have just added, renamed or retyped do not belong to your edit.

20.9.2 Status dots in the tree

Each POU in the tree carries a dot that compares that file with the controller's source:

Dot	Meaning
green	identical to what the controller runs
orange	changed — unsaved edits, or saved but not downloaded
grey	not on the controller at all
none	offline, not logged in, or not yet checked

The dots need a logged-in session. They are refreshed on connect, download, save and when the controller's program hash changes — not on every status poll.

20.9.3 Compare

Online → **Download & Upload** → **Compare...** (also the ⇌ rail button, or a click on the sync badge) opens **IDE compared with the controller**. It lists three groups:

Group	Items
Different	Items that differ, with the value in the project, the value on the controller and what that means
Identical	Items that match
Could not be checked	Items the controller could not answer; treat the controller as unverified

The items compared are:

Item	Compared by
program	SHA-256 of the linked bundle (or of the single program, for a project with fewer than two programs in the call list), computed on the controller
device name	presence of each enabled IO device, and its %I / %Q byte count when the project fixes it
scan period	the task period in the project against the controller's

Going online with **Monitor** runs the same comparison automatically and opens the dialog only when something differs, with the warning that live values come from what is listed under **On the controller**.

20.9.4 Verify

Debug → **Verify** answers one question for the POU in front: is the controller running exactly this source? It links the open POU (including unsaved edits) as a download would, sends it to the controller, and the controller compares the hash with the source it recorded.

- **MATCH** — logged as a note with the program name, download time and SHA-256.
- **MISMATCH** — logged as an error with both hashes. Upload to see what is actually on the machine.

Verify needs a text POU tab; for a ladder POU use **Compare...** or the sync badge.

20.10 Procedure: first download to a new controller

1. Build the project with **Build All** and clear every error.
2. Rehearse on the simulator: select **Simulator (local)** with **Target...**, run **Download all**, go online and check the program (Chapter 26).
3. Select the controller with **Target...** and confirm that the header shows its name.
4. Log in as an engineer.
5. Make sure the machine is safe to start: outputs will be driven as soon as the program is swapped in.
6. **Online** → **Download & Upload** → **Download all**. Confirm.
7. Watch the Messages panel: every hardware step `ok`, then `Downloaded ...` and the transfer report (a first download is a cold start: `Cold start - every variable is at its initial value.`).
8. Check that the header shows **RUNNING** and **IN SYNC**.
9. Save the project (**Ctrl+S**); with a controller connected this also records the controller's values in `data/saved.data.csv`.

20.11 Procedure: online change on a running machine

1. Make and save the change. **Build** it; clear every error and read every warning.
2. Check the header: the right controller, **RUNNING**, and **PROGRAM CHANGED** (the change is not yet on the controller).
3. Review what the change does to data: renamed or retyped variables will be reset, deleted ones dropped.
Rename with **Rename variable...** (F2) rather than by editing, so the old value's name disappears everywhere at once; note that a rename still resets that variable, because matching is by name.
4. Press **F8** and confirm.
5. Read the transfer report. If a variable you needed was reset, write its value back (Chapter 22) or restore it from a data snapshot (Chapter 23).
6. Confirm **IN SYNC**.
7. If RETAIN declarations or the data size changed, allow at least the retain interval before any power interruption (section 20.6.3).

20.12 Messages you may see

Message	Meaning	Action
the program did not compile	The controller rejected the source	Fix the listed errors; nothing changed on the controller
...is outside the process image...	An address does not exist in the configured image	Download hardware (or Download all) with the IO device first
download is disabled (monitor.allow_write=false)	The controller is configured read-only	Enable Allow write / force in Controller Settings and download the hardware
this identity is not permitted to issue that command	Your role is below engineer	Log in as an engineer
...is not called by any task (dialog)	The program is not in the call list	See section 20.5
globals.st could not be saved ... download stopped	The IDE could not save open global edits	Fix the file problem; nothing was sent
The hardware was downloaded, but the PROGRAM was not.	Download all stopped after the hardware step	Download the program again
the controller requires a signed program...	Program signing is enforced	See Chapter 34

21. Run states and faults

This chapter describes the states the controller's scan can be in, the commands that move it between them, what the program and the outputs do in each state, and what makes the controller fault. Read it before you commission a machine: the difference between PAUSED, CONFIG and FAULT decides what your outputs do while you are standing next to them.

21.1 The scan

The runtime executes one cyclic task, MainTask, on a real-time thread (Chapter 18). Every cycle it performs the same steps, in this order:

Step	What happens
1	Wait for the cycle's deadline (absolute time, so the period does not drift)
2	Take any pending command (run, pause, reset fault, config enter/leave)
3	Read all inputs into the %I area of the process image
4	Apply input forces
5	Update the clock bits (Chapter 11)
6	Execute the program(s) of the call list — RUNNING only
7	Apply output forces
8	Write the %Q area to the output devices
9	Publish the snapshot the monitor, OPC UA, EtherNet/IP and Modbus servers read; evaluate alarms; take trace samples
10	Update statistics (cycle count, execution time, jitter, overruns)

Which of these steps run depends on the state.

21.2 The run states

The state is shown in the header of the IDE (the pill next to the target badge), in **Task Configuration** → **System Events**, and on the controller's RUN and IO lamps.

State	Program	Inputs	Outputs	Typical cause
RUNNING	Executes every cycle	Read	Written from the program's %Q	Normal operation; the state after start and after a download
PAUSED	Frozen — not executed	Read	Written, holding their last values	Pause ; leaving CONFIG
CONFIG	Halted	Read	Not written	Config mode (hardware work)
FAULT	Stopped	Read	Written once with each device's fault policy, then not written	A runtime error, the instruction budget, a critical IO device, repeated overruns

State	Program	Inputs	Outputs	Typical cause
STARTING	—	—	—	Transitional, while the runtime starts
STOPPED	—	—	—	The scan engine is not running (runtime stopping or not started)

When the IDE is connected but not logged in to a controller that has accounts, the pill shows **NO SESSION** instead of a state; click it to log in. While the program is halted at a breakpoint the pill shows **STOPPED** followed by the line number (Chapter 24) — the scan state is still **RUNNING**.

21.2.1 RUNNING

The program runs every cycle and drives the outputs. Forces (Chapter 22) are applied on top of what the program computes.

21.2.2 PAUSED

The program is not executed, so every value it computes stays where it was: timer outputs are not updated, counters do not count, SFC charts do not move (but see the note on timers in section 21.3.1). The IO exchange continues:

- **%I** keeps updating — an input can change on the Variables page while nothing else does;
- **%Q** keeps being written with the last values the program produced, so the outputs **hold**. A valve that was open stays open; a running motor keeps running.
- Forces still apply, and writes to **%Q** still reach the outputs.

The Variables pages show a warning while the controller is paused: *The controller is PAUSED. Your logic is not executing...*

WARNING: PAUSED is not a safe state. The outputs stay energised at their last values while no logic — including interlocks written in the program — is being evaluated. Do not use Pause to make a machine safe to approach.

21.2.3 CONFIG

CONFIG is an operator halt for hardware work: wiring, terminating, replacing a card. The program does not run and the scan **stops writing outputs**, while inputs keep being read so that each channel can be watched on the Variables pages as it is wired.

What the outputs physically do in CONFIG is decided by each fieldbus and device, not by the runtime:

Fieldbus	Outputs in CONFIG
EtherCAT, Profinet	The device's own watchdog expires when output data stops arriving; outputs go to the device's configured fail-safe values
Modbus TCP	No further writes are sent; the module keeps the last written values unless its own communication watchdog is configured
EtherNet/IP	Depends on the adapter's connection timeout behaviour

WARNING: Configure the fail-safe or communication watchdog of every remote IO device (in the device, with its vendor's tool) before relying on CONFIG to de-energise outputs. Verify it on the bench.

CONFIG can be entered from any live state, including FAULT: a fault is often the reason for looking at the wiring, and entering CONFIG does not require clearing a fault that has not been diagnosed.

Leaving CONFIG goes to **PAUSED**, never directly to RUNNING. After hardware work the program's output image may describe a machine that no longer exists; check the IO, then press **Run**.

The Variables pages show a warning while the controller is in CONFIG, and the tooltip of the state pill repeats it.

A runtime without a Zeus license that uses real IO enters CONFIG by itself when its 2-hour demo time ends, and then refuses Run, Reset fault and leaving CONFIG until it is restarted or licensed (Chapter 37).

21.2.4 FAULT

The controller goes to FAULT when continuing would mean running a program whose state can no longer be trusted, or running blind (section 21.4). On the transition into FAULT the scan:

1. stops executing the program;
2. drives the outputs **once** according to each IO device's **On fault — outputs** setting (section 21.5): **clear** zeroes that device's %Q, **hold** leaves its last values;
3. from then on keeps reading inputs but **never writes outputs again** from the faulted program.

A FAULT is left by **Reset fault**, by a program download (section 21.3.4), or by **Pause**, which the controller accepts in every state: PAUSED then writes the current %Q image to the outputs again, and **Run** starts the program. Use **Reset fault**, after finding the cause, as the normal way out.

21.2.5 STOPPED

STOPPED means the scan engine is not running at all — the runtime is stopping, restarting or has not started. There is no Stop command in the IDE that leaves the runtime up with the scan stopped; use **Pause** or **Config mode** for that. What the outputs do when the runtime stops is decided by **On stop — outputs** (section 21.5) and by the devices' own watchdogs.

21.3 Commands

The run-state commands are under **Online → Controller** (the group expands in place) and on the action rail. Each asks for confirmation where it changes what the machine does.

Command	Rail	From	To	Minimum role
▶ Run	▶	PAUSED	RUNNING	operator
Pause		any live state	PAUSED	operator
🔧 Config mode	🔧	any live state, including FAULT	CONFIG	engineer
🔧 Leave config mode	🔧	CONFIG	PAUSED	engineer
⚠ Reset fault (Debug menu)	—	FAULT	RUNNING	operator
🔄 Restart runtime...	—	any	STARTING → RUNNING	engineer
Reboot controller (OS)...	—	any	(controller offline)	engineer

Command	Rail	From	To	Minimum role
Shut down controller...	—	any	(controller off)	engineer

The Config mode entry toggles: while the controller is in CONFIG, the menu item reads **Leave config mode** and the rail button is highlighted.

Commands are accepted only when the controller's configuration allows writes (**Controller Settings** → **Allow write / force**, `monitor.allow_write`); otherwise they are refused with `control is disabled` (`monitor.allow_write=false`). Every run-state command is recorded in the controller's audit log with the connection it came from.

Note: A command is taken by the scan at the start of the next cycle. With a 10 ms task, the state changes within 10 ms of the controller receiving the command.

21.3.1 Run and Pause

To pause the machine's logic:

1. Choose **Online** → **Controller** → **II Pause**.
2. Confirm: *IO exchange continues; the program stops executing and outputs hold their last values.*

To resume, choose ► **Run**. The program continues from exactly where its variables were. Standard timers measure real (monotonic) time from the moment they started, so a TON, TOF or TP that was timing when you paused has advanced by the paused time: a 10 s TON paused after 4 s for one minute completes on the first scan after **Run**.

21.3.2 Config mode

To wire or replace IO on a running controller:

1. Bring the machine to a safe condition by its normal means.
2. Choose **Online** → **Controller** →  **Config mode** and read the confirmation. If the controller is RUNNING, it begins **Enter config mode? THE MACHINE IS RUNNING**.
3. Confirm. The pill shows **CONFIG**.
4. Open the Variables page of the program (or **Global Variables**) and watch the %I channels as you wire them. Inputs are live.
5. When finished, choose  **Leave config mode**. The controller goes to **PAUSED**.
6. Check the IO, then choose ► **Run**.

21.3.3 Reset fault

To recover from a FAULT:

1. Find the cause: read the controller log (**Online** → **Controller** → **Controller log...**, Chapter 25) and the IO device pages.
2. Remove the cause (fix the program and download it, restore the IO device, ...).
3. Choose **Debug** →  **Reset fault** and confirm: *The program resumes on the next cycle. Make sure the cause has been addressed.*

The controller goes to RUNNING and resets its consecutive-overrun counter. If the cause is still present — a critical device still missing, an index still out of range — it faults again on the next scan.

21.3.4 Download and the run state

A program download (Chapter 20) pauses the scan for about 50 ms, swaps the program and resumes. **After an accepted download the controller is RUNNING** — also when it was PAUSED, in CONFIG or in FAULT before the download.

WARNING: Do not download a program to a machine in CONFIG or FAULT expecting it to stay halted. It will run the new program immediately.

A **hardware** download restarts the runtime (section 21.3.5).

21.3.5 Restart runtime, reboot and shut down

Command	What happens	Retained data	Program afterwards
Restart runtime...	The runtime service is restarted: the control loop stops, drivers reconnect, the OPC UA address space is rebuilt	Saved on the way down, restored at start	The boot application, in RUNNING
Reboot controller (OS)...	The whole operating system reboots; the controller is unreachable for about 30–60 s	Saved on the way down, restored at start	The boot application, in RUNNING
Shut down controller...	Retained data is saved, then the controller powers off. It does not come back by itself	Saved	— (someone must restore power)

On a real controller these are carried out over SSH and need the engineer role. On the simulator, **Restart runtime** relaunches the simulator (Chapter 26); reboot and shut down are not available.

21.3.6 Warm and cold restart

Zeus uses these terms as follows:

Restart	Non-retained variables	RETAIN variables	How
Warm	Initial values	Restored from the retain file	Restart runtime , reboot, power cycle
Cold	Initial values	Discarded, initial values	Runtime started with <code>--cold</code> on the controller (SSH)
Online change	Carried over by name and type	Carried over	Download program (Chapter 20)

There is no cold-reset or "reset origin" command in the IDE. A download from the IDE is always an online change; the first download to a controller with no program is, naturally, a cold start.

21.4 Faults

The controller distinguishes two classes of program error. The rule is: stop for what makes the program's state untrustworthy; keep running — and say so — for arithmetic that simply has no answer.

21.4.1 Minor faults

A minor fault is an operation that cannot be performed but has a defined result. The scan **continues**.

Operation	Result
Integer division by zero (/)	0
Integer MOD by zero	0
REAL / LREAL division by zero	IEEE 754: +INF, -INF or NaN (NaN for 0.0/0.0)

The controller counts minor faults and keeps the first one of each scan with its line number. The IDE adds a warning to the Messages panel whenever the count rises, for example:

```
line 12: division by zero – 1 more (37 since the program started). The scan kept running ...
```

The count since the program started is part of the controller status. The first occurrence is also written to the controller log (`st-minor-fault`).

Important: INF and NaN propagate: every later calculation using the result is also INF or NaN, and every comparison with NaN is FALSE. Guard divisions whose divisor can be zero, as in the example below. If arithmetic looks wrong while the machine still runs, look for minor-fault warnings first.

21.4.2 Major faults (runtime errors)

A major fault stops the scan and puts the controller in FAULT:

Error	Example message
Array index outside the declared bounds	array index 11 is outside the declared bounds [1..10]
Instruction budget exceeded (runaway loop)	instruction budget of 200000 exceeded - the program has an unbounded loop
Call stack exhausted (recursion deeper than 32)	call stack exhausted
Expression stack overflow	expression stack overflow
SQRT of a negative number	SQRT of a negative value (-4)
LN / LOG of zero or a negative number	LN of a non-positive value (0)
MUX selector outside its inputs	MUX selector 5 is outside 0..3
A % address outside the process image at run time	...is outside the process image...

Addresses in declarations (`AT %IW200`) are checked when the program is bound, at download or at start, so a program with an impossible address is refused before it runs rather than faulting later (Chapter 20).

The controller keeps the number of runtime errors and the text of the last one (`line N: message`) in its status. In this version the IDE shows the FAULT state but not that text; read the controller log (Chapter 25), which records the fault (`task-fault`) with the scan number, and the program statistics when the runtime stops.

21.4.3 The instruction budget

Each scan may execute at most **200 000** VM instructions, across all programs of the call list together. A loop that does not terminate — `WHILE TRUE DO ... END_WHILE`, or a `WHILE` whose exit condition depends on an input that will never change within the scan — is stopped by the budget and faults the controller instead of blocking the real-time core. The budget is not configurable from the IDE. **Task Configuration** → **Stack Usage** lists it with the other per-scan limits.

Write loops with a fixed bound, as in the second example below.

21.4.4 IO faults

Condition	Result
A device marked critical stops exchanging after it has been operational	FAULT (<code>critical-io-lost</code>)
A critical device has not come up within its startup wait (the device's Wait for connection at start , else <code>task.io_startup_grace_ms</code> , 30 s by default)	FAULT (<code>critical-io-never-came-up</code>)
A non-critical device stops exchanging	The device shows FAULT or DEGRADED; the controller keeps running; the device's inputs take its fail-safe pattern (or hold their last values with hold_last_on_fault)
A card missing from a modular rack	Per module policy : ignore, degrade, or fault the device (Chapter 7)
An inhibited device	Never started, never a fault (Chapter 7)

A device is "exchanging" while it is OPERATIONAL or DEGRADED. See Chapter 7 for the critical, inhibit and module-policy settings.

21.4.5 Scan overruns

A cycle that does not finish before the next deadline is an **overrun**. It is counted (**Task Configuration** → **Monitor** → **Overruns**) and logged. What happens next is the task's **Overrun policy** (**Task Configuration** → **Properties**, Chapter 18):

Policy	Behaviour	Use for
skip (default)	Skip the missed cycle(s) and continue on the original time grid	Control loops
catchup	Run the missed cycles back to back until caught up	Counting and logging
fault	After Sensitivity (overruns) consecutive overruns (10 by default), go to FAULT (<code>overrun-fault</code>)	Validated systems where a late cycle is a failure

21.4.6 Watchdog

The **Watchdog** fields in **Task Configuration** → **Properties** (**Enable**, **Time (ms)**) are stored in the project, but the runtime in this version does not act on them. Scan-time supervision is provided by the overrun policy (section 21.4.5) and the instruction budget (section 21.4.3); output supervision is provided by the fieldbus devices' own watchdogs.

21.5 Output behaviour on stop and fault

Each IO device has two settings on its device page, both defaulting to **clear**:

Setting	Values	Applies when
On stop — outputs	clear — safe state (default) / hold — last written values	The runtime stops (restart, hardware download, shutdown)
On fault — outputs	clear — safe state (default) / hold — last written values	The controller enters FAULT

In the configuration file they are `output_on_stop` and `output_on_fault`, with the values `"clear"` or `"hold"` (Appendix H).

- **clear** writes zeros to the device's `%Q` — for a digital output that is OFF, for an analogue output the value 0.
- **hold** leaves the last written values in place.

These settings concern **outputs** (`%Q`). They are distinct from **hold_last_on_fault**, which decides what a device's **inputs** (`%I`) read when the device itself has failed (its fail-safe pattern, or the last good values).

The runtime can only do what the bus lets it do:

- **On fault** is honoured for every fieldbus: the scan performs the clear or hold write itself before it stops writing.
- **On stop** is carried out by the Modbus TCP driver, which writes zeros (or leaves the last values) before it closes the connection. For EtherCAT and Profinet the segment's own fail-safe and watchdog behaviour is authoritative when the master stops.

WARNING: "clear" means zero. If zero is not the safe value of an output (for example a valve that must open on failure, or a 4–20 mA output where 0 mA is an alarm), the safe state must be designed into the field device and the wiring. The emergency stop and guard interlocking are the job of a certified safety system, not of this controller.

21.5.1 Summary: what the outputs do

Situation	Outputs
RUNNING	Driven by the program (plus forces)
PAUSED	Held at the last values the program wrote (plus forces and writes)
Halted at a breakpoint	Held at the last values the program wrote
CONFIG	Not written; each device's watchdog/fail-safe decides
FAULT	Written once per On fault — outputs , then not written
Runtime stopping	Per On stop — outputs (Modbus TCP) or the device's own fail-safe (EtherCAT, Profinet)
Download program	Held for about 50 ms while the program is swapped, then driven by the new program
Controller off / crashed	Each device's watchdog/fail-safe

21.6 Starting up: the boot application

When the runtime starts it loads the configuration, starts the IO drivers, compiles and binds the boot application (the last downloaded program, Chapter 20), restores retained data, and starts the scan directly in **RUNNING**. There is no "start in stop" option.

If the boot application cannot be used — it does not compile, it addresses IO that is not configured, or its signature is not accepted — the runtime starts **without a program**: the scan runs, IO exchanges, inputs update, the monitor answers, and the log says why. The IDE's online views then report *the controller has no program loaded*. Download a corrected program to recover.

Critical IO devices are given their startup wait (per device, else 30 s by default) to come up before their absence faults the controller.

21.7 Panel lamps

The IPC-CM400200 has two panel lamps, RUN and IO:

RUN	IO	Meaning
On	—	RUNNING, nothing forced
Fast blink	—	RUNNING with one or more points forced
Off	—	PAUSED, CONFIG, STOPPED, or the runtime not running
Fast blink	Fast blink, alternating with RUN	FAULT
—	On	Every configured remote IO device is OPERATIONAL
—	Slow blink	At least one configured device is not OPERATIONAL (including DEGRADED)
—	Off	No remote IO device configured

Inhibited devices and simulated devices are not counted for the IO lamp. The fast blink is twice the rate of the slow blink.

21.8 Examples

21.8.1 Guarding against minor and major faults

```

PROGRAM Rates
VAR
    Litres      : REAL;
    Seconds     : REAL;
    Flow_Raw    : REAL;      (* may be INF/NaN when Seconds = 0 *)
    Flow_Safe   : REAL;      (* guarded *)
    Recipe      : ARRAY[1..10] OF INT;
    Step        : INT := 1;
    Setpoint    : INT;
END_VAR
(* Minor fault: REAL division by zero gives INF or NaN, the scan continues *)
Flow_Raw := Litres / Seconds;

(* Guarded: decide what the answer is when there is none *)
IF Seconds > 0.0 THEN
    Flow_Safe := Litres / Seconds;
ELSE
    Flow_Safe := 0.0;
END_IF;

(* A major fault if Step ever Leaves 1..10: the scan stops, the controller
   goes to FAULT. Clamp an index that comes from outside the program. *)
IF Step < 1 THEN
    Step := 1;
ELSIF Step > 10 THEN
    Step := 10;
END_IF;
Setpoint := Recipe[Step];
END_PROGRAM

```

`Step` would typically be written by an operator screen or a SCADA system; clamping it before it is used as an index turns a possible FAULT into a defined behaviour.

21.8.2 A loop that cannot exhaust the budget

```

PROGRAM Search
VAR
    Codes      : ARRAY[0..99] OF DINT;
    Wanted     : DINT;
    Found      : INT := -1;
    i          : INT;
END_VAR
(* Bounded: at most 100 iterations per scan, whatever the data *)
Found := -1;
FOR i := 0 TO 99 DO
    IF Codes[i] = Wanted THEN
        Found := i;
        EXIT;
    END_IF;
END_FOR;
END_PROGRAM

```

21.9 Procedure: recovering from a FAULT

1. Note the time and what the machine was doing.
2. Make the machine safe by its normal means. In FAULT the outputs are at their **On fault** state and are no longer driven.
3. Open **Online** → **Controller** → **Controller log...**, tick **warnings and errors only**, and find the fault event (`task-fault` , `critical-io-lost` , `critical-io-never-came-up` or `overrun-fault`).
4. For an IO fault, open the device page and check its state, reconnect count and **Connected for** time (Chapter 25).
5. For a program fault, find the statement (the log and the program statistics give the line), correct the program, **Build**, and download it. The download leaves the controller RUNNING.
6. Otherwise, when the cause is removed, choose **Debug** →  **Reset fault**.
7. Watch the state pill for a minute. A fault that returns at once has not been fixed.

22. Online monitoring, write and force

This chapter covers watching a running program and changing its values: going online, the online variable tables, prepared values, writing, forcing and releasing, live values in the Structured Text and Ladder editors, and the Force / write table. It also states who may do what, and what is recorded. You need it from the first minute of commissioning.

WARNING: Writing and forcing change a **running machine** immediately. A forced output is driven regardless of the program's logic and interlocks. Before you write or force on real hardware, make sure the result is safe for people and plant, and tell the people at the machine.

22.1 Where live values come from

Every live value in the IDE is read from the controller — never computed by the IDE. The IDE polls the controller about 2.5 times a second (every 400 ms) while **Monitor** is on and while a Variables page is open; one poll carries the whole symbol table with values: every global, every program variable, and every member of every function-block instance and structure.

Values belong to the program the **controller** is running. When the header shows **PROGRAM CHANGED** (Chapter 20), what you see is the old program's data: a variable you have just added shows nothing, and a renamed one shows under its old name.

Reading needs a connection and, on a controller with accounts, a login of any role.

22.2 Going online: Monitor

Online → **Monitor** (or the  button on the action rail) switches online mode on and off. While it is on:

- the **Online** menu title shows **Online** ▾  and the rail button is highlighted;
- open POU editors show live values beside the code (section 22.6);
- ladder POUs animate power flow (section 22.7);
- open Variables pages and the Global Variables table refresh continuously.

When you switch Monitor on, the IDE first compares the project with the controller. If anything differs, the **IDE compared with the controller** dialog opens with the warning that live values come from what is listed under **On the controller**, and a warning is added to the Messages panel. If everything matches, a toast says **Program and hardware match the controller**.

Other messages when going online:

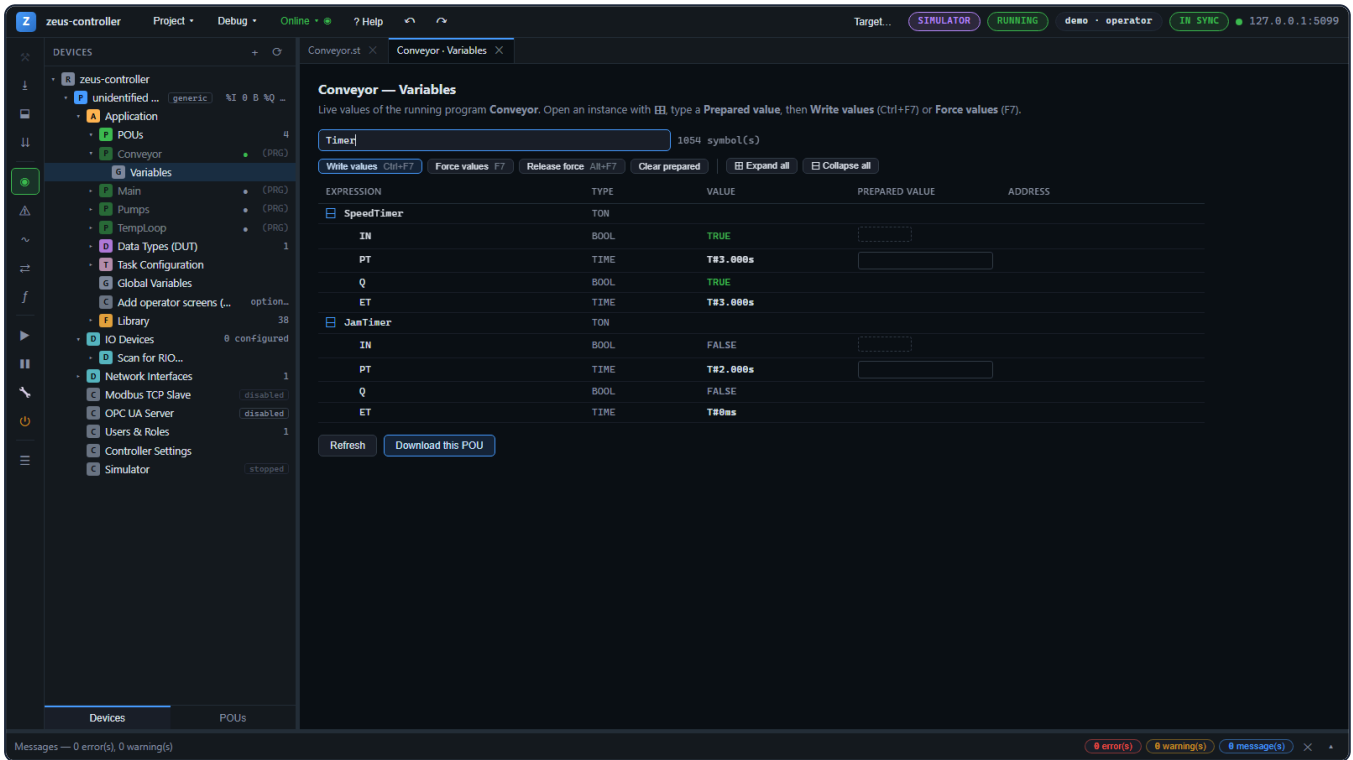
Message	Meaning
Online mode on, but the controller is not connected.	No controller link; check Target... and the connection indicator
Online mode on — open a POU to see values beside its declarations	Nothing to show values in yet; open a POU or a Variables page
<i>Online, but the controller has no program loaded...</i>	Download the program (Chapter 20)

Message	Meaning
NO SESSION in the state pill	The connection is not logged in; online mode is ended. Click the pill to log in

22.3 The online variable tables

Two views hold the same online variable table:

View	Opened from	Shows
<POU> — Variables	the Variables node under a POU in the tree, or Open Variables in the POU's context menu	Every symbol of the running program — all programs of the call list, all globals, all instances — with a filter
Global Variables → Live values on the controller	the Global Variables node	Only the globals declared in <code>globals.st</code> , and their members



Online variable table: the Conveyor Variables page filtered on "Timer", with the SpeedTimer and JamTimer instances expanded

A Variables page reads **Live values of the running program <name>** above the table and the number of symbols beside the filter box (for example `1054 symbol(s)`). If no program is running it says so and offers nothing to write. Both views update continuously while they are visible, whether or not Monitor is on; **Refresh** re-reads at once.

Note: Global variable lists (`gv1/...`) and generated IO aliases are not listed in the **Global Variables** live table, which shows only what `globals.st` declares. They are in every POU's Variables page; type the name in the filter.

22.3.1 Columns

Column	Content
Expression	The name. Instance members are indented under their instance; array elements appear as <code>[i]</code>
Type	The declared type; <code>instance</code> for an instance or structure row
Value	The live value. TRUE is shown green
Prepared value	The value you intend to send (section 22.4). Empty for rows that cannot be written
Address	The <code>%I</code> , <code>%Q</code> or <code>%M</code> address for located variables and IO aliases
<i>(last)</i>	RETAIN and CONST badges

A row whose address is forced has a **FORCED** badge next to the name — **FORCED TRUE** or **FORCED FALSE** for a forced bit, just **FORCED** for a word or byte that contains a forced bit.

22.3.2 Expanding instances, structures and arrays

A row with members has  in front of it. Click  to open it and  to close it; nested instances open level by level.  **Expand all** and  **Collapse all** in the toolbar act on every row currently listed. Which rows are open is remembered while the IDE is open, also across views.

An array is one more level: `Recipe` opens to `[1]` , `[2]` , ..., and an array of structures opens each element to its members, as in `gPlant.doors[2].Name` .

22.3.3 Sorting

Click a column header — **Expression**, **Type**, **Value** or **Address** — to sort by it; click again to reverse; click a third time to return to declaration order (the order of the source). The active column shows ▲ or ▼. Numbers sort as numbers (9 before 10), names in natural order, and empty cells go last. Sorting applies at every level of the tree.

22.3.4 Filtering and search

Type in the filter box — **Filter — name, type or address** on a Variables page, **Search — name, member (FM_01_01.Ai), type or address** on Global Variables. Matching is case-insensitive, on any part of the full dotted name, the type or the address.

- A match inside an instance opens that instance, so `SpeedTimer.ET` or just `.ET` finds the member and shows it in place.
- **Ctrl+F** on a Variables page puts the cursor in the filter box.
- **Esc** in the filter box clears it.
- Filtering uses the last values read and does not load the controller.

22.3.5 Large programs

The controller sends the whole table in one reply of up to 12 MB. A project so large that its table does not fit is sent in part, and the IDE adds a warning once per program: *The controller's variable list is too large to send whole: showing N of M top-level variables. The rest can still be read and written by name.* The variables that were not sent are still in the program; they can be traced, written with the Force / write table (by address), and read by OPC UA or EtherNet/IP.

22.4 Prepared values

The **Prepared value** column is a list of values you intend to send. Nothing is sent until you give a write or force command, so you can prepare several values and send them together.

Row type	How to prepare
BOOL	Click the box: it cycles TRUE → FALSE → (empty)
Number, TIME, STRING	Type the value: TRUE / FALSE , 1500 , -3.25 , 16#FF , T#500ms , 'text'
Any	Double-click the Value cell to copy the current value into the prepared box
Any	Right-click → Prepare value

- **Enter** moves to the next prepared box; it does **not** write. Sending values to a running machine is always an explicit command.
- **Esc** empties the box.
- **Clear prepared** empties the prepared column of the rows shown.
- The count of prepared values in the table is shown at the right of the toolbar (**3 prepared**).
- Prepared values are one list per IDE session, shared between the views, and survive the table being redrawn.

Rows that cannot be written have no prepared box: instance rows, **CONSTANT** variables, and the outputs of standard function blocks. The controller decides what is writable; the IDE only offers what it will accept.

22.5 Write, force and release

22.5.1 The difference

	Write	Force
Effect	Sets the value once	Holds the value every scan until released
Can the program change it again?	Yes, on its next assignment	No
Applies to	Program variables, globals, FB inputs and locals; %Q and %M points	Single bits (BOOL) at a %I , %Q or %M address
Inputs %I	Refused — the next input read would overwrite it	Yes: the forced value replaces what the device delivers
Survives a download?	Only as the program's data (online change)	Yes — forces are held in the process image
Shown by	Nothing after the write	FORCED badges, the FORCE header badge, the RUN lamp blinking fast

A write is right for a setpoint, a mode selector or a recipe parameter — a variable the program reads but does not assign every scan. Writing a variable the program assigns every scan appears to "not work": the value is replaced on the next cycle, typically within milliseconds.

A force is applied twice per scan — after the inputs are read and again after the program has run — so neither the fieldbus nor the program can win the value back. Only single bits can be forced; the force table in the controller is

a bit mask. To hold a word, write it (for an output channel the program does not drive, that holds until you change it), or force its individual bits.

22.5.2 Commands in the variable table

Command	Key	Acts on
Write values	Ctrl+F7	Every prepared value in the rows shown, once
Force values	F7	Every prepared value in the rows shown that is a BOOL at a % address
Release force	Alt+F7	The forces on the rows shown
Clear prepared	—	Empties the prepared column of the rows shown

The keys work while the table is the visible view.

Important: Elsewhere in the IDE, **F7** and **Ctrl+F7** are **Build** and **Build All**. While a variable table is on screen, only the table's meaning applies (Force, Write): no build is started. Build from a POU editor tab or from the **Debug** menu.

What happens on **Write values**:

1. If the controller has accounts and you are not logged in, the login dialog opens.
2. Program variables are sent in one batch; %-addressed points are written to the process image one by one.
3. Each value that was accepted leaves the Prepared column. A value that was refused **stays** in the column so that you can correct it, and the reason is added to the Messages panel, for example `Speed_Sp: 'fast' is not a number` or `Alarm_In: an input can only be forced (F7), not written`.
4. A toast and a Messages note summarise: `Wrote 3 value(s) on the running controller.` or `Wrote 2, 1 refused - see Messages`.

Force values works the same way, with the result logged as a warning (`Forced 1 value(s) on the running controller.`). A prepared value on a row that cannot be forced is refused with `only a BOOL at a %-address can be forced - use Write (Ctrl+F7)`.

Release force releases every forced bit among the rows shown; each point returns to its real value on the next scan. `Nothing shown here is forced.` means what it says — use the filter to show the forced row, or release all forces (section 22.5.4).

22.5.3 The right-click menu

Right-click a row for a shorter way to one item. The menu offers only what the row can do, in this order:

Entry	Shown when
Write all prepared values (N) — Ctrl+F7	Any prepared value exists in the table; N is how many
Write only this one (value)	This row has a prepared value
Force all prepared values — F7	A prepared value exists on a forceable row
Expand / Collapse	Instance, structure or array rows
Read-only - cannot be written (disabled)	A row the controller will not write
Write TRUE, Write FALSE, Toggle (now ...)	A writable BOOL

Entry	Shown when
Write value... (type it in Prepared value)	A writable non-BOOL with nothing prepared: puts the cursor in its prepared box
Force only this one (value), Force TRUE, Force FALSE, Release force — Alt+F7	A BOOL at a % address. Release force is enabled only when it is forced
Prepare value	A writable row: copies the current value into the prepared box
Copy name, Copy value	Always (value only when the row has one)

Write only this one sends the value already in the row's prepared box; it does not ask again. Every entry goes through the same checks, and the same audit record, as the toolbar commands.

22.5.4 The FORCE badge and releasing all forces

While any point on the controller is forced, the header shows **FORCE 1 point** or **FORCE N points**, and the controller's RUN lamp blinks fast (Chapter 21). The count comes from the controller, not from this IDE, so forces set from another IDE or left over from an earlier session are shown too.

To release every force on the controller, do one of:

- click the **FORCE** badge;
- choose **Debug** →  **Clear forces**;
- click  on the action rail;
- press **Clear all forces** in the Force / write table.

Each asks for confirmation — *Each forced point returns to whatever the program writes, which may move an output immediately.*

WARNING: Forces are held by the controller, not by the IDE. They survive closing the IDE, logging out, a program download and a network interruption. Logging out does **not** release them. Release every force before you leave a machine, and check that the **FORCE** badge has gone. (A runtime restart — including a hardware download — starts with no forces.)

A forced input is not visible from the plant: the device, the terminal and the input LED show the real signal, while the program sees the forced value. Record every force you set in the commissioning log.

22.6 Live values in the Structured Text editor

With Monitor on, each open ST POU shows a live-value strip at the right of both panes — declarations and body. A value appears on each line that starts with a variable name: a declaration, an assignment, or an instance call such as `SpeedTimer(...)`.

Inline values. With **Inline values** on (the default; the button in the editor's action bar), every variable in the code also shows its value right beside it — `level 2479`, `running TRUE`, `charge.state 12` — TRUE in green, FALSE in grey, numbers in blue, the same way the ladder's ST view shows them. Pin names in a call (`IN :=`, `PT :=`) carry no badge. Function block and METHOD tabs show inline values too, for the instance chosen in the **Instance** list.

Timers and counters. A block instance shows what it is doing, both where it is declared (`SpeedTimer : TON;`) and where it is called (`SpeedTimer(IN := Motor, PT := T#3s);`); `IN` and `Q` as lamps (green = TRUE) and the

elapsed time against the preset, for example `IN Q 1.25s / 3s`. While a timer counts - IN on, Q not yet - the time is amber. A counter shows `CU / CD`, `Q` and `CV / PV`. Double-click the badge for all the instance's members.

While inline values are on, the value column at the right of the editor is hidden - it would only repeat them. It returns when you edit, and stays at the right edge of the pane when a long line is scrolled sideways.

The values stay while you read and click:

You	The values
click anywhere - code, a value, empty space	stay; the click places the caret where it landed
double-click a value	stay; the write dialog opens for that variable
start typing	step aside, and the text goes in where you clicked
drag over code, or double-click a word of code	step aside, with that text selected
press Esc , or click outside the editor	come back (if you changed the code, after you save it)
leave the code alone for 10 seconds, without having changed it	come back
jump to a line from a message or a find result	step aside, so the line can be edited

Turned off with the **Inline values** button, they stay off until you press it again.

- A value that changed since the previous poll is highlighted.
- An instance shows **+ <Type>**; click it to open a pop-up with the instance's members and their live values.
- Double-click a value to write that variable. The prompt shows the current value and accepts `TRUE / FALSE`, a number, `16#FF`, `T#500ms`, or `'text'` for a STRING. A single click never writes.
- The editor strip writes program variables only. A variable located at a `%` address is refused by the controller from here (... use the process-image write or force commands for a `%`-address); write or force it from its Variables page or the Force / write table.

22.6.1 Hover information

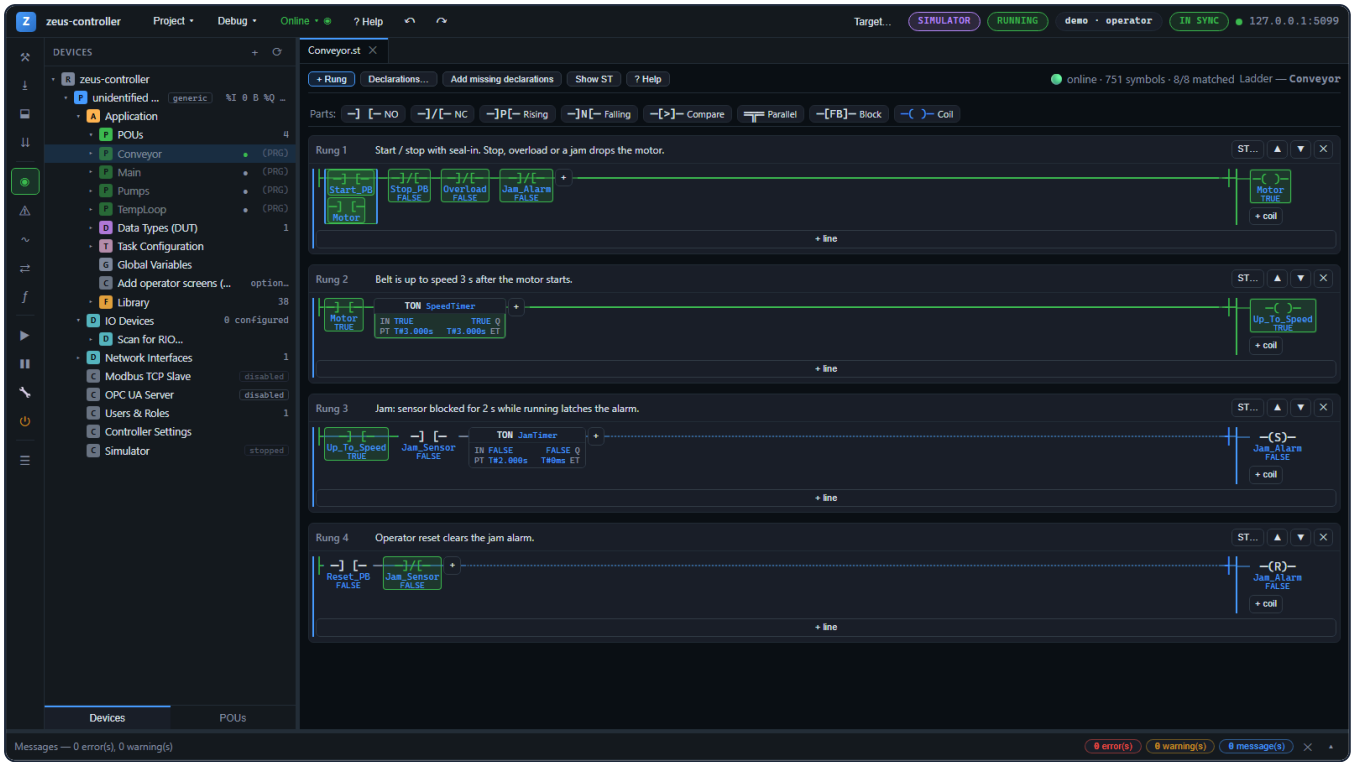
Hover the mouse over any variable name in the editor to see its declaration: the kind (Input, Output, In/Out (by reference), Local, Temporary, Static, Global, External, IO alias...), the type, the initial value, the comment written beside the declaration, and where it is declared. When online, the tooltip also shows the live value.

22.6.2 Function blocks: choosing an instance

Inside a FUNCTION_BLOCK the variables exist once per instance, so a bare name has no single value. A function-block tab therefore has an **Instance** selector in its toolbar. Choose the instance to watch; values and writes then refer to that instance. Until one is chosen, the hover tooltip says **choose an INSTANCE to see its value** and no values are shown. The selector lists the instances declared in the project, so it can be set before going online.

22.7 Live power flow in Ladder

A ladder POU animates from the running controller when Monitor is on.



Online ladder: the Conveyor ladder with live power flow — energised paths green, each contact and coil with its live value, timers with IN/PT/Q/ET

- The energised path is drawn green: from the left rail along each wire up to the first open contact, and through to the coil when the rung has power.
- Every contact and coil shows its variable's live value; a conducting element is lit.
- A function block box (TON, CTU, ...) shows its inputs and outputs live, for example `IN TRUE` , `PT T#3.000s` , `Q TRUE` , `ET T#3.000s` .
- The status at the top right reads, for example, **online · 751 symbols · 8/8 matched**: data is arriving, and 8 of the diagram's 8 variables were found on the controller. If it says **0/... matched**, download this program.
- Right-click an element → **Set value...** to write or force its variable: a `%`-address is forced, a plain variable is written.

See Chapter 16 for the Ladder editor itself.

22.8 The Force / write table

Debug → **Force / write table...** is a scratch table of process-image points to override during commissioning. It works by address, so it reaches any `%I` , `%Q` or `%M` point whether or not the program declares a variable for it.

Column / button	Meaning
Address	<code>%QX0.0</code> , <code>%IX2.3</code> , <code>%QW4</code> , <code>%MD8</code> , ...
Kind	force (bit) for an <code>X</code> address; write (INT) , write (DINT) ... for a word or byte
Forced now	The value the controller is forcing at that address, or —
Set value	The value to apply: <code>1 / 0</code> , <code>true / false</code> , <code>on / off</code> for a bit; a number for a word
Apply	Force the bit, or write the word

Column / button	Meaning
Clear	Release the force on that bit (bits only)
×	Remove the row from the table (does not release anything)
Add row	Add the address and value typed below the table
Clear all forces	Release every force on the controller

The rows are kept in the browser, so the table survives a reload. `%I` words cannot be written — force their bits instead.

22.9 Who may write and force

The controller checks every request against the role of the logged-in account (Chapter 34):

Action	viewer	operator	engineer
Read values, see forces, go online	yes	yes	yes
Write variables and <code>%Q</code> / <code>%M</code> points	—	yes	yes
Force and release	—	yes	yes
Release all forces	—	yes	yes
Run / Pause, Reset fault	—	yes	yes
Arm a trace, set breakpoints, download	—	—	yes

Knowing what is forced is deliberately a viewer right: the person who most needs to see a force is often the one with the least authority to change it.

If the controller's configuration has **Allow write / force** switched off (`monitor.allow_write: false`), every write and force is refused for every role, with `writes are disabled (monitor.allow_write=false)` or `forcing is disabled (monitor.allow_write=false)`.

A controller with no accounts and no token grants every connection engineer rights. That is acceptable only on an isolated bench; see Chapter 34.

22.10 What is recorded

The controller writes an audit record for every change made from the IDE:

Action	Audit record (controller log, level AUDIT)
Log in	<code>monitor: <address> logged in as "<user>" (<role>)</code>
Write a variable	<code>monitor write from <address>: <name> := <value></code>
Write a <code>%</code> point	<code>monitor write from <address>: %Q4 = 1500 (INT) (area and byte offset)</code>
Force / release a bit	<code>monitor force from <address>: %Q0.0 / monitor unforce from ... (area, byte.bit)</code>

Action	Audit record (controller log, level AUDIT)
Release all forces, run, pause, config, reset fault	monitor control from <address>: <command> -> <result>
Refused request	monitor: DENIED "<command>" to "<user>" (<role>) ...

Writes and forces carry the network address and port of the connection; the login record of that connection names the user. Read them in **Online** → **Controller** → **Controller log...** or the **Events** page (Chapter 25).

The IDE also adds a line to its Messages panel for every write, force and release, with the count of forces active on the controller.

22.11 Worked procedure: commissioning a conveyor

This procedure uses the conveyor program below — the ST equivalent of the ladder in section 22.7 — on the simulator, then on a controller. The buttons in the example are program variables, so they can be written; on a real machine they would be IO aliases at %I addresses and would be forced instead.

```

PROGRAM Conveyor
VAR
    Start_PB   : BOOL;           (* operator start, TRUE = pressed *)
    Stop_PB    : BOOL;           (* operator stop, TRUE = pressed *)
    Overload   : BOOL;           (* motor overLoad trip *)
    Jam_Sensor : BOOL;           (* photo-eye blocked *)
    Reset_PB   : BOOL;
    Motor      : BOOL;
    Up_To_Speed : BOOL;
    Jam_Alarm   : BOOL;
    SpeedTimer  : TON;
    JamTimer    : TON;
END_VAR

(* Rung 1: start / stop with seal-in *)
Motor := (Start_PB OR Motor) AND NOT Stop_PB AND NOT Overload AND NOT Jam_Alarm;

(* Rung 2: belt is up to speed 3 s after the motor starts *)
SpeedTimer(IN := Motor, PT := T#3s);
Up_To_Speed := SpeedTimer.Q;

(* Rung 3: sensor blocked for 2 s while running latches the jam alarm *)
JamTimer(IN := Up_To_Speed AND Jam_Sensor, PT := T#2s);
IF JamTimer.Q THEN
    Jam_Alarm := TRUE;
END_IF;

(* Rung 4: operator reset clears the alarm once the sensor is clear *)
IF Reset_PB AND NOT Jam_Sensor THEN
    Jam_Alarm := FALSE;
END_IF;
END_PROGRAM

```

To test the start/stop and the jam alarm:

1. Download the program (**F8**) and check that the header shows **RUNNING** and **IN SYNC**.

2. In the tree, open **Conveyor** → **Variables**. Type `Timer` in the filter and click  in front of **SpeedTimer** and **JamTimer** (as in the screenshot in section 22.3).
3. Clear the filter. In the **Prepared value** column click the box of **Start_PB** until it shows **TRUE**.
4. Press **Ctrl+F7 (Write values)**. **Motor** turns TRUE and stays TRUE through the seal-in.
5. Prepare **Start_PB = FALSE** and press **Ctrl+F7** — a push button must be released again; the program does not do it for you.
6. Watch **SpeedTimer.ET** count to `T#3.000s`; **SpeedTimer.Q** and **Up_To_Speed** turn TRUE.
7. Right-click **Jam_Sensor** → **Write TRUE**. After 2 s **JamTimer.Q** and **Jam_Alarm** turn TRUE and **Motor** drops.
8. Right-click **Jam_Sensor** → **Write FALSE**. Prepare **Reset_PB = TRUE**, press **Ctrl+F7**, then prepare it **FALSE** and write again. **Jam_Alarm** clears.
9. Switch **Online** →  **Monitor** on and open the POU tab: the same values appear beside the code; in the ladder version the rungs animate.
10. Save a data snapshot of the result for the commissioning record (Chapter 23).

On the real machine the inputs come from the field. To simulate an input there — for example to test the jam alarm without blocking the photo-eye — force the alias instead: prepare **TRUE** in its row and press **F7**, test, then press **Alt+F7** to release it and confirm that the **FORCE** badge has gone.

22.12 Troubleshooting

Symptom	Cause	Action
A written value returns to its old value at once	The program assigns the variable every scan	Write the variable the program reads (the setpoint), not the result; or force the <code>%</code> bit
an input can only be forced (F7), not written	The row is at a <code>%I</code> address	Force it (BOOL), or change the source of the signal
only a BOOL at a %-address can be forced	Forcing a program variable, or a word	Write it instead, or force the individual bits
'X' is CONSTANT ...	CONSTANT values are compiled into the code	Change the declaration and download
'X' is <type> - write its members individually	The row is a structure or instance	Open it with  and write the members
No values beside a new variable	PROGRAM CHANGED: the controller runs the old program	Download
No values in a function-block tab	No instance selected	Choose one in Instance
Values freeze	Controller PAUSED or in CONFIG (a note says so), or halted at a breakpoint	See Chapters 21 and 24
this identity is not permitted to issue that command	Viewer role	Log in as operator or engineer

23. Trace and data snapshots

This chapter covers the two ways of capturing data from a running controller. The **trace** (scope) records up to eight variables on every scan and plots what they did — the overshoot of a loop, a one-cycle glitch, the order in which two signals changed. A **data snapshot** records the value of every variable at one moment, for a commissioning record, a fault investigation, or to put values back later.

23.1 Trace or monitor?

Question	Tool
What is this value now?	Monitor and the Variables pages (Chapter 22) — refreshed about 2.5 times a second
What did this value do between two scans, during a step, while the valve moved?	Trace — one sample per scan, taken on the controller
What were all the values at 14:05, and can I put them back?	Data snapshot

The monitor polls; it cannot see a signal that is TRUE for one 10 ms scan between two polls. The trace samples on the scan thread itself, at the end of every cycle, into a buffer on the controller, so nothing that happens at scan rate is missed.

23.2 How the trace works

- The samples are taken **on the controller**, at the scan boundary, after the program and the output write — the values the outputs were given in that cycle.
- The buffer is a ring in the controller's memory, allocated when the trace is armed. Nothing is allocated, formatted or locked on the scan thread while recording.
- The IDE reads the ring about three times a second while the **Trace / scope** tab is open and appends what it receives to the plot.
- The horizontal axis is the controller's **scan number**, converted to time with the task period. It is machine time, not the time the browser received the data, so two traces of the same machine line up by it.
- There is one trace buffer per controller. Arming a new trace replaces the one that was there, whoever armed it.
- The ring lives in RAM. It is lost when the runtime stops, and a program download stops the recording (the recorded variables belong to the old program).

23.2.1 Limits

Limit	Value
Channels	8
Depth (samples in the ring)	500, 2 000, 5 000 or 20 000
Sample every	every scan, or every 2, 5, 10, 20 or 50 scans

Limit	Value
Traceable types	BOOL, SINT, USINT, BYTE, INT, UINT, WORD, DINT, UDINT, DWORD, LINT, ULINT, LWORD, REAL, LREAL, TIME
Not traceable	STRING, structures, arrays as a whole, function-block instances (their members are traceable)
Controller memory	at most 8 channels × 20 000 samples × 8 bytes ≈ 1.2 MB

The time a trace covers is $depth \times sample\ every \times task\ period$. With a 10 ms task:

Depth	Every scan	Every 10 scans	Every 50 scans
500	5 s	50 s	4 min 10 s
2 000	20 s	3 min 20 s	16 min 40 s
5 000	50 s	8 min 20 s	41 min 40 s
20 000	3 min 20 s	33 min 20 s	2 h 46 min

Recording one scan in N lowers the resolution: a pulse shorter than N scans may not be recorded. Choose the depth to cover the event, and "every scan" whenever the detail matters.

Note: The trace has no trigger condition in this version. It records from the moment it is armed and keeps the most recent *depth* samples in the ring. To capture an event, arm the trace, make the event happen, and press **Stop** soon enough that the event is still in the ring.

23.3 The Trace / scope page

Open it with **Online** → ~ **Trace / scope...** or the ~ rail button. It opens as a tab; there is one, because there is one buffer on the controller.

Element	Meaning
Depth (samples)	Size of the ring on the controller
Sample every	every scan , or one scan in 2, 5, 10, 20, 50
● Arm & record	Resolves the channels on the controller, allocates the ring and starts recording. While recording it reads 🔄 Re-arm & record
■ Stop	Stops recording; the samples stay on screen and on the controller
Clear	Stops and releases the buffer on the controller; clears the plot
↓ Export CSV	Saves the samples shown to a CSV file (section 23.5)
State pill	RECORDING , armed , stopped or not armed
Status line	samples taken, ring size and scan period, e.g. 4210 taken • ring 5000 • 10.00 ms scan
N sample(s) lost	Samples the IDE did not receive in time (section 23.4.2)
Plot	One line per channel; the time span is written at the bottom right

Element	Meaning
Legend	Per channel: colour, name (with (ms) for TIME), value at the cursor, minimum ... maximum
Channels	The list of traceable variables of the running program, with a filter; tick up to 8

23.3.1 Reading the plot

- Each channel is scaled to its **own** range, like a multi-pen recorder: a BOOL and a 0–27 648 analogue value are both readable on one plot. The legend gives each channel's minimum and maximum, so the shape of a line is never mistaken for its magnitude.
- REAL and LREAL channels are drawn as straight lines between samples. Every other type — BOOL, integers, TIME — is drawn as **steps**, because the value held until a scan changed it; a diagonal would show a ramp the controller never had.
- TIME values are shown in milliseconds.
- Move the pointer over the plot: a dashed cursor marks the sample under it, and the legend shows every channel's value at that sample. Without the pointer the legend shows the latest values.
- The page keeps at most twice the ring's depth; older samples scroll off.

23.4 Recording a trace

To record a trace:

1. Download the program you want to trace and check **IN SYNC**. The channel list is read from the program the controller is running.
2. Log in as an **engineer**. Arming, stopping and clearing a trace take scan time and replace any other user's trace, so they are engineering actions. Reading a trace someone else armed needs only viewer rights.
3. Open **Online** → ~ **Trace / scope...**
4. Under **Channels**, type part of a name in **Filter variables** and tick up to eight variables. Function-block members are listed with their instance, for example `Settle.ET`, `Settle.Q`; the instance itself is not listed.
5. Choose **Depth (samples)** and **Sample every**.
6. Press **• Arm & record**. The pill shows **RECORDING** and the lines start to grow.
7. Make the event happen.
8. Press **■ Stop**. The IDE reads the last samples and draws the complete tail.
9. Move the pointer across the plot to read values; export the CSV if you want to keep it.
10. Press **Clear** when finished, to release the controller's memory.

If a channel cannot be resolved, **Arm & record** fails and the page repeats the controller's reason, naming the variable. A trace never silently drops a channel.

Opening the Trace / scope tab while a trace is already armed on the controller (from another IDE, or before this tab was closed) picks it up: the channels, depth and state are read back and the plot continues.

23.4.1 Changing channels

Changing the ticked channels does not change a running recording. Press **↻ Re-arm & record** to start a new recording with the new channels; the previous samples are discarded.

23.4.2 Lost samples

If the IDE falls behind — a slow network, a busy PC, a deep ring filled at every scan of a fast task — the controller's ring can lap the IDE's read position. The samples in between are lost to the IDE (they were recorded, but overwritten before they were read). The page then:

- shows **N sample(s) lost** in the status line;
- **breaks the line** at the gap instead of joining across it;
- writes `# samples lost here - the ring was lapped` at that point in the CSV.

A trace drawn smoothly across a gap would show a machine nobody observed. If you see lost samples, sample every N scans, or stop and read sooner.

23.5 Exporting a trace

↓ **Export CSV** saves `trace-<date>T<time>.csv` through the browser. The format:

```
scan,time_ms,PV,Output,Settle.ET (ms),AtSp
812340,0.000,0,0,0,0
812341,10.000,0.24,12,0,0
812342,20.000,0.52,11.9,0,0
# samples lost here - the ring was lapped
812790,4500.000,49.8,50.1,0,0
```

Column	Content
scan	The controller's scan number — its own clock
time_ms	Milliseconds since the first sample, from the scan number and the period
one column per channel	The value; TIME channels in milliseconds, with (ms) in the header

Keep the scan column when you process the file: it tells a gap in the machine from a gap in the recording.

23.6 Example: tuning a control loop

The program below closes a PI loop around a simulated first-order process, so it can be traced on the simulator. Trace `Setpoint`, `PV`, `Output`, `Integral` and `AtSp`, change `Setpoint` from 50 to 80 with a write (Chapter 22), and watch the response.

```

PROGRAM TraceDemo
VAR
    Setpoint : REAL := 50.0;
    PV       : REAL;      (* simulated process value *)
    Error    : REAL;
    Integral : REAL;
    Output   : REAL;      (* 0..100 % *)
    Kp       : REAL := 2.0;
    Ki       : REAL := 0.05; (* per scan *)
    Lag      : REAL := 0.02; (* first-order plant, per scan *)
    AtSp     : BOOL;
    Settle   : TON;
END_VAR
(* PI controller, one step per scan *)
Error := Setpoint - PV;
Integral := Integral + Ki * Error;
IF Integral > 100.0 THEN Integral := 100.0; END_IF;
IF Integral < 0.0 THEN Integral := 0.0; END_IF;
Output := Kp * Error + Integral;
IF Output > 100.0 THEN Output := 100.0; END_IF;
IF Output < 0.0 THEN Output := 0.0; END_IF;

(* first-order plant: PV moves towards Output *)
PV := PV + Lag * (Output - PV);

(* in band for 2 s *)
Settle(IN := ABS(Error) < 0.5, PT := T#2s);
AtSp := Settle.Q;
END_PROGRAM

```

To use it:

1. Download `TraceDemo` to the simulator (Chapter 26).
2. Open **Trace / scope**, tick `Setpoint`, `PV`, `Output`, `Integral` and `AtSp`; set **Depth** 2000 and **Sample every** *every scan*; press **• Arm & record**.
3. On `TraceDemo` 's Variables page, prepare `Setpoint = 80` and press **Ctrl+F7**.
4. After about ten seconds press **■ Stop**.
5. Read the overshoot of `PV` against `Setpoint`, the time `Output` spends at 100 % (saturation), and when `AtSp` turns TRUE.
6. Write `Kp = 1.0` and repeat from step 2 to compare. Export each trace as CSV for the tuning record.

For real loops use the PID blocks of the library and their autotuner (Appendix C); the tracing procedure is the same.

23.7 Data snapshots

A data snapshot is a CSV file with the value of every variable the controller reports, taken at one moment:

```

name,type,value
Motor,BOOL,TRUE
g_FillCount,DINT,1482
SpeedTimer.PT,,T#3.000s
SpeedTimer.Q,,TRUE

```

The first line is always `name,type,value`. Values containing a comma, a quote or a line break are quoted. The names are those of the online variable tables, including instance members (for which the controller may leave the type column empty). A restore needs only the `name` and `value` columns, so a CSV edited or produced elsewhere can be restored as long as it has that header.

23.7.1 Where snapshots come from

Action	File written	Where
Online → ↓ Save data snapshot...	<code><controller name>- <date>T<time>.snapshot.csv</code>	Downloaded through the browser
Project → Save project (Ctrl+S) with a controller connected	<code>data/saved.data.csv</code> — replaced on every save	In the project
Upload (Chapter 20)	<code>data/<date>T<time>- upload_<program>.data.csv</code>	In the project

Saving the project records the data because the values on a running machine change while its source does not: a project saved without them would not describe the machine. If the values cannot be read, the save of the program still succeeds and a warning says that the controller's data was **not** saved.

`data/` travels with the project in **Save project as...** and in archives (Chapter 5).

23.7.2 Restoring values

Online → ↑ **Restore data snapshot...** writes the values of a snapshot back onto the running controller.

To restore a snapshot:

1. Log in as an **operator** or **engineer**.
2. Choose **Online** → ↑ **Restore data snapshot....**
3. Choose the source: **Saved in project** lists the files in `data/`, newest first (the newest is preselected); **...or a file** takes any snapshot CSV from disk. A file chosen from disk wins.
4. Press **Restore values**. The IDE reads what is running now and sorts the file's rows **before sending anything**:

Group	Not sent because
already match	the running value is already the one in the file
cannot be written	an FB output, a % address, a CONSTANT, a whole structure
not in this program	the variable no longer exists

5. Read the confirmation, which gives the number of values to write and the number skipped in each group, and confirm.
6. The values are written in one request. The dialog and the Messages panel report `Restored N/M value(s) ...` with the skipped and refused counts; the first 25 refusals are listed with their reasons.

Important: Only variables the program reads but does not assign every scan will hold — setpoints, mode selectors, recipe parameters, counters it only increments. Anything the program computes each scan reverts on the next scan. Hardware-mapped inputs cannot be written at all.

WARNING: A restore writes to the running machine. Restoring a mode selector or a sequence step changes what the machine does at once. Review the file before you restore it to a machine in operation.

23.7.3 Typical uses

Use	How
Commissioning record	Save data snapshot... after each test; keep the CSV with the test sheet
Before a risky change	Snapshot, change, and restore the setpoints if the change must be undone
Moving tuned values to a second machine	Snapshot on machine A; restore on machine B running the same program
After a download that reset values	Restore from <code>data/saved.data.csv</code> (see the transfer report, Chapter 20)
Fault investigation	Snapshot immediately after the fault, before anyone resets it

A snapshot is not a historian: it holds one moment. For continuous recording use a trace (short, at scan rate) or the SCADA historian (long, at a lower rate).

24. Breakpoints and debugging

This chapter describes source-level debugging of Structured Text: breakpoints, stepping, the call stack, and — first — what a stopped program means for a machine. Use it on the simulator or on a machine that is safe to freeze. For watching a program without stopping it, use online monitoring (Chapter 22) and the trace (Chapter 23), which do not interfere with the scan.

24.1 What a breakpoint does to the machine

WARNING: A breakpoint stops the program **part-way through a scan**. The IO scan keeps running: inputs stay live and outputs keep being written — from the values the program had computed when it stopped, which it is no longer updating. A machine in motion stays in motion, and no interlock written in the program is evaluated while it is stopped. Debug with breakpoints on the simulator or on a bench. On a real machine, use them only when freezing the logic is safe.

In detail, while the program is halted:

Item	Behaviour
The program	Not executed. All programs of the call list stop, not only the one with the breakpoint
Statements before the breakpoint in the halted scan	Already executed. Their assignments to %Q are written to the outputs at the end of that scan
Statements after the breakpoint	Not executed until you continue or step
Inputs %I	Read every cycle, as usual
Outputs %Q	Written every cycle with their current values (hold)
Forces	Applied as usual
Controller state	Remains RUNNING ; the IDE's state pill shows STOPPED <line> instead
RUN lamp	On (or fast blink if something is forced) — the panel does not show a halt
Alarms, trace, OPC UA, EtherNet/IP, Modbus slave	Keep running on the frozen values
Standard timers (TON, TOF, TP)	Measure real time. A timer that was timing when the program stopped has advanced by the halt time when it next runs, and may complete at once
Scan statistics	Keep counting; a halted scan is short, so it causes no overruns

This is the same exposure as **Pause** (Chapter 21), entered at a statement instead of at the end of a cycle, with one difference that matters: a scan halted in the middle may have written some of its outputs and not others. For example, a program that switches one valve off early in the scan and another on late in the scan, halted in between, leaves both off.

Because of this, setting a breakpoint and stepping are **engineer-only** commands, and every breakpoint set or cleared and every continue or step is recorded in the controller's audit log with the name of the logged-in user.

24.2 Debug menu

Command	Key	Effect
↶ Step over	F10	Runs to the next statement at the same call depth or shallower; a call is executed whole
⤵ Step into	F11	Runs to the next statement at any depth — into the function block or function being called
↷ Step out	Shift+F11	Runs until the current block returns, and stops at the next statement of the caller
▶▶ Continue	F5	Runs normally until a breakpoint is reached again
● Breakpoints...	—	The breakpoint list, the call stack and Remove all

The step and continue commands are available only while the program is halted; at any other time they answer **The program is not stopped**. **F5** is taken by the IDE only while the program is halted; otherwise it remains the browser's reload key. The menu entry shows the number of breakpoints set, for example ● **Breakpoints (3)...**

24.3 Setting breakpoints

A breakpoint is set on a line of a PROGRAM's body in the Structured Text editor.

To set or remove a breakpoint:

1. Download the program and check that the header shows **IN SYNC**. Breakpoints are placed in the program the controller is running, by line number; a program that differs from the one on screen puts them on the wrong statements.
2. Log in as an engineer.
3. In the body pane of the POU editor, click the **line number** in the gutter. A breakpoint dot appears on the line, and a warning is added to the Messages panel: *Breakpoint at line N. When it is reached the program stops MID-SCAN...*
4. Click the line number again to remove it.

The controller's list is the authority. If the controller refuses a breakpoint, no dot is drawn and the reason is shown:

Message	Meaning
line N has no executable code, so the program would never stop there - put the breakpoint on a statement	A comment, a declaration, a blank line, or <code>END_IF</code>
no more than 32 breakpoints	The limit is 32 breakpoints per program
No controller to set a breakpoint on	The IDE is not connected
no ST program is loaded	Download a program first

Breakpoints are held by the **controller**, in the running program. They stay set when the IDE is closed, and they are removed when a new program is downloaded or the runtime restarts.

Important: In this version, set breakpoints only in the PROGRAM that runs **first** in MainTask's call list (or in the only program of the project). A breakpoint clicked in a later program of the call list, or in a FUNCTION_BLOCK or FUNCTION tab, is not placed on the line you clicked. To stop inside a function block, set the breakpoint on the line that calls it and use **Step into** (F11). Method tabs and Ladder POU's have no breakpoint gutter.

24.4 When the program stops

When execution reaches a breakpoint:

1. The program stops **before** executing the statement on that line.
2. Within about a second the IDE notices (it asks on its status poll), and:
 - the state pill shows **STOPPED** followed by the line number, for example **STOPPED 27**; hovering it explains the halt and shows the call chain;
 - the editor scrolls to the line and marks it in the gutter;
 - a warning is added to the Messages panel, for example *Stopped at line 27. The program is not running; the IO scan is, and outputs hold their last computed values. Called from: Plant line 12.*
3. The Variables pages and the live values beside the code show the values **at the halt** — mid-scan values, including those the halted scan has already changed.

If the halt is inside a function block or function — which are compiled ahead of the program, in the prelude (Chapter 20) — the pill reads **STOPPED in <block>**, the message says *inside library block <block> at its line N* (the same wording is used for the project's own blocks and for blocks of `lib/`), and the editor does not follow the halt. The Breakpoints dialog marks such frames (**library, not in this project**).

While halted you can read every variable, and you can write and force as usual (Chapter 22). A write takes effect when the program continues.

24.5 Stepping

From a halt:

- **F10 — Step over** executes the current statement, including any calls it makes, and stops at the next statement at the same level. At the end of the program it continues into the next scan (and the next program of the call list first) and stops at the next statement reached.
- **F11 — Step into** stops at the very next statement executed, which is inside the called function block or function when the current statement calls one.
- **Shift+F11 — Step out** runs the rest of the current block and stops in the caller.
- **F5 — Continue** releases the program. It stops again the next time a breakpoint line is reached — for a breakpoint inside a loop or in the main body, typically on the next scan.

Each step is a new halt: the pill, the editor mark and the Messages panel follow it.

24.6 The call stack and the breakpoint list

Debug → • **Breakpoints...** (or a click on the **STOPPED** pill) opens the **Breakpoints** dialog:

- the table of breakpoints — **Line**, the **Source** text of that line, and a **Remove** button for each;

- while halted, the **Call stack**, innermost first: the first row is **stopped in** the block and line where execution stopped, each further row is **called from**. A frame inside a library block is marked **(library, not in this project)** and its line is the block's own;
- while halted, **Currently stopped at line N after K stop(s)**;
- **Remove all** — clears every breakpoint and **releases a halted program**.

Clearing the last breakpoint always lets a halted program go: a program must never be left stopped with nothing to stop on.

24.7 Ending a debugging session

A halted program stays halted until someone continues it. It is **not** released by closing the IDE, by logging out or by a network interruption. To end a session:

1. Press **F5** (Continue), or open **Debug** → **• Breakpoints...** and press **Remove all**.
2. Check that the state pill shows **RUNNING** again, not **STOPPED <line>**.
3. Check that **• Breakpoints...** shows none.

A program download or a runtime restart also clears every breakpoint and releases the program.

24.8 Example

The function block and program below are small enough to step through on the simulator.

pou/Totaliser.st :

```
FUNCTION_BLOCK Totaliser
VAR_INPUT
    Flow   : REAL;      (* units per second *)
    Dt     : REAL;      (* seconds per scan *)
    Reset  : BOOL;
END_VAR
VAR_OUTPUT
    Total  : REAL;
END_VAR
IF Reset THEN
    Total := 0.0;
ELSE
    Total := Total + Flow * Dt;
END_IF;
END_FUNCTION_BLOCK
```

pou/Plant.st (the only program in the call list):

```

PROGRAM Plant
VAR
    FlowA, FlowB : REAL;
    TotA, TotB   : Totaliser;
    Totals       : ARRAY[1..2] OF REAL;
    Grand        : REAL;
    ResetAll     : BOOL;
    k            : INT;
END_VAR
TotA(Flow := FlowA, Dt := 0.01, Reset := ResetAll);
TotB(Flow := FlowB, Dt := 0.01, Reset := ResetAll);
Totals[1] := TotA.Total;
Totals[2] := TotB.Total;
Grand := 0.0;
FOR k := 1 TO 2 DO
    Grand := Grand + Totals[k];
END_FOR;
END_PROGRAM

```

To step through it:

1. Download `Plant` to the simulator and write `FlowA = 2.5` and `FlowB = 1.0` (Chapter 22).
2. Log in as an engineer and click the line number of `TotB(Flow := FlowB, ...)`.
3. Within one scan the pill shows **STOPPED** and the line. `TotA.Total` has been updated in this scan; `TotB.Total` has not.
4. Press **F11**. The halt moves into `Totaliser`: the pill reads **STOPPED in Totaliser**, and the message and the Breakpoints dialog show the call stack with `Totaliser` stopped in and `Plant` called from.
5. Press **Shift+F11** to return to `Plant`, then **F10** several times to walk through the `FOR` loop and watch `k` and `Grand` change on the Variables page.
6. Open **Debug** → • **Breakpoints...** and press **Remove all**. The program runs again.

24.9 Alternatives to breakpoints

On a machine that must keep running, find the fault without stopping the program:

Need	Tool
See a value change at scan rate	Trace (Chapter 23)
Capture a condition that lasts one scan	Latch it in the program (<code>IF cond THEN seen := TRUE; END_IF; </code>) and watch <code>seen</code>
Count how often something happens	A counter variable in the program, watched online
Know which code writes a variable	Where used... (Shift+F12, Chapter 25)
Test a change without hardware	The simulator (Chapter 26)

25. Diagnostics and messages

This chapter describes where the IDE and the controller report what is happening: the Messages panel, the controller log and its audit records, the IO device diagnostics, the scan statistics, the panel lamps, and the cross-reference (**Where used**). Use it when something does not behave as expected, and as the reference for what each indicator means.

25.1 Where to look

Question	Look at
Why did the build or download fail?	Messages panel (section 25.2)
What has the controller done since it started? Who changed what?	Controller log and audit records (section 25.3)
Is the controller running this project?	Sync badge and Compare... (Chapter 20)
Is the fieldbus healthy? Has this link been solid?	IO device page, tree badges, IO lamp (section 25.4)
Is the scan on time?	Task Configuration → Monitor (section 25.5)
Is anything forced?	FORCE badge, RUN lamp (Chapter 22)
Which code writes this variable?	Where used... (section 25.7)
Which versions are running?	Online → Controller → Versions... (section 25.8)

25.2 The Messages panel

The Messages panel is the strip at the bottom of the IDE window. Its title counts the messages: **Messages — 2 error(s), 1 warning(s), 5 message(s)**.

25.2.1 Columns

Column	Content
Description	The message, with a severity icon: X error, Δ warning, i note
Object	The file the message refers to (<code>Main.st</code> , <code>globals.st</code> , <code>gvl/Recipes.st</code> , <code>dut/Pump_T.st</code> , <code>IO aliases</code>), and the method name for a message inside a METHOD (<code>Pump.st</code> · <code>Start</code>)
Position	<code>Line N</code> , <code>Column M</code> , when the message has a position
Source	What produced it (section 25.2.4)

Hover a description to read it in full.

25.2.2 Severity filters and panel controls

Control	Effect
N error(s) chip	Show or hide errors

Control	Effect
N warning(s) chip	Show or hide warnings
N message(s) chip	Show or hide notes
×	Clear the list
▼ / ▲	Collapse or expand the panel

Drag the splitter above the panel to resize it. A build or download replaces the list with its own messages; most other actions (writes, forces, online events, the simulator) add to it.

25.2.3 Click to jump

Click a message to go to its cause:

Message refers to	The IDE opens
A POU, with a line	The POU at that line and column
A METHOD, with a line	The method's own tab at the method's line
<code>globals.st</code>	Global Variables , cursor on the line
<code>gv1/<name>.st</code>	That global variable list, cursor on the line
<code>dut/<name>.st</code>	That data type, cursor on the line
A POU, without a line, naming a symbol in quotes	The POU, at the declaration of that symbol
<code>IO aliases</code>	Nothing to open: the message names the device, card and channel instead

A POU is found by its short name even when it lives in a folder (`CM/MotorPump.st`), as long as the short name is unique.

Messages from the generated IO-alias block are rewritten to say which channel they concern, for example: *alias 'pump_run' (Rack1 / DO-16 Ch#3, %QX2.3): this name is declared in the program as well...* — the fix is in the program or in the device tree, not in a file.

25.2.4 Sources

Source	Produced by
<code>build</code> , <code>build all</code>	Build, Build All (Chapter 20)
<code>download</code>	Program and hardware downloads, including the transfer report
<code>globals</code> , <code>dut</code>	Check in the declaration editors
<code>upload</code> , <code>verify</code> , <code>compare</code> , <code>sync</code>	Upload, Verify, Compare... , sync checks
<code>online</code> , <code>monitor</code>	Writes from the variable tables and the editor; online-mode notices; a truncated symbol list
<code>force</code> , <code>write</code>	Forces and <code>%</code> writes from the Force / write table and dialogs

Source	Produced by
runtime	Minor faults reported by the controller (Chapter 21)
debug	Breakpoints set and halts (Chapter 24)
snapshot , data	Data snapshots and data saved with the project (Chapter 23)
simulator	Simulator start steps (Chapter 26)
screens	Download screens
restart , reboot , shutdown	The controller commands of Chapter 21
ide	An error inside the IDE itself — report it with the message text

25.2.5 Runtime messages in the panel

Some controller conditions are reported in the panel while you work, once per new occurrence:

Message	Meaning
line N: division by zero – K more (T since the program started). The scan kept running...	Minor fault (Chapter 21)
The controller's variable list is too large to send whole...	The symbol table was truncated (Chapter 22)
The controller is running a different hardware configuration from this project...	HARDWARE CHANGED (Chapter 20)
Stopped at line N...	A breakpoint was reached (Chapter 24)
Online, but the controller has no program loaded...	Download a program

25.3 The controller log and the audit trail

The runtime writes one log. Each line carries the controller's local time, the level and, for most records, the source file and line of the runtime that wrote it:

```
2026-09-27 11:14:02.118 [INFO ] scan_engine.c:422: scan engine running: period 10000 us, 1 sync
driver(s), 0 async driver(s)
2026-09-27 11:15:40.502 [AUDIT] monitor.c:1919: monitor: 192.168.1.20:51844 logged in as "avi"
(engineer)
2026-09-27 11:16:11.930 [AUDIT] monitor.c:2481: monitor write from 192.168.1.20:51844: Speed_Sp := 1450
2026-09-27 11:17:03.004 [WARN ] [rt +5412.118204s] st-minor-fault a=42 b=1
```

The timestamps come from the controller's clock. Set it with **Online** → **Controller** → **Set clock / time...** (Chapter 6) — a log with the wrong date is of little use after an incident.

Level	Content
TRACE , DEBUG	Development detail; off unless the log level is lowered

Level	Content
INFO	Start-up, configuration, programs loaded, devices connected
WARN	Something is wrong but the controller carries on: a device that did not come up, an overrun, a minor fault
ERROR	A fault: a program that will not compile at start, a critical device lost, a task fault
FATAL	The runtime cannot start
AUDIT	Who did what: logins, failed logins and lockouts, downloads accepted or rejected, writes, forces, run-state commands, breakpoints, alarm acknowledgements, signatures, configuration and retain events

Events raised from the scan thread (overruns, faults, minor faults) are written in a compact form — the time since the runtime started, an event tag and two numbers, for example `[rt +5412.118204s] overrun a=<μs late> b=<scan>` or `task-fault a=<code> b=<scan>` — because the scan thread must never format text:

Tag	a	b
overrun	how late the cycle finished, μs	scan number
overrun-fault	consecutive overruns	—
task-fault	error code	scan number
st-minor-fault	source line of the first minor fault	number in that scan
critical-io-lost, critical-io-never-came-up	device index	device state
fault-reset	scan number	—

25.3.1 Controller log...

Online → **Controller** → **Controller log...** (or ☰ on the action rail) reads the log without an SSH session. Any role may read it.

Control	Effect
Lines	How many of the most recent lines to read: 20 to 2000 (200 by default)
Refresh	Read again
warnings and errors only	Show only <code>WARN</code> , <code>ERROR</code> , <code>FATAL</code> and <code>AUDIT</code> lines
(status)	The log file's path on the controller and its size

The newest lines are at the bottom and the view scrolls there. If the controller has no log file configured, the dialog says so: output then goes only to the system journal, which needs SSH to read. Set **log.file** in the controller configuration (Appendix H).

25.3.2 The Events page

When the project has operator screens (Chapter 32), the tree has **HMI** → **Events**, a page with the same log **newest first**, with **Refresh**, **follow** (re-read every 3 s) and **audit only** (show only `AUDIT` lines). It reads the last 300 lines.

25.3.3 Where the log is kept

Setting (<code>log_</code>)	Meaning	Default
<code>file</code>	The log file	none (journal only)
<code>audit_file</code>	A separate file for <code>AUDIT</code> lines only, flushed on every record	none
<code>level</code>	Minimum level written: <code>trace</code> , <code>debug</code> , <code>info</code> , <code>warn</code> , <code>error</code>	<code>info</code>
<code>max_file_mb</code> , <code>max_files</code>	Size at which the file is rotated, and how many rotated files are kept	16 MB
<code>syslog_host</code> , <code>syslog_port</code> , <code>syslog_level</code>	Send records to a remote syslog collector; <code>AUDIT</code> records are always sent	off, 514, <code>error</code>

On an SD-card controller the recommended arrangement is `file` on a RAM file system and `audit_file` on flash: the operational log costs no card wear, and the audit trail survives a power cut. The audit trail on the controller can be appended to by the runtime only; it is lost with the card, so for a regulated system send it to a collector with `syslog_host` (Chapter 34).

25.3.4 Reading the audit trail

Writes, forces and run-state commands name the **connection** they came from (address and port). The login record of that connection names the user:

```
[AUDIT] monitor: 192.168.1.20:51844 logged in as "avi" (engineer)
[AUDIT] monitor force from 192.168.1.20:51844: %Q0.3
[AUDIT] monitor control from 192.168.1.20:51844: unforce_all -> ok
```

Breakpoints and steps name the user directly. Refused requests are recorded too: `monitor: DENIED "force" to "eve" (viewer) from ... - needs operator`.

25.4 IO diagnostics

25.4.1 Device states

Each IO device reports one of these states, shown as a badge on the device in the tree and on its page:

State	Meaning
<code>OPERATIONAL</code>	Exchanging data normally
<code>DEGRADED</code>	Partly working — for example some Modbus blocks answer and others do not, or a card is missing from a modular rack under the degrade module policy. The devices that are up keep receiving outputs
<code>FAULT</code>	Not exchanging. Its inputs read the fail-safe pattern (or hold, with hold_last_on_fault); its outputs are not written
<code>STOPPED</code>	Not started
<code>inhibited</code>	Configured and addressed, never started, not expected (Chapter 7)

25.4.2 The device page

When the controller is connected, a device's page shows a live status line, for example:

OPERATIONAL – 4/4 block(s) up, 0 error(s), 2 timeout(s).
Connected for 3 h 12 min · 1 reconnect(s) since the runtime started

Field	Meaning
state	As above
block(s) up	For block-based drivers (Modbus TCP): how many configured blocks are answering
error(s), timeout(s)	Counts since the runtime started
detail	The driver's own explanation of its state
Connected for	How long the current connection has lasted; Not connected when it is down
reconnect(s) since the runtime started	How many times the link has dropped and come back

Connected for and the reconnect count together answer the question asked on site — *has this link been solid since I got here?* A count alone cannot tell a rack that dropped an hour ago from one that dropped last week.

Fieldbus-specific diagnostics — EtherCAT slave states, Profinet alarms, IO-Link port events — are described in Chapters 9, 10 and 12.

25.4.3 Network interfaces

The tree node **Network Interfaces** lists the controller's ports with link state, speed, MTU, addresses and the bus they hang on (a USB adapter is shown as such; it adds latency no real-time tuning removes). Use it to answer "which port is the fieldbus on" and "is the cable in".

25.5 Scan statistics

Task Configuration (Chapter 18) has six tabs. Three of them are diagnostics:

25.5.1 Monitor

Field	Meaning
Running for	Time since the last runtime restart (cycles × period)
Cycles	Scans executed since start
Overruns	Scans that did not finish before the next deadline. Shown highlighted when not zero. Should be zero
Jitter avg, Jitter max	Wake-up latency: actual start of the cycle minus its intended deadline, in µs
Exec last	Duration of the last scan: input read, program, output write

Jitter is what decides whether a control loop is deterministic, and it is the number to show a customer; the maximum matters more than the average. The simulator reports these figures too, but they are not comparable with a controller's (Chapter 26).

25.5.2 Variable Usage, Stack Usage, System Events

Tab	Shows
Variable Usage	Counts of the running program's symbols by storage: <code>address</code> (in the process image), <code>memory</code> (program and global data), <code>instance</code> (function-block instances)
Stack Usage	The per-scan limits: instruction budget 200 000, operand stack 256, call depth 32, function frames 64 KB
System Events	The scan engine's state and the number of forces, and each IO device's state with its detail

25.6 Panel lamps

The RUN and IO lamps on the controller front are described in Chapter 21, section 21.7. In short: RUN on = running; RUN fast blink = running with forces; RUN and IO blinking alternately = FAULT; IO on = all remote IO up; IO slow blink = a device is not up.

25.7 Where used (cross-reference)

Where used... lists every place a name appears in the project, classified by what the code does with it.

To find where a variable is used:

1. In a POU editor, put the caret on the name.
2. Press **Shift+F12**, or click **Where used...** in the POU toolbar, or right-click → **Where used...**
3. The dialog **Where is <name> used?** lists the hits: *N declaration(s) · N write(s) · N read(s)*.
4. Click a row to open the file at that line.

Column	Content
Use	declaration, write or read
Scope	The POU the line belongs to, <code>globals</code> for <code>globals.st</code> , or <code>(file scope)</code>
File, Line	Where it is
Code	The line

The search covers `globals.st`, every global variable list and every POU. Mentions in comments and string literals are ignored. Data-type files and library blocks are not searched.

If the name is declared more than once, the dialog says so first: **<name> is declared N times**, with where. Inside a POU that declares it, the local declaration wins — while the variable tables, the ladder display and this page resolve the name to the first match, which may be the other one. Two variables of one name then disagree on screen and nothing looks wrong. Rename one of them (**F2**).

If nothing uses the name, the dialog says so: a tag that nothing reads or writes is either dead, or spelled differently where it is used.

25.7.1 Example: two writers

```

VAR_GLOBAL
    Pump_Run : BOOL;
END_VAR

PROGRAM Auto
VAR
    Level_Low : BOOL;
END_VAR
Pump_Run := NOT Level_Low;          (* write #1 *)
END_PROGRAM

PROGRAM Manual
VAR
    Hand_Mode : BOOL;
    Hand_Run  : BOOL;
END_VAR
IF Hand_Mode THEN
    Pump_Run := Hand_Run;           (* write #2 - wins when both run *)
END_IF;
END_PROGRAM

```

(globals.st, pou/Auto.st and pou/Manual.st, with Auto before Manual in the call list.)

Where used... on Pump_Run shows one declaration and **two writes**, in Auto and in Manual. The pump follows Manual whenever Hand_Mode is TRUE, because Manual runs later in the scan; with Hand_Mode FALSE it follows Auto. Two writers of one output are legal and are usually a mistake: before you change a value on a running machine, check who else writes it.

25.8 Versions and compatibility

Online → **Controller** → **Versions...** shows the IDE version, the runtime version and product, the monitor protocol versions of both, and whether they are compatible. Quote it in every support request.

25.9 Checklist: something is wrong

1. Read the header: target, state pill, sync badge, **FORCE** badge.
2. Read the Messages panel; click the first error.
3. Open **Controller log...** with **warnings and errors only**.
4. Open the page of any IO device that is not **OPERATIONAL**; note its reconnect count and **Connected for**.
5. Look at **Task Configuration** → **Monitor** for overruns and jitter.
6. Use **Where used...** on the variable that misbehaves.
7. Record a trace of the signals involved (Chapter 23).
8. Save a data snapshot before anyone resets anything.

26. The simulator

This chapter describes the simulator: a Zeus runtime running on the engineering PC, with this project's IO configuration and no hardware. Use it to test a program, a ladder, the addresses and the operator screens before they meet a controller, to rehearse a download, and to connect a SCADA system to a running program on a desk.

26.1 What the simulator is

The simulator is the **real runtime**, not a model of it: the same scan engine, process image, Structured Text compiler and virtual machine, alarm engine and monitor server that run on the controller, compiled from the same source. The IDE connects to it exactly as it connects to a controller: it downloads, goes online, writes, forces and debugs in the same way.

What is replaced is the IO. Every enabled IO device of the project is replaced by the **sim driver**, which keeps the device's **name** and its exact **%I and %Q sizes**. The process image, every address and the symbol table are therefore identical to the controller's, so a program tested on the simulator is tested against the addresses it will really use. The simulated devices reach OPERATIONAL, and their inputs can be made to move (section 26.4).

The simulator answers *is this configuration and this program coherent?* It does not answer *does this machine work?*: it does not model the plant, and it does not prove the fieldbus (section 26.8).

26.1.1 Two simulator programs

Which runtime the IDE starts depends on what is installed:

Runtime	Where it comes from	What it contains
zeus-sim.exe	The Windows package (bin\zeus-sim.exe)	The scan engine, process image, compiler and VM, alarms, retain, batch context and the monitor server. No OPC UA server, Modbus TCP slave, EtherNet/IP tag server, operator-screen web server or fieldbus drivers
zeus-plcd (Linux build)	A Linux build of the runtime, run under WSL2 on a Windows engineering PC or natively on Linux	The complete controller runtime, including the OPC UA server, the EtherNet/IP tag server, the Modbus TCP slave and the web server

The IDE prefers the complete runtime when one is present and falls back to `zeus-sim.exe`. When it starts `zeus-sim.exe`, the first step in the Messages panel says so: *the Windows simulator: no OPC UA server, no Modbus slave, no EtherNet/IP tag server, no fieldbus. The scan engine, the program and the monitor are the real ones.*

Important: To connect a SCADA system to the simulator by OPC UA or EtherNet/IP (section 26.6), the complete runtime is needed. The Windows package's `zeus-sim.exe` does not contain those servers.

26.2 The Simulator page

Open the tree node **Simulator** (under the controller, after **Controller Settings**). Its badge shows **running** or **stopped**.

Element	Meaning
Status	RUNNING with its address (<code>127.0.0.1:<port></code>), or STOPPED
Start / Restart	Builds the simulation from the project and (re)starts it
Stop	Stops the simulator
Refresh	Re-reads the status, the device list and the log
Monitor port	The port the IDE uses to talk to the simulator (5020 by default)
Generate moving input signals	On: inputs move (section 26.4). Off: inputs hold 0
Scan cycle (µs)	The update cycle of the simulated IO devices; empty uses each device's own cycle, or 1000 µs
Simulated devices	One line per device: what it stands in for, its <code>%I</code> / <code>%Q</code> sizes and the number of generated signals
Runtime log	The last lines the simulator printed

The configuration fields are remembered by the browser and take effect on the next **Start / Restart**. The task period — the scan cycle of the program — is not set here; it comes from **Task Configuration** as on the controller.

26.2.1 Starting the simulator

To start the simulator:

1. Save the project. The simulator is built from the saved controller configuration.
2. Open **Simulator** in the tree and set the options.
3. Press **Start / Restart**.
4. Read the steps in the Messages panel (source `simulator`), each marked ✓, ⚠ or ✗:

Step	Content
runtime binary	Which runtime was chosen (see section 26.1.1)
simulator is STALE (⚠, development installations only)	The runtime binary is older than its source; rebuild it
configuration	Where the simulated configuration was written
device (one per device)	<code><device>: <driver> simulated, %I n B / %Q m B, k generated signal(s)</code>
previous simulator	Any simulator already running was stopped
simulator running	<i>listening on 127.0.0.1:<port></i> , or the last lines of the runtime's output if it did not come up

5. When it has started, the IDE selects the **Simulator (local)** target and connects to it, keeping your login. The header's target badge shows **SIMULATOR**.
6. Download the program (**F8**) — the simulator starts without one (section 26.3.2).

The IDE considers the simulator started only when the process on the port answers as a Zeus runtime; a port held by something else is reported as a failure, not as success.

To stop it, press **Stop**. Selecting another target with **Target...** does not stop the simulator.

26.2.2 Hardware download and Restart runtime

With the simulator selected as the target:

- **Download hardware** rebuilds the simulation from the current project configuration and restarts the simulator — the equivalent of a hardware download to a controller, without the warning about outputs, because there are none. **Download all** does this and then downloads the program.
- **Online → Controller → Restart runtime...** relaunches the simulator.
- **Reboot controller (OS)**, **Shut down controller** and **Set clock / time** are not available: the simulator has no operating system of its own and uses the PC's clock.

Note: A hardware download or **Restart runtime** builds the simulation with the default settings — moving input signals on, each device's own IO cycle — whatever the Simulator page is set to. Use **Start / Restart** on the Simulator page to apply your settings.

26.3 How the project is transformed

The simulator runs a copy of the project's configuration with these changes, and no others:

Item	In the simulation
Controller name	<name> (simulated)
Each enabled IO device	Driver replaced by <code>sim</code> , same name, <code>%I / %Q</code> sizes exactly as the real driver would ask for them (Chapter 7, Section 7.8 - a device that already uses the <code>sim</code> driver keeps its input_bytes / output_bytes , default 32), so every device after it starts at the same address as on the controller; the real driver's parameters dropped
A device whose size cannot be derived	Disabled, with a note: scan it or lay it out from its ESI/GSDML so that the size is known
Disabled devices	Stay disabled
Monitor	Bound to <code>127.0.0.1</code> on the chosen port, writes allowed, no token
Accounts (users and roles)	As in the project: log in as on the controller
Retain file, log file	In the IDE's <code>runtime\sim\</code> directory, never the controller's
Task call list	Programs excluded from the build are removed
OPC UA, EtherNet/IP tag server, Modbus slave, web server	As configured in the project (complete runtime only)

The simulated configuration is written to `runtime\sim\sim-active.json`, and the simulator's console output to `runtime\sim\sim.out`, under the IDE's installation directory.

26.3.1 Addresses and signal ranges

Because sizes and order are kept, `%IW0` on the simulator is the same channel as `%IW0` on the controller, and IO aliases bind to the same addresses (Chapter 11). Analogue scaling configured for a channel (signal ranges) is

applied as on the controller.

26.3.2 The program in the simulator

The simulator does not keep a boot application between starts. After every **Start / Restart**, hardware download or **Restart runtime** it starts with **no program**; the online views say *the controller has no program loaded*. Download the program again, or use **Download all**, which does both in one step.

Retained variables in the simulator are written to its own retain file. Do not expect them to survive a restart of the simulator: test retain behaviour on a controller (Chapter 33).

26.4 Simulating inputs and outputs

26.4.1 Generated signals

With **Generate moving input signals** on, the simulator drives the **inputs** of every simulated device so that something moves at every input address:

Channel kind	Signal
Digital inputs (1-bit blocks)	One square wave over the card's bytes, so every channel toggles; each card has its own period (from 0.9 s upwards)
Analogue inputs (16-bit and wider)	Alternately a sine and a ramp between 0 and 27 648 (the raw full scale that the library's scaling blocks assume), each channel with its own period (from 1.7 s upwards, in steps of 313 ms)

Every signal has a different period, so two channels never read alike for long. If two different addresses show the same value in the simulator, suspect an addressing mistake — two variables aliased to one channel.

At most 64 signals are generated per device; channels beyond that hold 0. Outputs are never generated: they are written by your program, as on the controller.

The values are not plant data. A program whose logic depends on realistic process behaviour — a level that rises when a valve opens — needs that behaviour written in a simulation program (section 26.7).

26.4.2 Driving inputs by hand

With generation off, every input reads 0 until you change it:

- **Digital inputs:** force them (Chapter 22) — in the variable tables with **F7**, or by address in **Debug** →  **Force / write table...** ($\%IX0.3 = 1$). Release with **Alt+F7** or  **Clear forces**.
- **Analogue inputs:** $\%I$ words cannot be written, only bits forced. Either leave generation on, force the individual bits of the word in the Force / write table, or read the channel through a variable that a simulation program can set (section 26.7).

26.4.3 Outputs

Watch $\%Q$ in the variable tables, the ladder view or the trace. Writing a $\%Q$ word from the Force / write table holds only while the program does not assign that channel.

26.5 Using the simulator

26.5.1 Procedure: rehearse a project before site

1. Save the project and **Build All**; clear every error.
2. Open **Simulator**, leave **Generate moving input signals** on, press **Start / Restart**.
3. Check every **device** line: the sizes must be the ones you expect on the controller. A disabled device means its size is unknown.
4. Press **Download all** (or **F8**). The header shows **SIMULATOR, RUNNING** and **IN SYNC**.
5. Go online (**Online** → **Monitor**). Open the Variables pages and check that each input alias moves and that each output follows the logic.
6. Exercise the sequences with writes and forces (Chapter 22); record traces of the control loops (Chapter 23).
7. Save a data snapshot and the traces as the rehearsal record.
8. Select the controller with **Target...** before the real download.

26.5.2 Example: checking the addresses

The program below assumes a device whose input image starts with one analogue channel (%IW0) followed by one digital input byte (%IB2), and whose output image starts with a digital output byte and an analogue output word. On the simulator `Level_Pct` sweeps between 0 and 100, `Do_Lamp` follows it, and `Changes` counts the square wave.

```
PROGRAM SimCheck
VAR
  Ai_Raw      AT %IW0   : INT;      (* generated sine/ramp 0..27648 in the simulator *)
  Di_Byte     AT %IB2   : BYTE;     (* generated square wave *)
  Do_Lamp     AT %QX0.0 : BOOL;
  Ao_Out      AT %QW0   : INT;
  Level_Pct   : REAL;
  Changes     : DINT;
  LastDi      : BYTE;
END_VAR
Level_Pct := INT_TO_REAL(Ai_Raw) * 100.0 / 27648.0;
Do_Lamp := Level_Pct > 50.0;
Ao_Out := Ai_Raw;          (* Loop the input straight out *)
IF Di_Byte <> LastDi THEN
  Changes := Changes + 1;
END_IF;
LastDi := Di_Byte;
END_PROGRAM
```

In a real project, use IO aliases instead of `AT` declarations (Chapter 11); the test is the same.

26.6 Connecting a SCADA system to the simulator

With the complete runtime (section 26.1.1), the simulator runs the project's OPC UA server, EtherNet/IP tag server and Modbus TCP slave exactly as configured, so a SCADA system — for example OptiZeus SCADA — can be developed and tested against a running program before the controller exists.

Note: A Modbus TCP slave counts as real IO: without a license for this PC the complete runtime stops the program after 2 hours (Chapter 37). The license for a PC's simulator is kept in `runtime\sim\license.json`.

Service	Port (default)	Configure in	See
IDE ↔ runtime (monitor)	5020 (Simulator page)	Simulator → Monitor port	—
OPC UA server	4840	OPC UA Server node	Chapter 29
EtherNet/IP tag server	44818	EtherNet/IP tag server settings	Chapter 30
Modbus TCP slave	5502 (bound to 127.0.0.1 unless changed)	Modbus TCP Slave node	Chapter 31
Operator screens (web server)	8080	HMI → Server	Chapter 32
The IDE itself	8090	—	Chapter 2

To connect OptiZeus SCADA by OPC UA:

1. Enable the OPC UA server in the project and choose what it publishes (Chapter 29).
2. Save, then **Start / Restart** the simulator and download the program. The OPC UA address space is built from the running program, so reconnect the client after every download that adds or removes variables, and after every restart.
3. In OptiZeus, add an OPC UA connection to `opc.tcp://127.0.0.1:4840` (or the engineering PC's address, if the SCADA runs on another machine and the port is reachable) with the security settings the project's server uses.
4. Browse the address space and bind the tags.

To connect by EtherNet/IP (tags by name):

1. Enable the EtherNet/IP tag server in the project (Chapter 30).
2. Start the simulator and download the program.
3. In OptiZeus (or any EtherNet/IP (CIP) client such as an HMI), add a connection to the engineering PC's address on port 44818 and read the tags by name.

Note: When the complete runtime runs under WSL2 on a Windows PC, Windows forwards its ports to `127.0.0.1` only if no Windows program already listens on them. A local OPC UA discovery server on port 4840, or another Modbus server on the slave port, hides the Zeus server; stop it or use another port. A client on another computer may not reach ports inside WSL2 at all; run the SCADA on the same PC, or run the complete runtime on a Linux machine.

The simulator binds the monitor port to `127.0.0.1` only, so no other computer can download to it or force it. The OPC UA, EtherNet/IP, Modbus and web servers use the bindings configured in the project.

26.6a Connecting a 3D process simulation (Factory I/O)

A simulation program (26.7) makes the plant react; a **3D process simulation** shows it. Factory I/O (factoryio.com, a separate product with a free trial) is a virtual factory - conveyors, sensors, push buttons, tanks - that a PLC drives over Modbus TCP. The simulator connects to it for real:

- a **Modbus TCP device whose address is this PC** - `127.0.0.1`, any `127.x.x.x`, `localhost` - is **not** simulated: the simulator really connects to it. Every other device is simulated as before;

- such a device is a process simulation, not a machine, so it is **not real IO** for the license: no time limit (Chapter 37).

The ready example **F1-FactoryIO-Conveyor (Project → Open an example project...)** drives a conveyor with a start and a stop button, an end sensor and a box counter on a display. Its README lists the scene.

In Factory I/O

1. Build the scene, or open one. F1 needs a Belt Conveyor, an Emitter, a Diffuse Sensor at the end, a Remover, a Start Button, a Stop Button and a Digital Display.
2. **File → Drivers**, choose **Modbus TCP/IP Server**. Configuration: host `127.0.0.1`, port `502`, the same slave/unit id as the device in the IDE (F1 uses 1); 16 digital inputs and outputs, 8 register inputs and outputs.
3. Drag each tag of the scene onto an input or output of the driver (F1: Input 0 = Diffuse Sensor, Input 1 = Start Button, Input 2 = Stop Button; Output 0 = Belt Conveyor, Output 1 and 2 = the button lights, Register output 0 = Digital Display). Press **Play**.

In the IDE

4. Open the project; the IO device **FactoryIO** is a **Modbus TCP** device on `127.0.0.1:502` with four blocks:

Factory I/O	Modbus	Zeus address
Input n (sensors, buttons)	discrete input n	$\%IX (n/8) \cdot (n \bmod 8)$ - Input 0 is $\%IX0.0$, Input 9 is $\%IX1.1$
Register input n	input register n	$\%IW (2+2n) - \%IW2, \%IW4, \dots$
Output n (actuators, lights)	coil n	$\%QX (n/8) \cdot (n \bmod 8)$
Register output n	holding register n	$\%QW (2+2n)$

5. **Simulator → Start / Restart**, then **Download all**. The device shows **OPERATIONAL**; press Start in Factory I/O and the belt runs.

Note: The simulator in the IDE package (`zeus-sim.exe`) runs on Windows, where Factory I/O runs, so `127.0.0.1` reaches it. A development PC that runs the simulator inside WSL has its own `127.0.0.1`: give the device the Windows host's address instead (then it counts as real IO), or use WSL's mirrored networking.

Note: For your own scene, add a Modbus TCP device with host `127.0.0.1` (IO Devices → Add device → Modbus TCP) and the four blocks above, then name the addresses in Global Variables.

26.7 Simulating the process

For logic that needs the plant to react — a tank that fills, a motor feedback that follows its command — write a small simulation program and add it to the call list while you test. Keep it separate from the machine program, and remove it from the call list (or right-click it → **Disable**) before the controller download.

A simple pattern is to route every input through a variable that the machine program reads, set either from the IO alias or from the simulation program:

```

VAR_GLOBAL
    SimMode      : BOOL := TRUE;    (* FALSE on the real controller *)
    Tank_Level   : REAL;            (* what the machine program reads, 0..100 % *)
    Valve_In_Cmd : BOOL;            (* written by the machine program *)
    Pump_Out_Cmd : BOOL;

END_VAR

PROGRAM TankSim
VAR
    Level : REAL := 20.0;
END_VAR
IF SimMode THEN
    IF Valve_In_Cmd THEN
        Level := Level + 0.05;      (* filling, per scan *)
    END_IF;
    IF Pump_Out_Cmd THEN
        Level := Level - 0.08;      (* emptying, per scan *)
    END_IF;
    IF Level < 0.0 THEN Level := 0.0; END_IF;
    IF Level > 100.0 THEN Level := 100.0; END_IF;
    Tank_Level := Level;
END_IF;
END_PROGRAM

```

On the controller, a program earlier in the call list sets `Tank_Level` from the scaled analogue input when `SimMode` is FALSE.

26.8 Limitations compared with a controller

Area	Simulator	Controller
Real time	No real-time scheduling on Windows; with <code>zeus-sim.exe</code> the task period floor is 1 ms. Jitter and execution times are reported but not comparable with a controller's	PREEMPT_RT, isolated core; period floor 0.5 ms
Field	Generated or zero inputs; no plant behaviour	Real signals
Fieldbus	Not exercised: coupler module lists, cabling, station names, EtherCAT states, Profinet alarms and IO-Link are not simulated	Real
Scanning for devices	Not possible without the real network; <code>zeus-sim.exe</code> refuses with <i>not on a simulator</i>	Available
Program kept across restarts	No — download after every start	Yes (boot application)
Retained data across restarts	Not to be relied on	Yes
Program signing	<code>zeus-sim.exe</code> cannot verify signatures: every program counts as unsigned (refused under enforce)	As configured
OPC UA, EtherNet/IP tag server, Modbus slave, web server	Complete runtime only	Yes

Area	Simulator	Controller
Trace	Complete runtime. <code>zeus-sim.exe</code> in this version arms a trace but records no samples	Yes
Reboot, shut down, clock	Not available	Available
Panel lamps, power-fail monitor, redundancy	Not present	Present
Monitor access	<code>127.0.0.1</code> only, no token	As configured

26.9 zeus-sim.exe from the command line

`zeus-sim.exe` can also be started without the IDE, for example on a test PC:

```
zeus-sim --config FILE [--program FILE]
```

- `--config` — a controller configuration file (for example the IDE's `runtime\sim\sim-active.json`). Every IO device in it is simulated, whatever its driver.
- `--program` — an ST source file to load at start, instead of the configuration's boot program.
- `--version` — print the version.

It prints its log to the console, runs until it is stopped with Ctrl+C, and serves the monitor on the configured address and port, so an IDE can connect to it as a target.

26.10 Troubleshooting

Symptom	Cause	Action
simulator running <i>X</i> with <i>another simulator is already on this port</i>	A previous simulator, or another program, holds the monitor port	Press Stop , then Start / Restart ; or choose another Monitor port
simulator running <i>X</i> , <i>did not come up</i>	The runtime rejected the configuration or exited	Read the quoted lines and the Runtime log on the Simulator page
A device is disabled in the simulation	Its process-image size is unknown	Give it fixed sizes, scan it, or lay it out from its ESI/GSDML
<i>the controller has no program loaded</i>	The simulator was (re)started	Download the program
Inputs never change	Generate moving input signals is off, or the device has no input blocks	Turn generation on and Start / Restart , or force the inputs
OPC UA client finds no server on 4840	<code>zeus-sim.exe</code> is running (no OPC UA server), the server is disabled in the project, or another program owns 4840	See sections 26.1.1 and 26.6
Header shows HARDWARE CHANGED on the simulator	The simulation was built with settings other than the defaults, or the configuration changed after the start	Start / Restart , or Download hardware

Symptom	Cause	Action
Jitter figures look poor	Windows is not a real-time system	Measure timing on a controller

ZEUS PLC IDE · OPERATION MANUAL

Part VI — Integration and maintenance

27. Alarms

This chapter describes the alarm engine built into the controller runtime: how an alarm is defined, how it moves through the ISA-18.2 states, how operators acknowledge, shelve and suppress alarms, and what the alarm journal records. It also covers the **ALARM_AN** library block for analogue limit alarms and the separate, screen-level alarm lists of the operator screens. Read it before you define the first alarm of a project, and again before you write an alarm philosophy for a validated plant.

27.1 How alarms work in Zeus

27.1.1 Raised in the controller, not in a screen

An alarm on a Zeus controller is evaluated **on the scan thread**, at the end of every scan cycle, and timestamped with the controller's own clock at the moment its state changes. A screen, the IDE or a SCADA system only displays a record the controller has already written.

This matters when an excursion is investigated. A screen that polls twice a second behind a 10 ms scan notices a change up to half a second late, and two alarms on two screens can be recorded in the wrong order. Timestamping at source keeps the order of events correct, which is the first question in any investigation: *what happened first?*

27.1.2 The condition lives in the program

The alarm definition does **not** contain the condition. The program computes the condition — `level > 9000 AND pump_running` is logic, and logic belongs in the program — and writes the result to a bit in the process image. The alarm engine watches that bit.

Part	Where it lives	Who changes it
Condition (<code>TT_01 > 85.0</code> , limits, hysteresis, interlocks)	The ST / Ladder program	Engineer, by download
Alarm identity, text, priority, on-delay, latching, suppression bit	<code>controller.json</code> , section "alarms"	Engineer, by Download hardware
Acknowledge, shelve, out-of-service	The controller, at run time	Operator / engineer, recorded with their name

The cyclic cost is one bit test per alarm per scan, and the alarm's meaning can be read in the program rather than reconstructed from a configuration file.

27.1.3 Two kinds of alarm list in a project

A Zeus project can contain two different things that are both called "alarms". Do not confuse them.

	Controller alarms (this chapter, §27.2–27.8)	Screen alarm lists (§27.10)
Defined in	<code>controller.json</code> → "alarms"	An Alarm list widget on an operator screen
Evaluated by	The runtime, every scan	The screen, each time it polls

	Controller alarms (this chapter, §27.2–27.8)	Screen alarm lists (§27.10)
Timestamped	Yes, at source, with clock-trust flag	No
States, acknowledge, shelve, journal	Yes (ISA-18.2)	No — shows which conditions are TRUE now
Where you see them	Online → Alarms...	On the screen; overview in HMI → Alarms
Suitable as a record	Yes	No

Use controller alarms for anything that must be acknowledged, recorded or investigated. Use screen alarm lists only as a live "what is wrong now" panel on a screen.

27.2 The state model (ISA-18.2)

An ISA-18.2 alarm is two independent facts — the condition (active or not) and the acknowledgement (acknowledged or not) — plus three states in which the alarm is deliberately not presented. The runtime uses these seven states:

State	Condition	Acknowledged	Meaning
NORMAL	not active	yes	Nothing to show.
UNACK_ACTIVE	active	no	Active, and nobody has seen it yet.
ACK_ACTIVE	active	yes	Active, and an operator has acknowledged it.
UNACK_RTN	returned to normal	no	The condition went away before anybody acknowledged it. Kept, not forgotten.
SHELVED	not evaluated	—	An operator silenced it for a limited time.
SUPPRESSED	not evaluated	—	Suppressed by design: the configured suppression bit says the alarm is meaningless now.
OUT_OF_SERVICE	not evaluated	—	Removed from service by maintenance (engineer only).

Transitions.

From	Event	To
NORMAL	condition true for the on-delay	UNACK_ACTIVE (count +1, journal RAISE)
UNACK_ACTIVE	operator acknowledges	ACK_ACTIVE (journal ACK)
UNACK_ACTIVE	condition clears (not latched)	UNACK_RTN (journal CLEAR)
ACK_ACTIVE	condition clears (not latched)	NORMAL (journal CLEAR)
UNACK_RTN	operator acknowledges	NORMAL (journal ACK)
any evaluated state	operator shelves for <i>t</i>	SHELVED (journal SHELVE)

From	Event	To
SHELVED	time expires	UNACK_ACTIVE if still active, else NORMAL (journal UNSHELVE , who = expired)
SHELVED	operator unshelves	as above (journal UNSHELVE , who = operator)
any	suppression bit TRUE	SUPPRESSED (journal SUPPRESS , who = design)
SUPPRESSED	suppression bit FALSE	NORMAL , condition re-evaluated from scratch (journal UNSUPPRESS)
any	engineer takes out of service	OUT_OF_SERVICE (journal OUT_OF_SERVICE)
OUT_OF_SERVICE	engineer returns to service	NORMAL (journal IN_SERVICE)

Note: A condition that returns to normal is never silently discarded. UNACK_RTN exists so that "it went away before anybody saw it" and "somebody saw it and it is still active" remain different facts on the operator's screen and in the record.

27.3 Priorities

There are four priorities. More than four cannot be distinguished during an alarm flood, and the middle ones stop being used.

Priority	Config value	Intended use
CRITICAL	"critical"	Act now.
HIGH	"high" (the default)	Act promptly.
LOW	"low"	Act when convenient.
DIAGNOSTIC	"diagnostic"	Engineering interest only.

The runtime does not assign priorities. Assigning them by consequence analysis and rationalising the alarm set is project work; the runtime's job is an honest record.

27.4 Defining alarms

27.4.1 The "alarms" section

Controller alarms are defined in the project's `controller.json`, in a top-level array named `"alarms"`. There is no form editor for this section in the current IDE: edit the file with a text editor.

```

"alarms": [
  { "id": 1, "name": "TT_01_HH", "message": "Reactor temperature high-high - heating stopped",
    "priority": "critical", "area": "M", "offset": 10, "bit": 0, "delay_ms": 0 },
  { "id": 2, "name": "TT_01_H", "message": "Reactor temperature high",
    "priority": "high", "area": "M", "offset": 10, "bit": 1, "delay_ms": 500 },
  { "id": 3, "name": "TT_01_L", "message": "Reactor temperature low",
    "priority": "low", "area": "M", "offset": 10, "bit": 2,
    "suppress": { "area": "M", "offset": 10, "bit": 7 } },
  { "id": 4, "name": "ESTOP_SEEN", "message": "Emergency stop was operated",
    "priority": "high", "area": "I", "offset": 0, "bit": 5,
    "invert": true, "latched": true }
]

```

Key	Type	Required	Default	Meaning
<code>id</code>	integer > 0	yes	—	Stable identity. Every journal entry and acknowledgement refers to it. Must be unique; a duplicate is a configuration error. Do not renumber alarms in a running plant.
<code>name</code>	string	yes	—	Short tag shown to operators and carried in the record (up to 63 characters).
<code>message</code>	string	no	the <code>name</code>	Operator text (up to 119 characters). Say what is wrong and, where possible, what to do.
<code>priority</code>	string	no	"high"	"diagnostic", "low", "high" or "critical" (case-insensitive).
<code>area</code>	string	no	"M"	Process-image area of the condition bit: "M", "I" or "Q".
<code>offset</code>	integer ≥ 0	yes	—	Byte offset of the condition bit in that area.
<code>bit</code>	0...7	yes	—	Bit number within the byte. "offset": 10, "bit": 1 is %MX10.1.
<code>invert</code>	bool	no	false	Alarm when the bit is FALSE (for a normally-closed signal).
<code>delay_ms</code>	number ≥ 0	no	0	On-delay: the condition must be continuously true this long before the alarm is raised. Fractions of a millisecond are accepted.
<code>latched</code>	bool	no	false	The alarm stays active after the condition clears, until it is acknowledged. For events too brief to see.
<code>suppress</code>	object	no	—	{ "area", "offset", "bit" } of a suppression bit. While that bit is TRUE the alarm is SUPPRESSED and not evaluated.

Limits. Up to **10,000** alarms per controller. The runtime reserves memory for them at start-up, never from the scan.

Validation. The runtime refuses to start with a configuration that has a missing or non-positive `id`, a duplicate `id`, an empty `name`, an unknown `priority`, an area other than `M`, `I` or `Q`, a missing `offset`, a `bit` outside 0...7, a negative `delay_ms` or an incomplete `suppress` object. The error names the entry, for example `alarms[3].bit` must be 0..7, got 8.

Tip: Use **Controller Settings** → **Validate with the runtime** after editing `controller.json` by hand. It runs the runtime's own configuration check and prints its verdict and messages, so a typo is found at the desk rather than at the next start of the controller.

Important: The IDE keeps a copy of `controller.json` in memory. If you edit the file by hand while the project is open, press the tree's refresh button before you save any settings page (Controller Settings, OPC UA Server, Modbus TCP Slave, Screens Server). Otherwise that page writes back the configuration it loaded earlier, and your hand edits are lost.

27.4.2 Choosing addresses for alarm bits

Alarm bits are normally marker bits (`%MX`). Reserve a block of `%M` for them and document it, for example:

Range	Use
<code>%MB10 ... %MB59</code>	Alarm condition bits, 8 per byte (400 alarms)
<code>%MB60 ... %MB69</code>	Suppression bits
<code>%MD200 , %MW204</code>	Batch context (reserved by <code>S88_CONTEXT</code> , see Chapter 28)

The default size of `%M` is 4096 bytes (**Controller Settings** → **%M marker bytes**).

An alarm bit may also be an input (`"area": "I"`) — useful for a hard-wired contact that needs no logic — or an output (`"area": "Q"`).

Note: `%M` is retained across power cycles (Chapter 33). An alarm bit in `%M` therefore keeps the value it had at power-off until the program writes it again in the first scan. Write every alarm bit unconditionally in every scan, as the examples in this chapter do, rather than only setting or resetting it on events.

27.4.3 On-delay, latching, invert and suppression

On-delay. A condition that flickers for one scan is noise, and noise in an alarm list teaches operators to ignore it. `delay_ms` starts counting when the condition bit becomes TRUE and restarts if it drops. The alarm returns to normal immediately when the condition clears — there is no off-delay. If you need hysteresis on an analogue value, compute it in the program (the **ALARM_AN** block does this, §27.9).

Latching. A latched alarm remains active after its condition clears until it is acknowledged. Use it for events that can be shorter than an operator's attention span — a pressure spike, a momentary loss of a permissive.

Important: Acknowledge a latched alarm **after** its condition has cleared. In this runtime version, a latched alarm that is acknowledged while its condition is still TRUE goes to `ACK_ACTIVE` and stays there after the condition clears, because a further acknowledgement is ignored for an alarm that is already acknowledged. To return such an alarm to `NORMAL` , take it out of service and back in (engineer), or restart the runtime.

Invert. `"invert": true` raises the alarm when the bit is FALSE. Use it for normally-closed, fail-safe wiring: a broken wire then raises the alarm.

Suppression by design. Suppression is the engineer declaring that an alarm means nothing in a particular state — a low-flow alarm on a pump that is stopped. It is part of the configuration, not an operator action, and it comes back by itself: when the suppression bit returns to FALSE the alarm goes to `NORMAL` and its condition is evaluated from scratch, so a condition that is still present is raised again (after its on-delay) as a new alarm.

```

PROGRAM AlarmBits
VAR
    (* The alarm engine watches bits; the program decides what they mean. *)
    TT_01_HIGH   AT %MX0.0 : BOOL;
    LOW_FLOW     AT %MX0.2 : BOOL;
    PUMP_STOPPED AT %MX0.3 : BOOL;
    reactor_temp : REAL;
    flow         : REAL;
    pump_running : BOOL;
END_VAR
    TT_01_HIGH := reactor_temp > 85.0;
    LOW_FLOW   := flow < 2.5;
    PUMP_STOPPED := NOT pump_running;
END_PROGRAM

```

With the matching configuration:

```

"alarms": [
  { "id": 1, "name": "TT_01_HIGH", "message": "Reactor temperature high",
    "priority": "high", "area": "M", "offset": 0, "bit": 0, "delay_ms": 500 },
  { "id": 2, "name": "LOW_FLOW", "message": "Circulation flow low",
    "priority": "low", "area": "M", "offset": 0, "bit": 2, "delay_ms": 2000,
    "suppress": { "area": "M", "offset": 0, "bit": 3 } }
]

```

`LOW_FLOW` is suppressed while `PUMP_STOPPED` (`%MX0.3`) is TRUE.

27.4.4 Bringing the definitions to the controller

Alarm definitions are part of the controller configuration and are read once, when the runtime starts.

To add or change controller alarms:

1. Write the program logic that sets the alarm bits, build it and **Download program** (F8).
2. Edit the `"alarms"` section of `controller.json` in the project directory.
3. In the IDE, press the tree's refresh button so the IDE reloads the file.
4. Open **Controller Settings** and click **Validate with the runtime**. Correct any error it reports.
5. Choose **Online** → **Download & Upload** → **Download hardware**. This sends the configuration and restarts the runtime.

WARNING: Download hardware stops the control loop while the runtime restarts. Outputs go to their configured stop state. Do it only when the machine is safe to interrupt.

6. Open **Online** → **Alarms...** and confirm the new alarms are listed.

On the simulator, **Online** → **Controller** → **Restart runtime...** also applies the project's current configuration (Chapter 26).

27.5 The alarm view (Online → Alarms...)

Online → **Alarms...** (toolbar rail button ) opens the **Controller alarms** tab. It refreshes every two seconds while it is open and visible. Reading alarms needs a login with at least the **viewer** role; acknowledging and shelving

need **operator** (§27.7).

27.5.1 Layout

Element	Content
N unacknowledged	Number of alarms not yet acknowledged. Highlighted while it is non-zero.
N active	Number of alarms whose condition is active (including acknowledged ones).
Acknowledge all	Acknowledges every unacknowledged alarm (§27.6.2). Disabled when nothing is unacknowledged.
Show journal / Hide journal	Shows the event journal under the list (§27.8).
Alarm table	One row per configured alarm.

The table columns are:

Column	Content
Priority	CRITICAL , HIGH , LOW or DIAGNOSTIC .
Name	The alarm's name .
Message	The alarm's message .
State	One of the seven states in §27.2.
Since	When the state last changed, on the controller's clock, in this PC's local time format. If the controller's clock was not synchronised at that moment, the time is followed by (<i>clock not synced</i>).
(buttons)	Ack for an unacknowledged alarm; Shelve... or Unshelve .

27.5.2 Sort order

The list is sorted **unacknowledged first**, then by **priority** (critical → diagnostic), then **newest first**. That is the order an operator in an alarm flood needs; alphabetical or numeric order would bury the alarm that matters.

27.6 Operator actions

27.6.1 Acknowledge

To acknowledge one alarm:

1. Open **Online** → **Alarms...**
2. Click **Ack** on the alarm's row.

The state changes from UNACK_ACTIVE to ACK_ACTIVE , or from UNACK_RTN to NORMAL . The acknowledgement is recorded in the journal and in the controller's audit trail with the logged-in user name, for example alarm ACK by op1: 2 TT_01_H .

27.6.2 Acknowledge all

To acknowledge every unacknowledged alarm at once:

1. Click **Acknowledge all**.
2. Confirm the dialog *Acknowledge all N unacknowledged alarm(s)?*.

Each alarm gets its own `ACK` journal entry and audit record, and the sweep itself gets one additional audit record (`alarm ACK ALL by op1: 12 alarm(s)`). "Acknowledged forty alarms at once" is a different act from acknowledging one, and a reviewer must be able to see which happened.

27.6.3 Shelve and unshelve

Shelving silences an alarm **for a limited time**. It exists because an operator faced with a chattering alarm and no way to silence it will stop reading the whole list. Shelving is time-limited and recorded, so it expires by itself and appears in the record.

To shelve an alarm:

1. Click **Shelve...** on its row.
2. Enter the number of minutes (default 15) and click **OK**.

A shelve of zero or negative duration is refused: an alarm silenced for ever is a deleted alarm. The shelve is recorded with the user name and the duration (`alarm SHELVED by op1: 3 TT_01_L for 900 s`).

While shelved the alarm is not evaluated. When the time expires, it returns to `UNACK_ACTIVE` if its condition is still active — so the operator sees it again — or to `NORMAL` . The journal records the automatic return with *who* = `expired` .

To end a shelve early, click **Unshelve** on the row.

27.6.4 Out of service

Taking an alarm out of service removes a protection, so it is an **engineer** action. An out-of-service alarm is not evaluated until it is returned to service; returning it clears its active and acknowledged flags and sets it to `NORMAL` . Both actions are journalled and audited with the user name (`alarm OUT OF SERVICE by eng1: 7 ... - it will not be evaluated until it is put back`).

The **Controller alarms** page in this IDE version has no button for out-of-service. The action is available through the monitor protocol command `alarm_oos ( "alarm": <id>, "out": true|false )`, for example from an OptiZeus SCADA maintenance screen.

27.7 Who may do what

Action	Minimum role	Recorded
View the alarm list and journal	viewer	—
Acknowledge, Acknowledge all	operator	journal + audit trail, with user name
Shelve, Unshelve	operator	journal + audit trail, with user name and duration
Out of service / in service	engineer	journal + audit trail, with user name
Change alarm definitions	engineer (Download hardware)	configuration SHA-256 in the audit trail at start

Viewing alarms is deliberately a viewer right: seeing what is wrong is never a privilege. Roles and logins are described in Chapters 6 and 34.

27.8 The alarm journal

27.8.1 What an entry records

Every state change caused by the engine or an operator writes one journal entry:

Field	Content
When	Wall-clock time of the event, on the controller's clock.
clock trusted	Whether the controller's clock was synchronised to a time source (NTP) when the entry was written. Shown as <i>(clock not synced)</i> when it was not.
Scan	The controller's scan number — its own monotonic clock. Use it to order events exactly; wall-clock times are displayed with a precision of a millisecond at best.
Alarm	The alarm name (the id is carried in the protocol).
Event	RAISE , CLEAR , ACK , SHELVE , UNSHELVE , SUPPRESS , UNSUPPRESS , OUT_OF_SERVICE , IN_SERVICE .
Priority	Priority of the alarm.
Batch	The batch, unit and phase in force when the event happened (Chapter 28), shown as unit/batch pN , or – when no batch was running.
Who	The user who acknowledged, shelved or changed service; expired for a shelve that ran out; design for suppression; empty for events raised by the engine.

27.8.2 Reading the journal

To read the journal:

1. Open **Online** → **Alarms....**
2. Click **Show journal**.

Entries appear newest first. The page reads forward from where it last stopped, so an event is never shown twice.

27.8.3 Depth, loss and the audit trail

The controller keeps the most recent **1,024** journal events in memory. If a reader falls further behind than that, the controller reports how many events it missed and the page shows a warning: *N event(s) were overwritten before this page read them*. Losing an entry is always reported, never hidden.

Record	Where	Survives restart	Contains
Alarm journal	Controller RAM, 1,024 events	No	All events: RAISE, CLEAR, ACK, SHELVE... with scan number and batch
Audit trail	Controller log (fsynced), optionally shipped to syslog	Yes	Operator actions only: ACK, ACK ALL, SHELVE, UNSHELVE, OUT/IN SERVICE

Important: RAISE and CLEAR events are held only in the in-memory journal. The controller is a **source** of alarm records, not their archive. For a permanent alarm history, have the SCADA or historian read the journal continuously (monitor command `alarm_journal1`, which returns entries from a cursor and reports any lost ones), and size its polling so it never falls 1,024 events behind.

27.8.4 Time trust

Each alarm and each journal entry carries whether the controller's clock was synchronised at the moment it was stamped. A controller without NTP or a hardware real-time clock free-runs from whatever time it booted with; its records are still evidence, but only if they say so. Set up time synchronisation before commissioning (Chapter 33).

27.9 Analogue alarms with ALARM_AN

27.9.1 The block

ALARM_AN (library file `10_signal.st`) evaluates a four-level analogue alarm — high-high, high, low, low-low — with hysteresis on the return and an on-delay on the way in. Hysteresis and delay are both needed because either alone still chatters: a value sitting exactly on a limit toggles without hysteresis, and a noisy spike trips without a delay.

Input	Type	Default	Meaning
<code>pv</code>	REAL	—	Process value, in engineering units.
<code>hh</code>	REAL	95.0	High-high limit.
<code>h</code>	REAL	90.0	High limit.
<code>l</code>	REAL	10.0	Low limit.
<code>ll</code>	REAL	5.0	Low-low limit.
<code>hyst</code>	REAL	1.0	Hysteresis, applied on the return only.
<code>delay</code>	TIME	<code>T#1s</code>	On-delay for each level.
<code>enable</code>	BOOL	TRUE	FALSE clears all outputs and resets the delays.

Output	Type	TRUE when
<code>hh_alm</code>	BOOL	<code>pv > hh</code> for <code>delay</code> ; clears when <code>pv <= hh - hyst</code> .
<code>h_alm</code>	BOOL	<code>pv > h</code> for <code>delay</code> ; clears when <code>pv <= h - hyst</code> .
<code>l_alm</code>	BOOL	<code>pv < l</code> for <code>delay</code> ; clears when <code>pv >= l + hyst</code> .
<code>ll_alm</code>	BOOL	<code>pv < ll</code> for <code>delay</code> ; clears when <code>pv >= ll + hyst</code> .
<code>any</code>	BOOL	Any of the four is TRUE.

The alarm appears after the delay once the limit is crossed, and disappears immediately once the value is back inside the limit by the hysteresis — an alarm that lingers after its condition cleared also teaches operators to ignore the list.

27.9.2 Example: ALARM_AN feeding the controller alarm engine

The block produces the conditions; the controller alarm engine turns them into ISA-18.2 alarms with priorities, acknowledgement and a journal.


```

PROGRAM ReactorAlarms
VAR
    TT_01      : REAL;          (* reactor temperature, deg C, from SCALE *)
    tt01_alm    : ALARM_AN;
    heater_on   : BOOL;

    (* The bits the controller's alarm engine watches (controller.json "alarms"). *)
    TT_01_HH AT %MX10.0 : BOOL;
    TT_01_H  AT %MX10.1 : BOOL;
    TT_01_L  AT %MX10.2 : BOOL;
    TT_01_LL AT %MX10.3 : BOOL;
    HEATER_OFF AT %MX10.7 : BOOL;  (* suppression bit for the LOW alarms *)
END_VAR

    tt01_alm(pv := TT_01,
             hh := 95.0, h := 85.0, l := 20.0, ll := 10.0,
             hyst := 1.5, delay := T#3s,
             enable := TRUE);

    TT_01_HH := tt01_alm.hh_alm;
    TT_01_H  := tt01_alm.h_alm;
    TT_01_L  := tt01_alm.l_alm;
    TT_01_LL := tt01_alm.ll_alm;
    HEATER_OFF := NOT heater_on;
END_PROGRAM

```

The matching configuration:

```

"alarms": [
  { "id": 101, "name": "TT_01_HH", "message": "Reactor temperature high-high",
    "priority": "critical", "area": "M", "offset": 10, "bit": 0 },
  { "id": 102, "name": "TT_01_H", "message": "Reactor temperature high",
    "priority": "high", "area": "M", "offset": 10, "bit": 1 },
  { "id": 103, "name": "TT_01_L", "message": "Reactor temperature low",
    "priority": "low", "area": "M", "offset": 10, "bit": 2,
    "suppress": { "area": "M", "offset": 10, "bit": 7 } },
  { "id": 104, "name": "TT_01_LL", "message": "Reactor temperature low-low",
    "priority": "high", "area": "M", "offset": 10, "bit": 3,
    "suppress": { "area": "M", "offset": 10, "bit": 7 } }
]

```

Notes on the design:

- The delay is in the block (`delay := T#3s`), so `delay_ms` is omitted in the configuration. Do not put the same delay in both places: they add up.
- The low alarms are suppressed while the heater is off, because a cold reactor that is not being heated is not abnormal.
- `ALARM_AN` needs an instance per measured value. It is a function block because each level keeps its own timer state.

Tip: With the library block, the limits are ordinary inputs. Make them retained variables (Chapter 33) or recipe parameters written by the SCADA if they must be adjustable; the alarm configuration then never needs to change when a limit is retuned.

27.10 Screen alarm lists (HMI → Alarms)

Operator screens (Chapter 32) have an **Alarm list** widget. Each row of the widget names a BOOL variable, the text the operator should read, and a severity (`alarm` , `warning` or `info`). The widget shows only the rows whose variable is currently TRUE, worst first, and shows ? for a row whose variable the controller does not report.

HMI → Alarms in the device tree gathers every such row from every screen of the project into one table:

Column	Content
Condition	The bound variable.
Severity	<code>alarm</code> , <code>warning</code> or <code>info</code> .
Now	<code>ACTIVE</code> , <code>clear</code> , or <i>not reported</i> — read live from the controller when the IDE is connected and logged in.
Shown as	The row text.
On screen	The screen that defines it.

The page warns when the same variable is worded differently on two screens: two operators reading two screens are then told different things about one condition.

Note: A screen alarm list has no acknowledgement, no journal and no timestamps. If a condition must be acknowledged or recorded, define it as a controller alarm as well, bound to the same bit.

27.11 Alarms on the simulator

The simulator runs the real alarm engine. Alarms configured in `controller.json` are raised, acknowledged, shelved and journalled exactly as on the controller, which makes the simulator the right place to test an alarm philosophy before commissioning. Its clock is the PC's clock, and time-trust reflects the PC's synchronisation, not the controller's.

27.12 Checklist for an alarm configuration

1. Every alarm bit is written unconditionally in every scan.
2. Every `id` is unique and will not be reused for a different condition.
3. Every `message` says what is wrong in words an operator uses, not the tag name.
4. Priorities are assigned by consequence, and there are few `CRITICAL` alarms.
5. Chattering conditions have an on-delay or hysteresis (in the program or `delay_ms` , not both).
6. Alarms that are meaningless in a known state have a suppression bit.
7. **Controller Settings** → **Validate with the runtime** reports the configuration valid.
8. After **Download hardware**, **Online** → **Alarms...** lists every alarm in `NORMAL` .
9. The SCADA or historian reads the journal continuously if alarm history is a record.
10. The controller clock is synchronised (Chapter 33), so entries are not marked *clock not synced*.

28. Batch control, e-signatures and validation

This chapter covers the parts of Zeus that exist for regulated batch manufacturing: the ISA-88 batch blocks of the standard library (phases, units, equipment arbitration, mode and ownership), the batch context that stamps alarm and signature records with the batch that was running, 21 CFR Part 11 electronic signatures, the validation document set that ships with the product, and library locking. Read it when you design the phase layer of a batch plant, and before a validation team starts asking for records.

WARNING: The controller is not a safety system. It carries no SIL or PL rating. Nothing in the batch layer — HOLD, STOP, ABORT, interlocks, mode — is a safety function. Emergency stop and guard interlocking belong in a certified safety relay or safety PLC wired independently of this controller.

28.1 Why there is a batch layer in the controller

A recipe says *charge 500 kg of water, then heat to 60 °C*. It does not say which valve, and it must never need to. Between the recipe (in the batch engine, for example OptiZeus) and the machine sits a **phase**: a fixed interface that any recipe can drive and that is written once per piece of equipment.

Without that fixed interface every phase is bespoke, every recipe change touches controller code, and in a regulated plant a change of setpoint becomes a revalidation.

Responsibility	Where it lives
Recipes, formulas, scheduling, batch records	The batch engine (OptiZeus or another ISA-88 engine)
Phase state machine, equipment arbitration, mode and ownership	The controller, library <code>50_s88.st</code>
The work a phase does (open this valve, run this agitator)	The controller, the phase's own program or chart
Which batch a controller record belongs to	The controller program, via <code>S88_CONTEXT</code> (§28.6)

The library deliberately contains no recipes, formulas, batch records or scheduling. Those must survive a controller being replaced, and a recipe is a document under change control, not code.

28.2 The library blocks

The batch blocks are in the library file `50_s88.st` (`@library zeus-s88 @version 1.0.0`), which is part of every new project's `lib/` folder.

Block	What it is
<code>S88_PHASE</code>	The phase state machine: the interface a batch engine drives.
<code>S88_UNIT</code>	A unit (a reactor, a vessel, a skid): which batch owns it, its mode, whether a phase is active. Uses <code>S88_RESOURCE</code> and <code>S88_MODE</code> internally.
<code>S88_RESOURCE</code>	Arbitration of shared equipment (a transfer line, a CIP skid) with a first-come, first-served queue.
<code>S88_MODE</code>	Mode (AUTO / SEMI / MANUAL) and ownership (PROGRAM / OPERATOR / EXTERNAL).
<code>S88_CONTEXT</code>	Publishes the running batch and phase to the controller's record layer.

The complete input and output lists are in Appendix C. This chapter describes how they are used together.

28.3 The phase: S88_PHASE

28.3.1 States

The state numbers and names are those of ISA-88. Learn them exactly: every batch engine uses them, and a phase that invents its own cannot be driven by one.

state	Name	Meaning	Left by
1	IDLE	Waiting to be started.	START (if <code>permissive</code> and <code>interlock_ok</code>)
2	RUNNING	Doing its work.	<code>logic_done</code> rising → COMPLETE; HOLD; PAUSE; STOP; ABORT; <code>interlock</code> lost → HOLDING; <code>fail</code> → ABORTING
3	COMPLETE	Finished normally.	RESET
4	PAUSING	Will pause at the next safe point.	<code>logic_done</code> rising, or at once when <code>auto_transient</code>
5	PAUSED	Paused between steps, logic intact.	RESUME or START → RUNNING; HOLD → HOLDING
6	HOLDING	Going to a safe holding condition.	<code>logic_done</code> rising, or at once when <code>auto_transient</code> → HELD
7	HELD	Held; the process is safe and can be resumed.	RESTART (if <code>interlock_ok</code>)
8	RESTARTING	Coming back from HELD.	<code>logic_done</code> rising, or <code>auto_transient</code> → RUNNING
9	STOPPING	Controlled stop in progress.	<code>logic_done</code> rising, or <code>auto_transient</code> → STOPPED
10	STOPPED	Stopped.	RESET
11	ABORTING	Fastest safe stop in progress.	<code>logic_done</code> rising, or <code>auto_transient</code> → ABORTED
12	ABORTED	Aborted.	RESET
13	RESETTING	Clearing back to IDLE (the owner is cleared).	<code>logic_done</code> rising, or <code>auto_transient</code> → IDLE

28.3.2 Commands

The batch engine writes a command number to the phase's `cmd` input. A command is acted on **once**: the phase program must clear `cmd` to 0 after calling the block (the template does this).

	Command	Accepted in
1	START	IDLE; also resumes from PAUSED
2	HOLD	RUNNING, PAUSED
3	RESTART	HELD
4	STOP	RUNNING, PAUSING, PAUSED, HOLDING, HELD, RESTARTING

cmd	Command	Accepted in
5	ABORT	any state except ABORTING and ABORTED
6	RESET	COMPLETE, STOPPED, ABORTED
7	PAUSE	RUNNING
8	RESUME	PAUSED

Priority of events in one scan, highest first: ABORT command; `fail`; STOP command; loss of `interlock_ok` (while RUNNING, RESTARTING, PAUSING or PAUSED → HOLDING); then the normal state machine.

WARNING: HOLD, STOP and ABORT are not interchangeable. HOLD means the batch is not over and the product is still good. STOP ends the phase in a controlled way. ABORT is the fastest safe end and usually loses the batch. A phase that treats them alike either ruins batches that could have been saved, or takes too long to stop when something is wrong.

- An interlock that drops mid-run **holds** the phase; it does not abort it. HELD is the state the batch can come back from.
- A failure (`fail` TRUE) aborts and keeps its code in `last_fail`, so the record can say what failed.
- `permissive` FALSE blocks starting but does not hold a running phase.
- Transient states (HOLDING, RESTARTING, STOPPING, ABORTING, RESETTING, PAUSING) complete when the logic reports `logic_done`, or in one scan when `auto_transient` is TRUE (the default). Each transient state is visible for at least one scan, so a batch engine sees that a phase is *going* to a safe condition rather than discovering it already there.

28.3.3 What the phase logic reads

The phase's own logic reads these outputs instead of decoding `state`, so its code says what it means:

Output	TRUE while
<code>run_logic</code>	RUNNING or RESTARTING — do the work.
<code>hold_logic</code>	HOLDING — go to the safe holding condition.
<code>stop_logic</code>	STOPPING.
<code>abort_logic</code>	ABORTING.
<code>reset_logic</code>	RESETTING.
<code>is_idle</code> , <code>is_running</code> , <code>is_held</code> , <code>is_complete</code> , <code>is_stopped</code> , <code>is_aborted</code>	The named state.
<code>is_safe</code>	IDLE, HELD, COMPLETE, STOPPED or ABORTED.
<code>t_state</code>	Time in the current state.
<code>t_running</code>	Total time spent RUNNING or RESTARTING in this run.
<code>owner</code>	The <code>owner_req</code> (batch or unit id) that started the phase; 0 = nobody.
<code>last_fail</code>	The <code>fail_code</code> that ended the last run.

Timing uses the `dt_ms` input (default 10) rather than a timer per instance. **Set `dt_ms` to the period at which the phase program actually runs** — the task period, or the program's own period if it runs slower (Chapter 18).

28.3.4 Creating a phase in the IDE

To create a phase:

1. Select the **POUs** pane and click **+** above it. The **New POU** dialog opens.
2. Enter a name, for example `Charge`.
3. In **Type**, select **Program — S88 phase (batch)**.
4. Click **Create**.

Unlike the other templates, this one is not an empty frame. The wiring of the state machine is the part that must be identical in every phase — a batch engine drives the interface, not the logic — so it is written for you. The chart underneath is the part that differs from phase to phase.

The template contains:

- the interface variables the batch engine writes: `cmd`, `batch_id`, a parameter `target` and a report value `actual`;
- the `ph : S88_PHASE` instance and its equipment conditions `interlock_ok`, `permissive`;
- the `step_done`, `failed` and `fail_code` variables the chart sets;
- a call of `ph(...)` **before** the chart, followed by `cmd := 0;`;
- an SFC chart with `Idle` and `Working` steps.

Important: Keep the call of the phase block **outside** the chart, before the first step. A POU may contain statements and then a chart; the statements run every scan whatever step the chart is in. If you move the call into a step, it stops being called the moment the chart leaves that step — exactly when the phase most needs to be told to abort.

Add the program to the task's call list (Chapter 18) before downloading.

28.3.5 Example: a charge phase

The following phase charges a vessel by weight. It tares the scale on a fresh start, keeps the tare across a HOLD/RESTART, aborts with a fail code if the fill takes more than ten minutes, and closes the valve in every state other than running.

```

PROGRAM Charge
VAR
    (* Interface the batch engine drives (OPC UA or Modbus). *)
    cmd      : INT;          (* 1 START 2 HOLD 3 RESTART 4 STOP 5 ABORT 6 RESET *)
    batch_id : DINT;
    target_kg : REAL;        (* parameter from the recipe *)
    actual_kg : REAL;        (* report back to the recipe *)

    ph       : S88_PHASE;
    interlock_ok : BOOL := TRUE;
    permissive  : BOOL := TRUE;
    step_done   : BOOL;
    failed      : BOOL;
    fail_code   : INT;

    (* Equipment *)
    inlet_valve : BOOL;
    scale_kg    : REAL;
    start_kg    : REAL;
    lid_closed  : BOOL;
    agitator_ok : BOOL;
    fill_timeout : TON;
END_VAR

    interlock_ok := lid_closed;
    permissive   := agitator_ok;

    ph(cmd := cmd, owner_req := batch_id,
        logic_done := step_done, fail := failed, fail_code := fail_code,
        interlock_ok := interlock_ok, permissive := permissive,
        dt_ms := 10);
    cmd := 0;
    charge_state := ph.state;    (* published for the unit, see globals.st *)
    step_done := FALSE;
    failed := FALSE;

INITIAL_STEP Idle:
    inlet_valve := FALSE;
END_STEP

STEP Tare:
    start_kg := scale_kg;
END_STEP

STEP Filling:
    inlet_valve := TRUE;
    actual_kg := scale_kg - start_kg;
    fill_timeout(IN := TRUE, PT := T#10m);
    IF actual_kg >= target_kg THEN
        step_done := TRUE;
    ELSIF fill_timeout.Q THEN
        failed := TRUE;
        fail_code := 101;    (* no flow *)
    END_IF;
END_STEP

STEP Safe:
    inlet_valve := FALSE;
    fill_timeout(IN := FALSE, PT := T#10m);
END_STEP

```

```

TRANSITION FROM Idle TO Tare := ph.run_logic AND ph.state <> 8;
END_TRANSITION
TRANSITION FROM Idle TO Filling := ph.state = 8;  (* RESTARTING: keep the tare *)
END_TRANSITION
TRANSITION FROM Tare TO Filling := TRUE;
END_TRANSITION
TRANSITION FROM Filling TO Safe := NOT ph.run_logic;
END_TRANSITION
TRANSITION FROM Safe TO Idle := ph.is_safe;
END_TRANSITION
END_PROGRAM

```

`charge_state` is a global variable, declared in `globals.st`, so that the unit program in §28.4.2 can read the phase state:

```

VAR_GLOBAL
    charge_state : INT;      (* S88 state of the Charge phase, 0 = none *)
END_VAR

```

How it behaves:

Situation	What happens
START with lid closed and agitator OK	IDLE → RUNNING; chart Idle → Tare → Filling.
Weight reached	<code>step_done</code> → COMPLETE; chart Filling → Safe → Idle; valve closed.
Lid opened while filling	Interlock lost → HOLDING → HELD; chart → Safe; valve closed.
RESTART after the lid is closed	HELD → RESTARTING; chart Idle → Filling without re-taring (the transition that tests <code>ph.state = 8</code> is written first and wins); RUNNING.
Fill longer than 10 min	<code>failed</code> with <code>fail_code</code> 101 → ABORTING → ABORTED; <code>ph.last_fail</code> = 101.
RESET	COMPLETE / STOPPED / ABORTED → RESETTING → IDLE.

28.4 Units, arbitration and mode

28.4.1 S88_RESOURCE — equipment arbitration

WARNING: On a multi-product plant arbitration is not optional. One transfer line and two batches with nothing arbitrating means both drive it — a cross-contamination event, and the first failure a regulator asks about.

`S88_RESOURCE` grants a shared piece of equipment to one requester at a time:

- A request is a **level**, not an edge. A requester presents its id (`requester`, non-zero) with `request` TRUE; it is queued once however long it asks.
- A second requester is **queued**, not granted. The queue holds up to 8 requesters and is served first come, first served — not by priority — so "why did batch 12 never start" is answered by the queue.

- Only the **owner** can release: `release` TRUE with `requester` equal to the current `owner`. A release presented by anybody else is ignored.
- `granted` is TRUE only for the requester that holds it (`owner` = `requester`).
- Optional watchdog: with `timeout` > 0, an owner whose `heartbeat` stays FALSE for that long is removed and `timed_out` is set — reported separately from a clean release, because the two mean different things to whoever reads the record.

28.4.2 S88_UNIT — a unit's supervision

`S88_UNIT` combines a resource (who owns the unit) and a mode block. It carries the batch the unit is making, so that every record can say which batch it belonged to without the phase having to know.

Important: To release a unit, present the **owning batch id** on `batch_id` together with `release`. The obvious wiring — `batch_id := <network word>`, `release := <that word> = 0` — releases with 0 and is refused every time. It looks as if it worked (`allocated` goes FALSE), but the unit stays owned by a batch that has ended and the next batch queues behind it for ever.

`phase_active` is TRUE when the phase state (input `phase_state`) is RUNNING, PAUSING, HOLDING, RESTARTING, STOPPING or ABORTING. Mode changes are refused while it is.

The following unit program uses the phase of §28.3.5 through the global `charge_state` and publishes the batch context (§28.6):

```
PROGRAM Unit_TK01
VAR
  (* From the batch engine *)
  req_batch   : DINT;      (* batch asking for the unit; 0 = none      *)
  acquire_req : BOOL;
  release_req : BOOL;
  mode_req    : INT;       (* 1 AUTO 2 SEMI 3 MANUAL, 0 = no request *)

  tk01       : S88_UNIT;
  ctx        : S88_CONTEXT;
  active_phase : INT;      (* 0 = none, 1 = Charge *)
END_VAR

IF charge_state <> 0 AND charge_state <> 1 THEN
  active_phase := 1;      (* Charge is doing something other than IDLE *)
ELSE
  active_phase := 0;
END_IF;

tk01(batch_id := req_batch, acquire := acquire_req, release := release_req,
    phase_state := charge_state, req_mode := mode_req, req_owner := 0,
    allow_manual := TRUE, dt_ms := 10);
mode_req := 0;

(* Stamp every alarm and signature record with the batch that owns TK01. *)
ctx(batch_id := tk01.batch, phase_id := active_phase);
END_PROGRAM
```

28.4.3 S88_MODE — mode and ownership

mode	Name	Who drives
1	AUTO	The batch engine.
2	SEMI	The operator steps it; the logic still interlocks.
3	MANUAL	The operator drives it directly.

owner	Name
1	PROGRAM
2	OPERATOR
3	EXTERNAL (SCADA / MES)

Outputs `program_may_write` (AUTO and PROGRAM) and `operator_may_write` (MANUAL, or SEMI with OPERATOR ownership) tell the equipment logic whose commands to accept.

A request is refused, with `refused` TRUE and a code, when:

refused_code	Reason
1	A phase is active on this equipment. Stop or hold it first.
2	MANUAL requested and <code>allow_manual</code> (site policy) is FALSE.
3	MANUAL requested and <code>manual_key</code> (a physical key switch, if fitted) is FALSE.
4	Unknown mode or owner number.

Mode cannot change under a running phase: equipment half-driven by a recipe and half by a person is how a batch record stops matching what happened. A rising edge on `reset` clears `refused`.

28.5 Driving phases from a batch engine

The batch engine writes `cmd`, `batch_id` and the phase parameters, and reads `ph.state`, `ph.last_fail` and the report values. Publish them over OPC UA (Chapter 29) — function-block members such as the phase's `ph.state` are published individually, under the names the OPC UA page's symbol list shows — or map them to Modbus registers (Chapter 31).

Recommendations:

- Publish `cmd`, `batch_id` and the parameters **writable**; publish `ph.state`, `ph.last_fail`, `ph.t_running` and the report values **read-only**.
- Have the batch engine write the parameters **before** START, and verify `ph.state` = 2 before it considers the phase started.
- Never let the batch engine write the equipment outputs directly. The phase owns them.

28.6 Batch context in controller records

28.6.1 What it does

Alarm journal entries (Chapter 27) and electronic signatures (§28.7) carry three extra fields: the **batch**, the **unit** and the **phase** in force when the record was written. A batch report can then answer *what alarms occurred during batch 1207, on which unit, in which phase?* from the controller's own records, instead of reconstructing it by correlating two clocks.

The batch identity comes from the **program**, not from the SCADA. The program already knows it (it is in `S88_UNIT`), and letting the SCADA push it would make the batch stamped on a GxP record depend on a network message having arrived. An empty batch in a record means no batch was running — never "the previous batch": a stale batch id reads exactly like a correct one.

28.6.2 Setting it up

1. Call one `S88_CONTEXT` instance every scan, from the same program that calls the unit block (see `Unit_TK01` above): `ctx(batch_id := tk01.batch, phase_id := active_phase);`
2. `S88_CONTEXT` writes the batch id to `%MD200` (DINT) and the phase id to `%MW204` (INT). Do not use these addresses for anything else.
3. Add a "batch_context" section to `controller.json`:

```
"batch_context": {
  "enabled": true,
  "unit": "TK01",
  "batch_id": { "area": "M", "offset": 200 },
  "phase_id": { "area": "M", "offset": 204 }
}
```

Key	Required	Meaning
<code>enabled</code>	no (default <code>true</code>)	Stamp records.
<code>unit</code>	yes	The unit name written into every record. A record that names a batch but not the unit is half a record.
<code>batch_id</code>	yes when enabled	{ "area", "offset" } of the DINT holding the running batch id; 0 = no batch.
<code>phase_id</code>	no	{ "area", "offset" } of the INT holding the active phase number; 0 = no phase. The phase name stays with the recipe.

4. **Download hardware** to apply the configuration.

The IDE shows the context in the **Batch** column of the alarm journal (**Online** → **Alarms...** → **Show journal**) and of **Online** → **Signatures...**, as `unit/batch pN`, or `–` when no batch was running.

Note: `S88_CONTEXT` writes fixed addresses, so there is one context per controller. A controller running several units stamps the batch of the unit its `S88_CONTEXT` call names.

28.7 Electronic signatures

28.7.1 What a signature is

Online → **Sign...** records that a named person took responsibility for one specific thing, at a stated time, with a stated meaning and reason — the components 21 CFR Part 11 §11.50 requires.

Important: Logging in is not signing. The controller verifies the signer's password **again**, as part of the signing command. A signature that an open session could produce is a signature anybody passing an unattended screen could produce.

What is signed is a **reference and a digest**, not a document. The controller does not hold the batch record — the batch engine does — so a signature binds a name and, optionally, a SHA-256: *"recipe R-4471 rev 3, sha256 abcd..., approved by avi"*. Anyone holding a document can check that it is the one that was signed.

28.7.2 Signing

To sign:

1. Choose **Online** → **Sign...**. The **Electronic signature** dialog opens.
2. Fill in the fields:

Field	Required	Content
What is being signed	yes	The reference, e.g. <code>recipe R-4471 rev 3</code> or <code>batch 1207 release</code> . Up to 95 characters.
SHA-256 of it (optional)	no	The document's SHA-256 in hex. Binds the signature to one exact document.
Meaning	yes	One of <code>approved</code> , <code>reviewed</code> , <code>released</code> , <code>verified</code> , <code>authored</code> , <code>rejected</code> .
Reason	yes	Why you are signing. Free text, up to 159 characters.
User	yes	Pre-filled with the logged-in user; may be another person's account.
Password	yes	The signer's password, typed now.

3. Click **Sign**.

The controller answers with the signature's sequence number (*Signature #12 recorded*). If its clock was not synchronised, the IDE message says so.

The meaning comes from a **closed list** because it is the field an auditor reads across hundreds of records; free text would turn that into archaeology.

28.7.3 When a signature is refused

Refusal	Cause
<i>A signature needs what it is about and why.</i>	Reference or reason empty (checked by the IDE and again by the controller).

Refusal	Cause
User or password not correct	Unknown user or wrong password. Counts towards the per-address login lockout (Chapter 34).
Account still has its initial password	The account was created with a documented default (for example <code>admin / admin</code>) and has never been changed. A credential anybody could guess is not evidence that a particular person took responsibility. Change the password in Users & Roles... and Download hardware .
Controller has no accounts	Signing needs named accounts on the controller (Chapter 34).

Every refusal is written to the audit trail (`SIGNATURE REFUSED from ...`).

28.7.4 What is recorded

Field	Content
Sequence number	1, 2, 3... never reused.
User and role	The signer and the role of that account.
Meaning, reference, SHA-256, reason	As entered.
Time, clock trusted	Controller time and whether it was synchronised.
Peer	The network address the signature came from.
Batch, unit, phase	The batch context in force (§28.6).

The signature is written to the controller's **audit trail** — append-only and fsynced — which is the record of legal weight. The controller also keeps the most recent 256 signatures in memory for display.

Online → **Signatures...** lists the signatures taken since the runtime last started: **#, Who, Meaning, What** (with reason and the first characters of the digest), **Batch, When**. Signatures from before the last restart are in the audit log only.

28.8 Validation documents

A regulated customer's validation team asks for a specification chain before anything else. It ships with the product, in the `docs` folder:

Document	Purpose
27 — Validation Plan (GAMP 5)	How the platform is validated, its GAMP 5 categories, and where the supplier's responsibility stops.
28 — User Requirements Specification	The requirements, each risk-classed (H/M/L) and tied to the regulation or process need behind it. Your own URS is written against it.
29 — Functional Specification	How the product satisfies each requirement.
30 — IQ / OQ protocols	Ready to execute on your installation, step by step, with the evidence to attach.
31 — Traceability Matrix	Requirement → design → test evidence, checked in both directions.

Document	Purpose
25 — QA and Memory Report	What was run, what it found and what was done about it, including defects.
32 — Batch Record Context	Design and qualification of the batch context (§28.6).
09, 10, 13	Security architecture, security requirements, evidence package.
18, 19	Time synchronisation; backup, restore and key rotation.

28.8.1 The boundary of responsibility

Layer	Contains	Validated by
Platform	Runtime, compiler, IDE, standard library, servers	The supplier (documents 27–31)
Application	Your program, IO configuration, phases, recipes, operator screens	You — your URS/FS, your PQ, your process validation

The boundary is the **download**. Everything up to "the controller executes the program it was given, deterministically, and records what happened" is the supplier's. Everything about what that program should do is yours.

The IQ and OQ protocols ship **unexecuted**, deliberately. Only you can qualify your installation on your hardware; completed records from a supplier qualify the supplier's bench, not your plant.

28.8.2 What the platform gives your validation

These controls exist in the platform so that you do not have to build and justify them in your application:

- the layout signature on retained data (Chapter 33), which refuses a restore into reshaped variables;
- the audit trail on every operator action, with user name and address (Chapter 34);
- **Verify** and **Compare...** (Chapter 20), which ask the controller what it is running;
- alarm timestamps taken in the controller, with a clock-trust flag (Chapter 27);
- signature records bound by digest (§28.7);
- program signing with an anti-rollback version (Chapter 34);
- library locking with a hash per file (§28.9).

28.9 Library versions and locking

28.9.1 Why a project locks its library

The function-block library in `lib/` is compiled into every POU of the project. A new project gets its own copy of the standard library, so retuning a block in one project affects that machine only.

In a validated project the question is: *is the library that will be downloaded the library that was validated?* A version number is a claim; an edited file that kept its number is exactly the case worth catching. The lock therefore records a SHA-256 per file, in `library.lock.json` in the project directory.

A library file declares its name and version in a header comment within its first 40 lines:

```
(* @library zeus-control @version 1.4.0 *)
```

Files without a header are still hashed, so a change to them is still detected.

28.9.2 The Library versions dialog

To open it: click the **Library** node in the device tree, then **Versions....**

The dialog shows when, by whom and why the library was last locked, and a table:

Column	Content
File	The library file.
Library	The <code>@library</code> name.
Version	The <code>@version</code> now.
Locked at	The version recorded in the lock.
State	See below.

State	Meaning
unchanged	Same content as when locked.
CHANGED	The content differs from the lock, whether or not the version number moved. The code that will be downloaded is not the code that was validated.
added	A file that was not there when the lock was taken.
REMOVED	A file that has gone. Programs using its blocks will not compile.
not locked	No lock has been taken yet.

When anything is CHANGED, added or REMOVED, the dialog warns *The library is not what this project was locked to*. Restore the library, or lock again and say why.

If the shipped default library contains files this project does not have (a project with its own `lib/` never receives blocks added to the product later), the dialog lists them with **Add to this project**. Adding them to a locked library shows them as *added* until the library is locked again.

28.9.3 Locking the library

To lock the library:

1. Open **Library** → **Versions....**
2. In **Why are you locking this?**, enter the reason, for example `validated for the 2026-09 campaign, change request CR-118`.
3. Click **Lock this library** (or **Lock again** if a lock exists).

The lock records the time, the user and the note, with the name, version, SHA-256 and size of every file. Lock when the library is the one you intend to validate against — locking is an explicit act, and a lock that updated itself whenever a file changed would record nothing anybody decided.

Tip: Put `library.lock.json` under version control with the project and include it in every archive (Chapter 5). It is the evidence that the blocks compiled into a validated program are the blocks that were reviewed.

28.10 Summary: a batch-ready project

1. One `S88_PHASE` -based program per phase, created from the **S88 phase** template, with its state published.
2. One `S88_UNIT` per unit, released with the owning batch id.
3. One `S88_CONTEXT` call per controller, and a `"batch_context"` section in `controller.json`.
4. Shared equipment arbitrated by `S88_RESOURCE`.
5. Phase interface published over OPC UA or Modbus; outputs owned by the phase.
6. Named accounts for every person; no account left on its initial password (Chapter 34).
7. Controller clock synchronised (Chapter 33).
8. Library locked, with a reason.
9. Alarms defined as controller alarms and the journal collected by the historian (Chapter 27).
10. Practices of Chapter 36 applied.

29. OPC UA server

The controller runtime contains an OPC UA server (built on open62541) that publishes the controller status, the process image and selected program variables to SCADA systems, historians and batch engines — OptiZeus SCADA among them. This chapter explains how to enable the server, choose which variables it publishes, secure it, connect a client and understand the address space it builds. Read it before connecting any OPC UA client to a controller.

29.1 Overview

Property	Value
Transport	<code>opc.tcp</code> , default port 4840
Default state	Disabled
Address space	Built once, when the runtime starts
Program variables	Published opt-in , per variable, read-only unless marked writable
Security	Signed-and-encrypted policies (Basic256Sha256 and the AES policies of the stack); optionally also the unencrypted <code>None</code> policy
Authentication	Anonymous (off by default) and/or up to 8 user name/password logins
Server certificate	Loaded from a file, or generated self-signed (2048-bit RSA, 10 years)
Function-block instance folders	Up to 1,000 ; further instances are published flat
Writes	Audited in the controller log with the value written

The server runs in its own thread and never blocks the scan. Values are read from the program when a client reads them, by name, so a symbol that survives a download keeps working (§29.8).

Note: The standalone Windows simulator in the IDE package (`bin\zeus-sim.exe`) is built without the OPC UA server. The simulator has an OPC UA server only when the IDE runs the full Linux runtime (on a development PC, under WSL2). The first line of the simulator's start report names the binary it chose and, for the Windows simulator, says that it has no OPC UA server. See Chapter 26.

29.2 Enabling the server

29.2.1 The OPC UA Server page

Open **OPC UA Server** in the device tree (under the controller node). The badge beside it shows `enabled` or `disabled`.

Field	Config key (<code>"opcua"</code>)	Default	Meaning
Enabled	<code>enabled</code>	off	Start the server with the runtime.

Field	Config key (<code>"opcua"</code>)	Default	Meaning
Port	<code>port</code>	4840	TCP port.
Endpoint URL	<code>endpoint</code>	<code>opc.tcp://0.0.0.0:4840</code>	The endpoint URL recorded in the configuration. The server listens on all interfaces at Port ; keep the port in this URL consistent with Port .
Application URI	<code>app_uri</code>	<code>urn:zeus:plc:<controller name></code> in the runtime; the page pre-fills <code>urn:zeus:plc</code>	The server's application URI. It is also written into the generated certificate's SubjectAltName; clients check that the two match.
Allow anonymous	<code>anonymous</code>	off	Accept sessions without a user identity.
Require security policy	<code>security_required</code>	on	Offer only signed-and-encrypted endpoints. When off, an unencrypted <code>None</code> endpoint is offered as well.
Certificate (DER)	<code>cert</code>	empty	Path on the controller of the server certificate.
Private key (DER)	<code>key</code>	empty	Path on the controller of its private key.
expose every symbol	<code>expose_all</code>	off	Publish every top-level program variable that is not a function-block instance (§29.4.4).
Published variables list	<code>expose</code>	empty	The variables to publish (§29.4).

Keys that the page does not show, and that you set by editing `controller.json` :

Key	Meaning
<code>users</code>	Array of <code>{ "user": "...", "password": "..." }</code> , up to 8 (§29.5.2).
<code>app_name</code>	Application name shown by clients. Default <code>OPCUAServer@<hostname></code> .
<code>cpu</code>	CPU core for the server thread; <code>-1</code> (default) leaves it to the scheduler.

Note: On a real controller, OPC UA settings reach the controller only with **Download hardware**. The page's button is therefore **Send to controller & restart** there: a plain runtime restart reloads the configuration the controller already has. Until the settings are sent, the top bar shows **SETTINGS CHANGED** (not **HARDWARE CHANGED**), and the IO devices keep their live state, because the IO is unchanged.

Note: **Apply** on the OPC UA Server page writes the fields the page shows and keeps the keys it does not show (`users` , `app_name` , `cpu`). Before version 0.1.0.9 it removed those keys and did not pre-tick **expose every symbol** from the saved configuration.

29.2.2 Procedure

To enable the OPC UA server:

1. Download the program you want to publish from (the list of variables on the page comes from the **running** program).
2. Open **OPC UA Server**.
3. Tick **Enabled**. Leave **Port** at 4840 unless another server uses it.
4. Decide the security posture (§29.5): leave **Require security policy** ticked and add users for a plant; tick **Allow anonymous** only on an isolated bench.
5. Enter a **Certificate (DER)** and **Private key (DER)** path so the server keeps the same identity across restarts, for example `/var/lib/zeus-plc/opcua-server-cert.der` and `/var/lib/zeus-plc/opcua-server-key.der` (§29.5.4).
6. Tick the variables to publish (§29.4).
7. Click **Apply**. The message reports how many variables are published.
8. Bring the configuration to the controller and restart the runtime:
 - on a controller: **Online** → **Download & Upload** → **Download hardware**;
 - on the simulator: **Restart runtime** on the page, or **Online** → **Controller** → **Restart runtime...**

WARNING: Both actions stop the control loop while the runtime restarts, drop outputs briefly and reconnect the drivers. **Restart runtime** is a full runtime restart, not an OPC UA-only restart. Do it only when the machine is safe to interrupt.

9. Reconnect the OPC UA client and browse `Objects`.

The runtime logs the server's start: `opcua: listening on opc.tcp://0.0.0.0:4840 (authenticated)` and `opcua: N of M program symbol(s) published under Objects/Symbols ...`. Read them with **Online** → **Controller** → **Controller log...**

29.3 The address space

All Zeus nodes are in namespace index **1** and have **string** node ids. The server creates three folders under `Objects`:

```
Objects
├── Controller          ns=1;s=Controller
│   ├── State          String      ns=1;s=State
│   ├── ScanCount      UInt64      ns=1;s=ScanCount
│   ├── Overruns       UInt64      ns=1;s=Overruns
│   └── JitterMaxNs    UInt64      ns=1;s=JitterMaxNs
├── ProcessImage       ns=1;s=ProcessImage
│   ├── Inputs         Byte[]      ns=1;s=Inputs      (%I, read-only)
│   ├── Outputs        Byte[]      ns=1;s=Outputs      (%Q)
│   └── Markers         Byte[]      ns=1;s=Markers      (%M)
└── Symbols            ns=1;s=Symbols
    ├── Level_SP       Float       ns=1;s=Level_SP
    ├── Pump_Run       Boolean     ns=1;s=Pump_Run
    └── LT_01_02       ns=1;s=LT_01_02 (instance folder)
        ├── EU_Value   Float       ns=1;s=LT_01_02.EU_Value
        └── Fault      Boolean     ns=1;s=LT_01_02.Fault
```

29.3.1 Controller

Node	Type	Content
State	String	Run state: <code>RUNNING</code> , <code>PAUSED</code> , <code>CONFIG</code> , <code>FAULT</code> , ...
ScanCount	UInt64	Scans executed since start.
Overruns	UInt64	Scans that overran their period.
JitterMaxNs	UInt64	Worst scan-start jitter observed, in nanoseconds.

These are always published and always read-only. Trend `Overruns` and `JitterMaxNs` in the SCADA: they are the first evidence of a controller that is overloaded (Chapter 25).

29.3.2 ProcessImage

Each area is one **byte array** covering the whole area, read from the snapshot of one scan. An area whose size is 0 is not published.

Node	Area	Writable
Inputs	<code>%I</code>	Never. An input is what the field says it is; use forcing (Chapter 22), which is audited and visible.
Outputs	<code>%Q</code>	Only when Require security policy is off, and only as a whole array of exactly the area's length.
Markers	<code>%M</code>	As <code>Outputs</code> .

A partial write of an area is refused (`BadOutOfRange`): it would leave the image half old and half new for a scan. Prefer published symbols to the byte arrays; the arrays exist for diagnostics and for clients that genuinely need the raw image.

29.3.3 Symbols

Published program variables appear under `Symbols`. A top-level variable's node id is its name. A function-block member is placed in a folder named after its instance: the folder's node id is the instance name, the member's **display name** is the short member name (`EU_Value`), and its **node id** is the qualified name (`LT_01_02.EU_Value`). Address variables by node id in a client configuration; it is unique and does not depend on the folder structure.

29.3.4 Data types

IEC type	OPC UA type
BOOL	Boolean
SINT	SByte
BYTE, USINT	Byte
INT	Int16
WORD, UINT	UInt16
DINT	Int32
DWORD, UDINT	UInt32
LINT	Int64

IEC type	OPC UA type
LWORD, ULINT	UInt64
REAL	Float
LREAL	Double
TIME	String (the runtime's text form, e.g. <code>T#1.5s</code>)

Every value carries a source timestamp set when it was read.

Publish elementary scalars. Structure members are published like function-block members (`Recipe.Target` , in a folder named after the structure). Top-level arrays, structures and STRING variables appear in the page's list but are carried internally as bytes, and a client read of such a node fails with a Bad status in this version. Array elements cannot be named in `expose` . Where a SCADA needs an array element or a text, assign it to an elementary variable in the program and publish that variable.

29.4 Publishing variables

29.4.1 Why publishing is opt-in

A project has thousands of symbols. An address space holding all of them is one no operator can browse, slow to enumerate every time a client connects, and a write path into internals that were never meant to leave the controller. An empty publish list therefore publishes no program variables — the right default for a machine that has not been commissioned. The controller status and the process image are always served.

29.4.2 The Published variables list

The list on the **OPC UA Server** page is read from the **running controller**, not from the project source: what a client can subscribe to is what the downloaded program actually has. If no program is running, the page says so; download one, or edit `opcua.expose` by hand.

Control	Effect
filter box	Shows only names containing the text. Instance groups whose members match are expanded.
checkbox at the left of a variable	Publish it (read-only).
write checkbox at the right	Publish it writable . Ticking write also publishes the variable. Disabled for a <code>CONSTANT</code> .
instance group row (▶ / ▼)	A function-block instance. Its checkbox publishes or removes all its scalar members; the row shows <i>n/m member(s) published</i> and is shown indeterminate when only some are. Click the triangle to pick members one by one.
N published	Count of published entries.

29.4.3 Function-block instances and composites

A function-block instance is a **composite** — it has no single value — so it cannot itself be published. Its **scalar members** are published individually, in the instance's folder.

Members that are themselves function blocks (for example a timer `TMR_Fault` inside a transmitter block) are shown as a sub-heading *function block* — *publish its members* and cannot be ticked. Their own members (`TMR_Fault.Q`, `TMR_Fault.ET`) are published individually where the list offers them. The same applies to a whole timer or counter declared in a program: publish `JamTimer.ET` and `JamTimer.Q`, not `JamTimer`.

If you need one value from deep inside a structure, assigning it to a plain variable in the program and publishing that variable is always possible and keeps the SCADA interface independent of the block's internals.

29.4.4 Expose every symbol

Select all publishes every variable the list shows (only those matching the filter, when one is typed), including function-block members; **Clear** unpublishes them. **Write all** makes every published variable writable, as ticking **write** on each one does, and unticking it makes them all read-only again. It is half-ticked when only some are writable. Constants stay read-only, and with **Require security policy** on, the runtime publishes every variable read-only whatever the page says; the count beside **Write all** says so.

expose every symbol (`"expose_all": true`) publishes every top-level variable of the running program that is not a function-block instance, read-only (arrays, structures and STRINGS included, with the limitation of §29.3.4). Function-block members are **not** included; list them in `expose` explicitly. Use it for a bench test or a first commissioning; for a production system publish a deliberate list.

29.4.5 The `expose` list in `controller.json`

The page writes `opcua.expose`. A bare string publishes read-only; an object with `"writable": true` publishes writable:

```
"opcua": {
  "enabled": true,
  "port": 4840,
  "endpoint": "opc.tcp://0.0.0.0:4840",
  "app_uri": "urn:zeus:plc:line1",
  "anonymous": false,
  "security_required": true,
  "cert": "/var/lib/zeus-plc/opcua-server-cert.der",
  "key": "/var/lib/zeus-plc/opcua-server-key.der",
  "users": [ { "user": "scada", "password": "change-this-now" } ],
  "expose_all": false,
  "expose": [
    "Pump_Run",
    { "name": "Level_SP", "writable": true },
    "TT_01_01.EU_Value",
    "TT_01_01.Fault",
    "LT_01_02.EU_Value",
    "LT_01_02.Fault"
  ]
}
```

These names refer to the following program:

```

FUNCTION_BLOCK Transmitter
VAR_INPUT
    raw      : INT;          (* engineering units from the IO Layer, e.g. 4000..20000 uA *)
    lo_eu    : REAL := 0.0;
    hi_eu    : REAL := 100.0;
END_VAR
VAR_OUTPUT
    EU_Value : REAL;
    Fault    : BOOL;
END_VAR
VAR
    TMR_Fault : TON;
END_VAR
    EU_Value := lo_eu + (INT_TO_REAL(raw) - 4000.0) * (hi_eu - lo_eu) / 16000.0;
    TMR_Fault(IN := (raw < 3600) OR (raw > 20800), PT := T#2s);
    Fault := TMR_Fault.Q;
END_FUNCTION_BLOCK

PROGRAM Plant
VAR_CONSTANT
    TANK_MAX : REAL := 12.5;
END_VAR
VAR
    TT_01 AT %IW0 : INT;
    LT_02 AT %IW2 : INT;
    TT_01_01 : Transmitter;
    LT_01_02 : Transmitter;
    Level_SP : REAL := 8.0;
    Pump_Run : BOOL;
    Level_Hist : ARRAY[0..9] OF REAL;
    idx : INT;
END_VAR
    TT_01_01(raw := TT_01, lo_eu := 0.0, hi_eu := 150.0);
    LT_01_02(raw := LT_02, lo_eu := 0.0, hi_eu := TANK_MAX);
    Pump_Run := LT_01_02.EU_Value < Level_SP AND NOT LT_01_02.Fault;
    Level_Hist[idx] := LT_01_02.EU_Value;
    idx := (idx + 1) MOD 10;
END_PROGRAM

```

Note: Level_Hist is deliberately not in the list: an array is not an elementary scalar (§29.3.4). To trend the history, publish LT_01_02.EU_Value and let the SCADA historian keep the history.

29.4.6 What the runtime reports about the list

At every start the runtime checks the list against the program and writes to its log:

Log line	Meaning
opcua: exposed tag 'X' is not a symbol in program '...' - renamed or removed?	A top-level name in <code>expose</code> does not exist. Usually a renamed variable; the SCADA tag would otherwise simply stop updating.
opcua: exposed member 'A.B' is not a scalar or does not exist - renamed, removed, or a composite?	A member name that does not resolve to a scalar.

Log line	Meaning
opcua: N function-block instance(s) not exposed - a composite has no single value to publish	expose or expose_all named instances; publish their members instead.
opcua: more than 1000 instances - the rest are published flat under Symbols	Instance-folder limit reached (§29.3.3).
opcua: no program variables published - list them in opcua.expose ...	The list is empty and expose_all is off.
opcua: no program is loaded, so no named variables are exposed	The runtime started without a program (§29.8).

29.5 Security and users

29.5.1 Security policies

Require security policy	Endpoints offered	Client may...
on (default)	Signed-and-encrypted policies only (Basic256Sha256 and the stack's AES policies)	Connect only with a secure channel.
off	The encrypted policies and None	Also connect unencrypted.

29.5.2 Anonymous and user logins

Setting	Effect
Allow anonymous on	Any client that can reach the port may open a session.
users configured	The listed user names and passwords are accepted. User/password login works only over an encrypted policy — the stack refuses to send a password on an unencrypted channel.
Neither	No client can log in. The runtime logs <code>opcua: no anonymous access and no users configured - no client can log in.</code>

A new project has anonymous off, security on and no `users`: the server starts but nobody can log in until you add users (or allow anonymous on a bench).

29.5.3 Writable variables and the security setting

Important: In this runtime version, variables are writable through OPC UA **only when Require security policy is off**. With it on (the default), every published variable — including those ticked **write** — and the `Outputs` and `Markers` arrays are published read-only. A `CONSTANT` is never writable.

This means that a controller that must accept writes from a batch engine or SCADA over OPC UA runs with the `None` endpoint available. If you do that:

- restrict the OPC UA port at the network level to the SCADA server (Chapter 34, host firewall);

- do not allow anonymous access on a routable network — configure users, and have the client use an encrypted endpoint for its session even though `None` is offered;
- publish writable only the variables the client genuinely has to set (setpoints, commands, recipe parameters).

The runtime logs a warning for anonymous access without security: *acceptable on an isolated bench, never in a GxP plant.*

Every successful write is audited: `opcua write symbol Level_SP := 7.5`. A refused write (a type or range the variable cannot hold) is logged with the reason and returns `BadNotWritable` to the client.

29.5.4 Server certificate

At start the server loads the certificate and key from the **Certificate (DER)** and **Private key (DER)** paths. If they are not set or cannot be read, it generates a self-signed certificate (2048-bit, valid 10 years, subject `CN=Zeus PLC OPC UA Server, O=Zeus Automation, C=IL`, SubjectAltName = the application URI and `DNS:localhost`) and saves it to those paths **if they are set**.

Paths	Result
Set, files exist	The same identity at every start.
Set, files missing	Generated once, saved, then reused at every start.
Empty	A new certificate at every start. A client that trusted the previous one must trust it again after every restart.

Use paths in a directory the runtime can write, such as `/var/lib/zeus-plc/`. The configuration directory on the controller is read-only to the runtime service. The certificate files are part of the controller backup (Chapter 33).

A certificate supplied by your site PKI is used as-is; its SubjectAltName URI must equal the **Application URI**.

29.5.5 Client certificates

The server accepts any client certificate (trust on use). The channel is still signed and encrypted; what is skipped is a managed trust list. The consequence is that authentication rests on the user name and password, not on the client certificate.

29.6 Connecting a client

To connect a generic OPC UA client:

1. Enter the endpoint `opc.tcp://<controller IP>:4840` (or the port you configured).
2. Discover the endpoints. With security required you see only `Basic256Sha256` and the AES policies, in `Sign` and `SignAndEncrypt` modes.
3. Choose `SignAndEncrypt` and a policy.
4. Trust the server certificate in the client when it asks.
5. Choose **user name** authentication and enter a user from `opcua.users` (or **anonymous** if allowed).
6. Browse `Objects` → `Symbols`.

To connect OptiZeus SCADA, add an OPC UA connection with the same endpoint, security mode and credentials, and browse `Symbols` to create tags. Use the node ids (`ns=1;s=LT_01_02.EU_Value`) as tag addresses.

Recommended subscription settings: a sampling interval no faster than the scan period of the program that writes the value (10 ms is rarely useful to a SCADA; 250–1,000 ms is typical), and publishing intervals that keep the total below a few thousand value changes per second.

Tip: Set **OPC UA** CPU affinity ("cpu" , or **Task Configuration** → **service cores**, Chapter 18) on a controller whose scan core is not isolated, so a browse storm from a client cannot disturb the scan.

29.7 Ports on a PC running the simulator

When the simulator runs the full runtime under WSL2, WSL forwards the simulator's port 4840 to Windows only if nothing on Windows already listens on it. **4840 is commonly taken by a local OPC UA Local Discovery Server**; a client connecting to `opc.tcp://127.0.0.1:4840` then reaches the discovery server, not Zeus.

To connect a client on the same PC to the simulator:

1. Either stop the local discovery service, or give the project a free port (for example 4841) on the **OPC UA Server** page and click **Apply**.
2. Restart the simulator runtime.
3. Connect to `opc.tcp://127.0.0.1:<port>`.

29.8 When the address space is built

The address space is built **once, at runtime start**, from the program loaded at that moment (the boot program) and the publish list in the configuration.

Change	What a client sees	What to do
Download that changes values or logic only	Nothing changes; published nodes keep working.	—
Download that removes or renames a published variable	Reading that node returns <code>BadNodeIdUnknown</code> . A stale number is never served.	Update <code>expose</code> and restart.
Download that adds a variable, or a new member	The new variable is not in the address space.	Apply the publish list and restart the runtime (Download hardware on a controller, Restart runtime on the simulator).
Change of publish list, users or security	Not effective until restart.	As above.
Runtime started without a program	Only <code>Controller</code> and <code>ProcessImage</code> ; <code>Symbols</code> is empty, and a later download does not fill it.	Make sure the program is the controller's boot program (every successful download stores it), then restart.

Nodes resolve their variable **by name at every access**, never by index. A variable that survives a download therefore keeps working, and one that disappeared says so instead of returning the value of whatever variable now occupies its old slot.

After a restart, reconnect the client so it re-reads the address space.

Note: On the simulator a restart may come back without the downloaded program (Chapter 26). If `Symbols` is empty, check the runtime log for the line *no program is loaded*.

29.9 Troubleshooting

Symptom	Cause	Remedy
Nothing listens on 4840	Server not enabled, configuration not downloaded, or the runtime has no OPC UA server (the Windows simulator)	Check the log for <code>opcua: listening ...</code> ; Download hardware ; see §29.1.
Client reaches a different server on the PC	Local discovery server holds 4840	§29.7.
<code>BadIdentityTokenRejected</code> / cannot log in	Anonymous off and no user, or user login attempted on the <code>None</code> endpoint	Add <code>users</code> ; use an encrypted endpoint.
Secure channel rejected	Application URI does not match the certificate's SubjectAltName (own certificate)	Make Application URI equal to the URI in the certificate, or remove the paths to let the server generate a matching one.
Server certificate changes after every restart	Certificate paths empty	Set the paths (§29.5.4).
<code>Symbols</code> folder empty	Empty publish list, or no program at start	§29.4.6, §29.8.
A new variable is missing	Address space built before it existed	Restart after Apply .
Writes rejected (<code>BadNotWritable</code> / read-only)	Security required is on, variable not ticked write , or it is a <code>CONSTANT</code>	§29.5.3.
A tag reads <code>BadNodeIdUnknown</code>	Variable renamed or removed by a download	Update <code>expose</code> , restart, fix the client tag.
<code>users</code> disappeared from <code>controller.json</code>	Apply on the OPC UA page rewrote the section	Re-add them (§29.2.1).

30. EtherNet/IP tag server

The controller can answer as an EtherNet/IP (CIP) device that serves its program variables **by name**, the way tag-based EtherNet/IP controllers do. An HMI panel or a SCADA system with a tag-based EtherNet/IP driver can then browse the Zeus program's variables and read or write them directly — including function-block members such as `TT_01_01.EU_value` — with no register map to design or maintain. This chapter explains how to enable the tag server, which types and names it serves, how to generate the controller's EDS file, and how to connect clients, including on a PC that runs the simulator. Read it before connecting any EtherNet/IP HMI or SCADA to a Zeus controller.

Note: This chapter is about the controller **serving** tags. Using the controller as an EtherNet/IP **scanner** of remote IO devices is described in Chapter 7.

30.1 Overview

Property	Value
Protocol	EtherNet/IP encapsulation over TCP, explicit messaging (CIP)
Port	44818 TCP (sessions) and UDP (ListIdentity discovery)
Default state	Disabled
Access	Read-only unless Allow writes is enabled
Connections	Unconnected messaging and Class 3 connected messaging; default 4 clients, maximum 16
Tag addressing	By ST name: <code>Level_SP</code> , <code>TT_01_01.EU_value</code> , <code>Level_Hist[3]</code>
Browsing	Tag list (Symbol object) with the CIP type of each tag
Data types on the wire	BOOL, SINT, INT, DINT, REAL
Not provided	Class 1 (cyclic implicit IO) connections, produced/consumed tags, user-defined type templates
Authentication	None — the protocol has none (§30.7)
Conformance	Functional; not ODVA-conformance tested or certified

Values are read from the running program at the moment of each request, through the same symbol reader the monitor uses, and a value that cannot be read is answered with a CIP error rather than a stale number.

30.2 Enabling the tag server

The tag server is configured on the **Controller Settings** page, section **HMI tag server (EtherNet/IP)**.

Field	Config key (<code>enip_tags</code>)	Default	Meaning
Enable EtherNet/IP tags	<code>enabled</code>	off	Start the tag server with the runtime.

Field	Config key (<code>"enip_tags"</code>)	Default	Meaning
Bind address	<code>bind</code>	empty (= <code>0.0.0.0</code> , all interfaces)	Interface address to serve on. Use the controller's address on the HMI network.
Port	<code>port</code>	44818	TCP and UDP port. Keep 44818: clients and browsers expect it.
Allow writes	<code>allow_write</code>	off	Accept Write Tag requests.
—	<code>max_clients</code>	4	Simultaneous client connections, 1...16. Not on the page; edit <code>controller.json</code> . The page keeps the existing value.

```
"enip_tags": {
  "enabled": true,
  "bind": "10.10.0.20",
  "port": 44818,
  "allow_write": false,
  "max_clients": 4
}
```

To enable the tag server:

1. Open **Controller Settings**.
2. In **HMI tag server (EtherNet/IP)**, tick **Enable EtherNet/IP tags**.
3. In **Bind address**, enter the controller's IP address on the network the HMI is on. Leave it blank only if the controller has one network.
4. Leave **Port** at 44818.
5. Leave **Allow writes** off unless the HMI must change values (§30.5).
6. Click **Apply**.
7. Choose **Online** → **Download & Upload** → **Download hardware** (on the simulator: **Online** → **Controller** → **Restart runtime...**).

WARNING: A hardware download stops the control loop while the runtime restarts. Do it only when the machine is safe to interrupt.

8. Check the controller log (**Online** → **Controller** → **Controller log...**) for the line `enip-tags: listening on <address>:44818 ...`.

The tag server serves whatever program is running. A download of a new program is served at once; no restart is needed for new variables (unlike OPC UA, Chapter 29).

Note: The standalone Windows simulator of the IDE package (`bin\zeus-sim.exe`) contains no EtherNet/IP tag server. The simulator serves tags only when the IDE runs the full Linux runtime under WSL2 (§30.8).

30.3 Tags: names, types and browsing

30.3.1 Tag names

A tag is addressed by its ST name, exactly as it appears in the monitor and in the online variable table:

ST declaration	Tag name
Level_SP : REAL; (program or global variable)	Level_SP
TT_01_01 : Transmitter; with output EU_Value	TT_01_01.EU_Value
Recipe : Recipe_T; (a STRUCT) with member Target	Recipe.Target
Level_Hist : ARRAY[0..9] OF REAL;	Level_Hist[3] (one element)
JamTimer : TON;	JamTimer.ET, JamTimer.Q (a TIME member such as ET is not served, §30.3.2)

The client encodes the name as symbolic path segments; each segment separated by a dot becomes one member step, and a numeric element segment becomes [n]. Only one array index per path is supported (one-dimensional arrays).

30.3.2 Types

Real HMI and SCADA drivers decode only the five basic tag types reliably, so every Zeus type is presented as one of them. The value travels at the width of the mapped type.

Zeus type	Served as (CIP)	Width	Note
BOOL	BOOL	1 byte	
SINT	SINT	1 byte	
USINT, BYTE	INT	2 bytes	widened, so values 128...255 stay positive
INT	INT	2 bytes	
UINT, WORD	DINT	4 bytes	widened
DINT	DINT	4 bytes	
UDINT, DWORD	DINT	4 bytes	values above 2,147,483,647 read as negative
LINT, ULINT, LWORD	DINT	4 bytes	truncated to the low 32 bits
REAL	REAL	4 bytes	
LREAL	REAL	4 bytes	rounded to single precision
TIME, STRING	—	—	not served; not listed in a browse
Function-block instance, STRUCT, whole ARRAY	—	—	not a tag; address its members or elements

Important: LINT/LWORD values are truncated and LREAL values lose precision on the wire. Do not publish a 64-bit totaliser or an LREAL counter to an HMI through this server; copy it to a DINT or REAL scaled for display.

30.3.3 Browsing the tag list

A client that browses the controller (the Symbol object's instance-attribute list) receives a **flat** list of tags, each with the CIP type it reads as:

- every top-level scalar variable of the running program (globals and program variables);
- every direct scalar member of every function-block instance and structure, as `Instance.Member`.

Not listed: TIME and STRING variables; names beginning with `_` (internal); members of members (a timer inside a function block, such as `TT_01_01.TMR_Fault.Q`); array variables and their elements. An array is not listed as one tag per element. Unlisted names can still be read and written **by name** if the client lets you type a tag address — for example `TT_01_01.TMR_Fault.Q` or `Level_Hist[3]`.

The list is sent in pages that each fit in one TCP segment, because common client libraries do not reassemble a reply split across segments. A client with a correct paging loop receives the whole list; a program with thousands of symbols produces many pages, so browse once and save the tag selection rather than browsing at every start.

30.3.4 Arrays

An array is one variable with a dimension, not a set of tags. Read or write an element by index: `Level_Hist[0]` ... `Level_Hist[9]`. A Read Tag request for several elements at once returns only the first element; read elements individually or use a Multiple Service Packet (§30.4), which most drivers do automatically when several tags are on one screen.

30.4 Services

Service	Code	Supported
Read Tag	0x4C	Yes, one element per request
Write Tag	0x4D	Yes, when Allow writes is on
Multiple Service Packet	0x0A	Yes — several reads/writes in one request
Get_Instance_Attribute_List on the Symbol object (0x6B)	0x55	Yes — tag browse
Get_Attributes_All on the Identity object	0x01	Yes
Unconnected_Send	0x52	Yes (unwrapped and routed to the services above)
Forward_Open / large Forward_Open / Forward_Close	0x54 / 0x5B / 0x4E	Yes, Class 3 (explicit) connections
Encapsulation: RegisterSession, ListIdentity, ListServices, SendRRData, SendUnitData	—	Yes; ListIdentity also on UDP
Read/Write Tag Fragmented, template (UDT) services	—	No
Class 1 (implicit IO) connections	—	No

An unsupported service is answered with the CIP status *service not supported*; a name that does not exist or cannot be read is answered with *path destination unknown*.

30.5 Writing tags

With **Allow writes** on, a Write Tag request is converted to the ST value and written through the same writer the IDE uses, so the same rules apply:

- a `CONSTANT` cannot be written;
- a value outside the variable's range is refused;
- an input bound to `%I` is overwritten by the next input read — force it instead (Chapter 22);
- the program may overwrite the value in its next scan if it assigns the variable itself. Write setpoints and commands that the program **reads**, not values it computes.

The client must send a type the server understands (BOOL, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL, LREAL); it is converted to the variable's own type.

WARNING: EtherNet/IP has no authentication. With **Allow writes** on, anyone who can reach port 44818 can change any writable variable of the program — setpoints, commands, mode requests. Enable writes only on a dedicated control network, bind the server to that network's address, and restrict the port with the host firewall (Chapter 34).

Important: In this runtime version, writes through the EtherNet/IP tag server are **not** recorded in the controller's audit trail, unlike writes from the IDE, OPC UA, the operator screens and the Modbus slave. On a validated system, keep the tag server read-only, or record HMI actions in the HMI or SCADA itself.

30.6 The EDS file

30.6.1 Why you need one

An EtherNet/IP engineering tool that browses the network finds the controller through ListIdentity and then looks for a registered EDS (Electronic Data Sheet) whose vendor, device type, product code and major revision match. Without one, the controller appears as an unknown device. The IDE generates the EDS for you; you register it once on each engineering PC.

30.6.2 Creating the EDS

To create the EDS:

1. Set up and **Apply** the tag server settings first (§30.2). The EDS records them.
2. On **Controller Settings**, under the EDS note, select **Controller model: IPC-CM400200 (CM4), IPC-CM500200 (CM5), or Both models (two files)**.
3. Click **Create EDS file....** The browser saves `Zeus_IPC-CM400200_r1.1.eds` (and/or the CM5 file).
4. Register the file with the engineering tool's EDS installation utility.
5. Browse the network again.

The IDE message pane confirms what was generated, for example *Created Zeus_IPC-CM400200_r1.1.eds (vendor 4919, device type 14, Zeus IPC-CM400200 = product code 1, revision 1.1)*. If the tag server is not enabled in the

saved settings, the message says so: the controller does not answer a browse until it is enabled and downloaded.

30.6.3 What the EDS contains

EDS entry	Value
Vendor code	4919 — a placeholder until an ODVA vendor id is assigned
Vendor name	Zeus Automation
Product type	14 (0x0E), Programmable Logic Controller
Product code	1 = IPC-CM400200 (CM4), 2 = IPC-CM500200 (CM5)
Revision	1.1 (major 1)
Product name / catalog	Zeus IPC-CM400200 / IPC-CM400200 (or CM5)
Capacity	MaxMsgConnections = <code>max_clients</code>
Connection Manager	None: no Class 1 connection is declared, because the controller has no produced/consumed assembly implementation

The file also carries, as comments, the tag-server options at generation time (enabled, port, writes, bind address, maximum connections, board) and the list of services served, so it is a complete record of the configuration.

The identity numbers are read from the runtime's own header in the IDE installation, so the EDS always matches the runtime the IDE was built with. A controller whose runtime reports a different **major** revision needs a regenerated file. The serial number reported by the controller is the board's own, so two controllers on one network are two devices to a browser.

30.7 Connecting HMI and SCADA clients

30.7.1 HMI panels

To connect an HMI panel:

1. In the HMI configuration software, add a device using the panel's **tag-based (symbolic) EtherNet/IP controller** driver — the driver intended for controllers that serve tags by name, not a generic EtherNet/IP adapter or assembly-based driver.
2. Enter the controller's IP address (the tag server's bind address) and port 44818. Where the driver asks for a slot or path, use slot 0 / backplane path to the controller itself.
3. Browse the controller's tags if the driver supports it; otherwise type tag names as in §30.3.1.
4. Bind screen objects to the tags. Choose the data type the tag is served as (§30.3.2).
5. Download the panel project and check a few values against the IDE's online variable table.

Tip: Keep the number of distinct tags on a screen and the poll rate reasonable. Each tag read is a symbol lookup on the controller, served by the tag server's thread, outside the scan. A few hundred tags at 250–500 ms is comfortable.

30.7.2 SCADA systems

OptiZeus SCADA and other SCADA systems with a tag-based EtherNet/IP driver connect the same way: controller IP, port 44818, tag names as in §30.3.1. Where the SCADA also supports OPC UA, prefer OPC UA (Chapter 29) for a validated system: it supports authentication, encryption and audited writes.

30.7.3 Discovery

The controller answers ListIdentity broadcasts on UDP 44818, so it appears in a network browse. When a **Bind address** is set, a discovery request that arrives on another interface is ignored, so the controller stays out of a browse from the other network. If UDP 44818 cannot be opened, the tag service still works — only browsing is lost — and the runtime logs `enip-tags: UDP 44818 (ListIdentity) unavailable`.

30.8 Port 44818 on a PC running the simulator

On a development PC the simulator with the tag server is the full Linux runtime running under WSL2. WSL forwards the simulator's TCP port 44818 to Windows (`127.0.0.1:44818`) only if no Windows program already listens on that port.

EtherNet/IP engineering and communication software commonly installs a Windows service that listens on TCP and UDP 44818 permanently. If such software is installed, a client on the PC that connects to `127.0.0.1:44818` reaches that software, not the simulator — it appears to connect, and then finds no Zeus tags or a different identity.

To find out whether port 44818 is taken on the PC:

1. Open a command prompt.
2. Run:

```
netstat -ano | findstr :44818
```

3. If a line shows `LISTENING` on port 44818, note the process id (last column) and identify it with `tasklist /FI "PID eq <pid>"`.

Remedies, in order of preference:

Remedy	How
Stop the other software's communication service while you test	Stop the service in Windows Services, test, restart it afterwards.
Test from another device	Point the HMI or SCADA at a real controller or a test controller rather than the PC.
Give the simulator a different port	Set Port to a free port (for example 44819), Apply , restart the simulator runtime, and configure the client with that port. Many HMI drivers cannot use a port other than 44818, and browsers expect 44818.

Other limitations on the simulator PC:

- The WSL2 forwarding makes the port reachable from the same PC. Whether other devices on the network can reach it depends on the WSL2 networking mode of that PC; if an HMI panel on the network cannot connect, test against a controller.

- UDP discovery broadcasts are not expected to reach the simulator through WSL2; connect by address instead of by browsing.

30.9 Troubleshooting

Symptom	Cause	Remedy
Client cannot connect	Tag server not enabled or not downloaded; wrong bind address; port blocked by firewall; runtime without tag server (Windows simulator)	Check the log for <code>enip-tags: listening ...</code> ; §30.2; §30.8.
Connects but finds no tags, or a different device identity	Another program on the PC holds port 44818	§30.8.
<code>enip-tags: bind 0.0.0.0:44818: Address in use</code> in the log	Another process on the controller or simulator host uses the port	Stop it or change Port .
Device shows as unknown in the engineering tool	EDS not registered, or registered for the other board or an older major revision	§30.6.
A tag reads <i>path destination unknown</i>	Name misspelt, variable renamed or removed, a TIME/STRING variable, or a whole instance/structure/array	Use the browsed name; address a member or element.
Browse shows only part of the program	Nested members and arrays are not listed	Type the name (§30.3.3).
Writes are refused	Allow writes off; variable is a <code>CONSTANT</code> ; value out of range	§30.5.
A written value springs back	The program assigns the variable every scan, or it is an input	Write a variable the program only reads.
A large counter reads wrong	64-bit value truncated to DINT	§30.3.2.
Additional clients are refused	<code>max_clients</code> reached	Raise <code>max_clients</code> (up to 16) in <code>controller.json</code> .

31. Modbus TCP slave

The controller can answer as a Modbus TCP server (slave), so that a Modbus master — an HMI panel, a SCADA system, a gateway or another PLC — can read and write ranges of the process image as coils and registers. This chapter explains how to enable the slave, how the four Modbus object spaces map onto `%I`, `%Q` and `%M`, how addresses and byte order work, and how to keep an unauthenticated protocol from becoming an open door to the outputs. Read it before connecting any Modbus master to a controller.

Note: This chapter is about the controller **serving** Modbus. Reading remote IO from Modbus TCP couplers (the controller as Modbus **master**) is described in Chapter 8.

31.1 Overview

Property	Value
Protocol	Modbus TCP (MBAP header), one unit id
Default state	Disabled
Default bind address	127.0.0.1 (this controller only)
Default port	5502
Default access	Read-only
Clients	Default 4, maximum 8 simultaneous connections
Function codes	1, 2, 3, 4, 5, 6, 15, 16
Data source for reads	The published snapshot of one scan
Writes	Into the live process image; audited with the master's address
Authentication, encryption	None — Modbus TCP has neither

31.2 Enabling the slave

31.2.1 The Modbus TCP Slave page

Open **Modbus TCP Slave** in the device tree (under the controller node). The badge shows `disabled`, `read-only` or `writable` (highlighted), and the node shows the bind address and port when enabled.

Field	Config key (<code>"modbus_slave"</code>)	Default	Meaning
Enabled	<code>enabled</code>	off	Start the server with the runtime.
Bind address	<code>bind</code>	127.0.0.1	Interface address to listen on. 127.0.0.1 = reachable only on the controller (or through an SSH tunnel). Use the controller's address on the OT network for a master on that network. 0.0.0.0 = every interface.

Field	Config key (<code>"modbus_slave"</code>)	Default	Meaning
Port	<code>port</code>	5502	TCP port. See §31.2.3 about port 502.
Unit id	<code>unit_id</code>	1	Unit identifier this slave answers to. Requests for another unit id are ignored silently, except 255 (always answered). <code>0</code> answers every unit id.
Read-only	<code>read_only</code>	on	Refuse every write function code.
Max clients	<code>max_clients</code>	4	Simultaneous TCP connections, 1...8. A connection beyond the limit is accepted and closed at once, and the refusal is logged.
Address map	<code>map</code>	—	Which areas the four object spaces cover (§31.3).
—	<code>cpu</code>	-1	CPU core for the server thread (not on the page; edit <code>controller.json</code> or use service cores in Task Configuration, Chapter 18).

31.2.2 Procedure

To enable the Modbus TCP slave:

1. Open **Modbus TCP Slave**.
2. Tick **Enabled**.
3. Enter **Bind address**: the controller's IP address on the network the master is on.
4. Leave **Port** at 5502 unless the master cannot use another port than 502 (§31.2.3).
5. Set **Unit id** to what the master sends.
6. Leave **Read-only** ticked unless the master must write (§31.5).
7. Set **Max clients**.
8. Fill in the **Address map** (§31.3). At least one object space must have a count greater than 0.
9. Click **Apply**. If writes are allowed on an address other than `127.0.0.1`, confirm the dialog *Allow writes on ...? Modbus TCP has no authentication...*
10. Choose **Online** → **Download & Upload** → **Download hardware** — the server binds at startup, so the runtime must restart.

WARNING: A hardware download stops the control loop while the runtime restarts. Do it only when the machine is safe to interrupt.

11. Check the controller log for `modbus-slave: listening on <address>:<port> unit <id>, READ-ONLY (or writes ALLOWED)`.

31.2.3 Port 502

The standard Modbus TCP port 502 is a privileged port (below 1024). The runtime drops privileges at start and does not keep the capability needed to bind it, so binding 502 normally fails with `modbus-slave: bind ...: Permission denied - port <1024 needs CAP_NET_BIND_SERVICE, or use 5502`. Use 5502 and configure the master for it. If a master can only use 502, forward 502 to 5502 in the controller's host firewall (Chapter 34) rather than giving the runtime more privileges.

31.3 The address map

31.3.1 The four object spaces

Object space	Unit	Function codes	Default area	Default access
Discrete inputs	bits	2 (read)	%I	read-only
Input registers	16-bit words	4 (read)	%I	read-only
Coils	bits	1 (read), 5 (write single), 15 (write multiple)	%Q	read/write
Holding registers	16-bit words	3 (read), 6 (write single), 16 (write multiple)	%Q	read/write

For each space the map gives:

Column	Config key	Meaning
Area	area	%I, %Q or %M ("I", "Q", "M").
Byte offset	offset	Where the range starts in the area, in bytes .
Count	count	Size of the range in objects : bits for coils and discrete inputs, 16-bit registers for the register spaces. 0 leaves the space unmapped.

A space with count 0 is **unmapped**, and the server answers *ILLEGAL DATA ADDRESS* for it rather than returning zeros that would look like working equipment reading zero.

Discrete inputs and input registers are always read-only, whatever their area. Coils and holding registers are writable unless the whole slave is **Read-only**; in `controller.json` a space can also be made read-only individually with `"read_only": true`.

```
"modbus_slave": {
  "enabled": true,
  "bind": "10.10.0.20",
  "port": 5502,
  "unit_id": 1,
  "read_only": false,
  "max_clients": 4,
  "map": {
    "discrete_inputs": { "area": "I", "offset": 0, "count": 64 },
    "input_registers": { "area": "I", "offset": 0, "count": 32 },
    "coils": { "area": "M", "offset": 100, "count": 64 },
    "holding_registers": { "area": "M", "offset": 200, "count": 100,
                          "read_only": false }
  }
}
```

31.3.2 Address translation

A Modbus address is an **index into an object space** (zero-based, as sent on the wire), not a byte offset.

Space	Modbus address n is	Example (map above)
Bit spaces (coils, discrete inputs)	bit ($offset \times 8 + n$) of the area	coil 0 = %MX100.0 , coil 10 = %MX101.2 , discrete input 13 = %IX1.5
Register spaces	the word at byte $offset + 2 \times n$	holding register 0 = %MW200 , holding register 5 = %MW210 , input register 1 = %IW2

A request that extends beyond **Count** is answered *ILLEGAL DATA ADDRESS*. A request that fits the count but not the area (the area is smaller than offset plus count) is answered *SLAVE DEVICE FAILURE*; size the map inside the process image.

Note: Many masters number registers from 1 (for example "40001" for holding register 0). Check whether your master subtracts 1. On the wire, and in this chapter, the first object is address 0.

31.3.3 Byte order

The process image holds registers in the controller's native order, which is what an ST program sees as a plain INT or WORD at %IW / %QW / %MW . The slave converts to Modbus big-endian on the wire. So %MW200 is the same 16-bit value to the ST program and to the master; no byte swapping is needed for 16-bit values.

32-bit values (DINT, REAL) occupy two consecutive registers. The first register holds the **low** 16 bits of the value as the controller stores it in memory. Configure the master's word order accordingly (often called "little-endian word order" or "word swap"), and verify it with a known value — for example write REAL 1.0 in the program and check the master reads 1.0.

31.3.4 Limits per request

Request	Maximum per request
Read coils / discrete inputs	2,000 bits
Read holding / input registers	125 registers
Write multiple coils	1,968 bits
Write multiple registers	123 registers

A count of 0 or above these limits is answered *ILLEGAL DATA VALUE*. A single-coil write must use 0xFF00 (ON) or 0x0000 (OFF).

31.3.5 Exception responses

Code	Name	Sent when
01	ILLEGAL FUNCTION	Unsupported function code; any write while the slave or the space is read-only.
02	ILLEGAL DATA ADDRESS	Unmapped space; range beyond Count .
03	ILLEGAL DATA VALUE	Count 0 or too large; malformed request; bad single-coil value.
04	SLAVE DEVICE FAILURE	Mapped range lies outside the process image area; image could not be read or written.

A frame with a protocol id other than 0 or an impossible length is a desynchronised stream; the connection is closed and the event logged (modbus-slave: bad MBAP from ...).

31.4 Reads: consistency and timing

Reads are served from the **published snapshot** of one scan, so every value in one response belongs to the same scan — a master never sees half of scan *N* and half of scan *N*+1 in one request. The snapshot is published at the end of each scan, so one scan period is the floor of the read latency.

31.5 Writes

31.5.1 Where writes land

A master's write goes into the **live** process image and the program sees it on its next scan. Every write is recorded in the controller's audit trail with the master's address, for example `modbus-slave write reg 5 := 1200 from 10.10.0.50:51234` or `modbus-slave write 8 coil(s) from 0 from 10.10.0.50:51234` .

31.5.2 Choose the area for written data carefully

Map writable spaces to	Result
<code>%M</code> (recommended)	The master writes a marker; the program reads it and decides what to do. Nothing else overwrites it.
<code>%Q</code>	The master writes the output image directly. If the program also assigns that output, the program overwrites the master's value in its next scan, and the master sees its write "not take". If the program does not assign it, the master drives the physical output with no interlock in between.
<code>%I</code>	Not useful: the next input read overwrites the value. Map inputs only to the read-only spaces.

Map holding registers and coils to a reserved `%M` block, and let the program copy commands and setpoints from there to where they are used, with the interlocks and limits it applies to any other command. Declare the variables at those addresses so the program has names for them:


```

PROGRAM ModbusInterface
VAR
    (* Holding registers 0..3 of the map in 31.3.1 (area M, byte offset 200) *)
    mb_speed_sp  AT %MW200 : INT;      (* HR 0, 0..1500 rpm *)
    mb_level_sp   AT %MW202 : INT;      (* HR 1, 0..1000 = 0..100.0 % *)
    mb_command    AT %MW204 : WORD;     (* HR 2, bit 0 = start, bit 1 = stop *)
    (* Coils 0..7 (area M, byte offset 100) *)
    mb_reset      AT %MX100.0 : BOOL;   (* coil 0 *)

    speed_sp      : INT;
    level_sp      : REAL;
    start_req     : BOOL;
    stop_req      : BOOL;
END_VAR

(* Accept only plausible values from the network. *)
IF mb_speed_sp >= 0 AND mb_speed_sp <= 1500 THEN
    speed_sp := mb_speed_sp;
END_IF;
level_sp := INT_TO_REAL(LIMIT(0, mb_level_sp, 1000)) / 10.0;
start_req := (mb_command AND 16#0001) <> 0;
stop_req  := (mb_command AND 16#0002) <> 0;
IF mb_reset THEN
    mb_reset := FALSE;          (* acknowledge a one-shot request *)
END_IF;
END_PROGRAM

```

31.5.3 Security

WARNING: Modbus TCP has no authentication and no encryption — the protocol offers neither. With writes allowed, anyone who can reach the port can change the mapped values and, through them, drive the machine. Keep **Read-only** on unless a master genuinely has to command this controller; bind to the OT network address, never 0.0.0.0 on a controller connected to a plant LAN; and restrict the port in the host firewall (Chapter 34).

The runtime logs a warning at every start when writes are allowed on a routable address:

```

modbus-slave: bound to 10.10.0.20 with writes allowed - Modbus TCP has no authentication,
so anyone who can reach this port can drive the outputs. Use "read_only": true or bind to the OT subnet.

```

The **Network Interfaces** page of each port lists the services listening on it, and highlights a Modbus slave that allows writes or is bound to 0.0.0.0.

31.6 Unit id and gateways

A Modbus gateway forwards by unit id. The slave answers requests addressed to its **Unit id** and to 255, and ignores other unit ids without answering — the specified behaviour for a unit that is not addressed. Set **Unit id** to 0 to answer every unit id.

31.7 Performance and scan impact

The slave runs in one thread, multiplexed over the listener and up to eight connections, and takes no lock the scan thread holds. Heavy polling therefore does not stretch the scan, but it does use CPU; on a controller whose scan core is not isolated, pin the Modbus thread to another core (`"cpu"` , or service cores in Task Configuration). Poll at a rate the master needs — every 250–1,000 ms for an HMI — and read contiguous blocks rather than single registers.

31.8 Availability on the simulator

The standalone Windows simulator of the IDE package (`bin\zeus-sim.exe`) has no Modbus slave. The simulator provides it only when the IDE runs the full Linux runtime under WSL2 (Chapter 26). Test a master against a controller or against that runtime.

31.9 Troubleshooting

Symptom	Cause	Remedy
Connection refused	Slave not enabled or not downloaded; bound to <code>127.0.0.1</code> ; wrong port; firewall	Check the log for <code>modbus-slave: listening ...</code> ; set Bind address ; Download hardware .
Permission denied binding port 502	Privileged port	Use 5502 (§31.2.3).
Runtime refuses to start: <i>"modbus_slave" is enabled but maps nothing</i>	All counts are 0	Map at least one space.
Master gets no answer at all	Unit id mismatch (silently ignored)	Set Unit id , or 0 for any.
<i>ILLEGAL DATA ADDRESS</i>	Space unmapped, or request beyond Count ; master counts from 1	§31.3.2.
<i>ILLEGAL FUNCTION</i> on a write	Slave or space read-only	§31.5.
<i>SLAVE DEVICE FAILURE</i>	Map extends beyond the process image area	Reduce offset or count, or enlarge the area.
Written value springs back	The program assigns the same address	Map writes to <code>%M</code> (§31.5.2).
32-bit values read wrong	Word order	§31.3.3.
Extra masters disconnected immediately	Max clients reached	Raise it (up to 8).

32. Operator screens (HMI)

A Zeus project can contain operator screens: simple, declarative screens that show live values, lamps, bars, trends, plant symbols and alarm lists, and let an operator press buttons and enter values. The screens are drawn in the IDE and served by the **controller itself**, so a panel with a browser keeps working with the engineering workstation switched off. This chapter explains how to add screens to a project, use the screen editor, bind widgets to variables, download the screens and configure the controller's screens server and its users. Read it when a machine needs a local operator panel and a full SCADA system is not justified.

32.1 Concepts

Term	Meaning
Screen	One page, stored as <code>hmi/<name>.json</code> in the project. A fixed-size layout of widgets.
Widget	A rectangle with a kind (label, value, lamp, button...), a position, a size and — for most kinds — a binding to a program variable.
Binding	The name of a variable as the controller reports it: <code>Level_SP</code> , <code>TT_01_01.EU_Value</code> , <code>Filling.X</code> .
Screens server	The small web server inside the runtime that serves the screens to panels.
Download screens	Copies the screens, the operator page and imported drawings to the controller.

A screen contains **no scripting**, no expression language and no event handlers, and there will not be any. An operator screen that can run arbitrary code cannot be reviewed; on a machine that moves, *what can this button do* must be answerable by reading the screen file. Logic belongs in the program: if a screen needs a derived value, compute it in the program and bind the screen to it.

Screens are plain JSON files in the project, so they can be diffed, reviewed, kept under version control and archived like every other part of the project.

32.2 Adding operator screens to a project

A new project has no HMI section. To add one:

1. In the device tree, under **Application**, open **Add operator screens (HMI)...**
2. The IDE adds the **HMI** node and asks for the first screen's name (§32.3.1).

The **HMI** node contains:

Node	Opens
Screens	The list of screens; opening it with no screen selected starts a new one. Each screen is a child node showing its widget count (marked template for a faceplate template, §32.4.9), or broken if its file cannot be read.
Alarms	Every alarm-list row defined on any screen, with live state (Chapter 27, §27.10).
Events	The controller's own log, newest first, with an audit only filter (Chapter 34).
Server	The Screens Server page (§32.6).

Right-click **HMI** for **New screen...**, **Generate faceplate...** (§32.4.9), **Download screens to controller** and **Remove HMI from this project**. Removing the HMI section hides it but does **not** delete the files in `hmi/` ; adding it again brings them back. Screens already downloaded to a controller are still served until you replace them there.

Note: Adding screens changes nothing on the machine. The controller serves them only after the screens server is enabled (§32.6) and the screens are downloaded (§32.7).

32.3 The screen editor

32.3.1 Creating a screen

To create a screen:

1. Right-click **HMI** and choose **New screen...** (or open **Screens**).
2. Enter a name — letters, digits, dot, dash and underscore, up to 64 characters, for example `main` or `filler_overview`.
3. Leave **Kind** at **Screen**. (**Template (faceplate)** creates a template instead — §32.4.9.)
4. Click **Create**. The screen opens in its own tab, `<name> · Screen`, at 1024 × 600 pixels with a poll interval of 300 ms.

The screen named `main` is the one a panel opens by default.

32.3.2 Editor layout

Area	Content
Toolbar, Add:	One button per widget kind (§32.4). The new widget is placed on the canvas and selected.
Delete, Duplicate	Act on the selected widget.
Symbol library...	Browse the built-in library of about 1,080 plant drawings and insert one as a plant symbol (§32.4.4, <i>Symbol library</i>).
Import SVG...	Import a drawing to use as a plant symbol (§32.4.4).
Images...	Import, list and remove the pictures used by Image widgets (§32.4.6).
Generate faceplate...	Create a template from a function-block type (§32.4.9).
Save screen	Writes <code>hmi/<name>.json</code> .
Open as operator...	Opens the screen in a new browser window as an operator sees it, live, through the IDE (§32.8).
Colour boxes	Every colour field (Fill, Line colour, Text colour, Colour, Colour when ON, colour rules, meter bands, alarm colours) has a swatch button in front of it: click it for a palette of 40 colours, the panel theme colours (they follow a light or dark panel), the colours used recently, Custom... for any colour, and Clear for <i>as drawn</i> . Typing <code>#rrggbb</code> , a colour name or <code>var(--ok)</code> still works.

Area	Content
Variable lists	Every Variable box drops down the variables you can bind. Connected to a controller, they are the controller's (what is downloaded and running); offline, they are every variable the project declares — globals, <code>Program.variable</code> , function-block members (<code>Motor1.Q</code>) and structure fields. Type part of a name to narrow the list, — clicking a Variable box (or typing in it) opens the variable browser under it, already filtered by what you typed; the ... button beside the box opens it too: a searchable tree with Global variables , one folder per Program , and the members of each function block and structure, each with its type. Click a variable to put it in the box, or use ↑ ↓ and Enter. Esc closes the browser and lets you type a name yourself.
Canvas	The screen at its authored size. Drag to move; arrow keys nudge (Shift+arrow: one pixel); Delete removes. Positions snap to an 8-pixel grid. Right-click a widget for Cut , Copy , Paste , Duplicate , Delete , Undo/Redo , the drawing order (Bring to front , Bring forward , Send backward , Send to back) alignment to the screen (centre, left, right, top, bottom edge), Save screen and Download screens to controller ; right-click empty canvas for Paste here . Download screens is also on the editor toolbar: it saves the screen first, then sends every saved screen, and warns when the controller's screen server is off or listens on 127.0.0.1 only. Keys: Ctrl+C, Ctrl+X, Ctrl+V, Ctrl+D, Ctrl+Z, Ctrl+Y (click the canvas first). The clipboard is shared by all screen tabs, so a widget can be copied from one screen to another. Undo is per screen tab and also works from the ↶ ↷ buttons in the top bar.
Properties panel	With nothing selected: the screen's Title , Width , Height , Poll ms and Background . With a widget selected: its fields (§32.4), X , Y , Width , Height , its Style (§32.4.7) and its Dynamics (§32.4.8).

The screen is authored at a fixed size and scaled to fit whatever panel opens it. It is a layout: reflowing it would move things relative to each other.

Poll ms is how often the screen reads its values from the controller. 300 ms is a good default; below 200 ms rarely helps an operator and loads the controller's web server.

32.4 Widgets

32.4.1 Widget kinds

Kind (toolbar)	Binding	Fields	What it does
Label	—	Text, Font px, Colour	Static text: a caption, a unit, a heading.
Value	Variable	Decimals, Unit, Font px	A live number or text.
Lamp	Variable (BOOL)	Text, Colour when ON, Font px	An indicator that changes colour with a BOOL.
Button	Variable (BOOL)	Text, On press or Actions, Confirm text, Font px	Writes a BOOL (§32.4.2), or runs a list of actions (§32.4.12).
Input field	Variable	Decimals, Writes, Keypad, Limits, Confirm text, Font px	A field the operator types a value into, or enters with the on-screen keypad (§32.4.11).
Bar	Variable (number)	Minimum, Maximum, Decimals, Unit, Colour	A bar filled between minimum and maximum.
Box	—	Text, Font px	A plain rectangle for grouping and framing.

Kind (toolbar)	Binding	Fields	What it does
Trend	Variable (number)	Minimum, Maximum, Window (s), Colour	The value against time for the last <i>Window</i> seconds (5...3600).
Alarm list	per row	Text, rows, Font px	Shows the rows whose condition is TRUE (§32.4.3).
Screen link	—	Opens screen, Open as, Text, Font px	A button that opens another screen, in place or as a dialog over this one (§32.4.9).
Plant symbol	Variable, optional Fault bit	Shape, Minimum, Maximum, Text, Colour, Font px	A tank, valve, pump, motor... (§32.4.4).
Rectangle	— (rules)	Text, Font px	A rectangle, optionally rounded (§32.4.6).
Ellipse	— (rules)	Text, Font px	An ellipse or circle.
Line	— (rules)	Points	A straight line: a pipe run, a connection.
Polygon	— (rules)	Points, Text, Font px	A closed outline through three or more points.
Image	— (rules)	Picture	A picture from the project's image pool (§32.4.6).
Faceplate	per argument	Template, Fit, Arguments	A template drawn in the box with its parameters filled in (§32.4.9).
Alarm table	— (controller alarms)	Text, Show from, Alarms, Rows, Format, Font px, Columns, Colour per priority	The controller's alarms with Acknowledge , Acknowledge all shown and History (§32.4.14).
Slider, Knob, Meter, 7-segment, Checkbox, Radio buttons, Drop-down, Array table, Tabs, Date/time	see §32.4.10		Input and display elements (§32.4.10).

The **Variable** field offers the names the running controller reports (a list read from the connected controller; function-block instances themselves are not offered, only their members). Type a name if the controller is not connected; it is checked when the screen runs.

32.4.2 Buttons and input fields

On press for a button:

Action	Effect
<code>momentary</code>	TRUE while held, FALSE on release.
<code>set</code>	Writes TRUE.
<code>reset</code>	Writes FALSE.
<code>toggle</code>	Writes the inverse of the current value.

A momentary button is released on pointer up, on cancel, when focus leaves the page and when the page is hidden — a jog whose release was missed is a machine that keeps moving. Even so, do not use a screen button as the only way to stop motion; the program must time out a jog command it no longer sees refreshed if that matters to the machine.

Instead of **On press**, a button can run a list of **actions** — for example reset one bit and set another (§32.4.12). An input field can check limits and open an on-screen keypad (§32.4.10, §32.4.11).

Confirm text, when not empty, is shown in a confirmation dialog before the write is sent. Use it for anything that starts, stops or changes a mode.

Buttons and input fields write to the running controller. Every write is recorded in the controller's audit trail against the logged-in user, for example `web: op1 wrote setpoint := 4242 from 10.0.0.9`.

WARNING: A screen button is not an emergency stop and must never be wired or programmed as one. Emergency stop belongs in a hard-wired safety circuit, independent of this controller.

32.4.3 Alarm list

Each row of an **Alarm list** has:

Column	Content
When this is TRUE	A BOOL variable.
Show this	The text the operator reads: what is wrong and what to do, not the tag name.
severity	alarm, warning or info.

Only **active** rows appear on the screen, worst first. A row whose variable the controller does not report shows **?** rather than vanishing: not knowing whether an alarm is active is itself something the operator has to be told. The list has no acknowledgement and no history; for those, use controller alarms (Chapter 27) and an **Alarm table** (§32.4.14).

32.4.4 Plant symbols

A **Plant symbol** draws a piece of plant. The built-in shapes are:

Group	Shapes
Vessels and gauges (show a level)	tank, hopper, gauge
Rotating equipment (show a state)	pump, motor, fan, agitator, conveyor
Valves and process items (show a state)	valve, three-way valve, heater, filter
Piping	horizontal pipe, vertical pipe
Panel devices	toggle switch, pilot light, emergency-style button

A **level** shape is bound to a number and fills between **Minimum** and **Maximum**. A **state** shape is bound to a BOOL; add a **Fault bit** (a BOOL that means broken) and the symbol turns red when it is TRUE — a symbol with one binding cannot show a machine that is broken. Anything the controller does not report is drawn grey, never in the healthy colour.

Imported drawings. Import SVG... stores a drawing in `hmi/symbols/` and offers it under **Imported** in the **Shape** list. The file is cleaned before it is stored: scripts, event handlers, embedded HTML and references to other hosts are removed, and the dialog shows what was removed. To make part of a drawing follow the machine, give that element `class="zeus-state"` in your drawing program; it is then coloured by the state of the bound variable, and the rest stays as drawn.

Symbol library

The IDE carries a library of about 1,080 ready-made plant drawings in 39 categories — tanks and vessels, pumps, valves, instruments, piping and 3D pipes, blowers, HVAC ducts, heat exchangers, motors, electrical, callouts, arrows and more — taken from the OptiZeus SCADA symbol set.

To insert one:

- 1. In the screen editor click **Symbol library...** (or choose **Symbol library...** at the bottom of a plant symbol's **Shape** list, to change the drawing of that symbol).
- 2. Pick a category on the left (each shows how many drawings it has), or type in **Search all symbols...** — the search looks in every category, in titles and tags (`pump` , `3d valve` , `flow`).
- 3. **Click** a thumbnail to insert it at the centre of the screen; **double-click** to insert it and close the dialog. The symbol is about 120 pixels on its long side, in the drawing's own proportions, and is selected so it can be dragged into place.

Inserting copies the drawing into the project as `hmi/symbols/<name>.svg`, through the same cleaning as **Import SVG...**. From then on it is an imported drawing like any other: it is listed under **Imported** in the **Shape** list and in **Import SVG...**, it is saved and versioned with the project, and it goes to the controller with **Download screens** (§32.7). Inserting the same drawing again reuses the file. If the project already has a drawing of that name with different content (your own import), it is left alone and the library copy is stored as `<name>_2`.

Colour with state. An imported drawing has a **Colour with state** choice in its properties:

Choice	Effect
none	The drawing is shown exactly as drawn, whatever the bound variable says.
whole symbol	The whole drawing is tinted with the state colour — green running, red fault (Fault bit TRUE), grey off or unknown — and keeps its shading.
elements with class zeus-state	Only the parts marked <code>class="zeus-state"</code> follow the state; the rest stays as drawn. This is what an imported drawing without the setting does.

Library drawings mark no parts, so they are inserted with **none**; choose **whole symbol** and bind the variable (and a **Fault bit**) for a pump or valve that shows its state. A drawing you import with `zeus-state` parts starts with **elements with class zeus-state**. Dynamic properties (§32.4.8) — visibility, blinking, position, size, rotation — work on library symbols as on any widget. For an impeller, fan, mixer or turbine, right-click it and choose **Add spin while running...** (or **Add impeller pulse while running...**): the rule is added with the symbol's own **Variable** already filled in, and turns the drawing only while that variable is TRUE. A drawing whose variable is unknown stands still.

The thumbnails in the dialog are pictures (`<img>`): nothing in a library file can run in the IDE. Every library drawing has passed the SVG cleaning with nothing to remove, carries no reference to any other host, and has element ids unique across the library, so two different drawings on one screen never borrow each other's gradients.

32.4.5 Trends

A trend starts empty when the screen opens and is gone when it closes. It is not a historian: it has no storage and no clock discipline, and does not pretend to have a history. Use a SCADA historian for records. Tick **Recorded** to have the controller keep the last minutes or hours in memory, so the trend shows them when the screen opens (§32.4.17).

32.4.6 Shapes and pictures

Rectangle, **Ellipse**, **Line** and **Polygon** draw plain shapes. They have no binding of their own; they come alive through dynamic properties (§32.4.8) — a rectangle whose height follows a tank level, an ellipse whose colour follows a pump.

Points of a line or polygon are pixel positions inside the widget's box, written `x,y x,y ...` — for example `60,0 120,100 0,100` for a triangle in a 120 × 100 box. A line takes exactly two points; without them it is drawn across the middle of its box. A polygon takes 3 to 256. Resizing the box by dragging its corner scales the points with it.

An **Image** shows a picture from the project's **image pool**, `hmi/images/`. To add pictures:

1. In the screen editor, click **Images...**
2. Choose a `.png`, `.jpg` or `.svg` file and, if needed, change the **Name**.
3. Click **Import**. The dialog lists every picture in the project, with **Remove**.

Type	Limit	On import
PNG, JPEG	1 MB	Checked to really be a PNG or JPEG (by its content, not its name).
SVG	512 kB	Cleaned exactly like an imported symbol (§32.4.4); what was removed is shown.

Then select the Image widget and choose the file under **Picture**. A widget refers to a picture by its file name, for example `line2_layout.png`; a picture cannot be taken from a web address. Removing a picture from the pool does not change the screens that use it — they show an empty frame until the picture is imported again or another is chosen.

Note: Scale a photograph to the size it is shown at before importing it. A panel loads every picture each time the screen opens.

32.4.7 Style

Every widget has a **Style** section in the properties panel. A field left empty (or at its default) means *as the widget draws itself*, so screens made before these fields existed look exactly as they did.

Field	Values	Applies to
Fill	colour	The inside of a shape; the background of any other widget. On a lamp, a fill replaces the lamp's own on/off colouring.
Line colour	colour	The outline of a shape; the border of any other widget.
Line width	0...100 px	Outline or border width. Default 1.
Corner radius	0...1000 px	Rounded corners (not on ellipses, lines and polygons).
Rotation (°)	−360...360	Rotates the widget about its centre.

Field	Values	Applies to
Opacity 0–1	0...1	1 is solid, 0 is invisible.
Text colour	colour	The text of the widget.
Text align	as drawn, left, center, right	Where the text sits.

A **colour** is written `#rgb`, `#rrggbb` (or with alpha, `#rrggbbaa`), `rgb(...)` / `rgba(...)` / `hsl(...)`, a common name such as `red`, `orange`, `grey`, or one of the panel's own colours `var(--ok)`, `var(--warn)`, `var(--err)`, `var(--accent)`, `var(--dim)`. Anything else is refused when the screen is saved, with a message naming the widget and the field.

32.4.8 Dynamic properties

A **dynamic property** makes one property of a widget follow one variable: a colour, a size, a position, the text, whether the widget is shown, whether it blinks. Each widget can have one rule per property. The rules are declared, not programmed — a comparison is an operator and a value, a scaling is two pairs of numbers — and they are checked when the screen is saved.

To add a rule:

1. Select the widget.
2. Under **Dynamics**, click **Add rule**.
3. Choose the **Property**. Only the properties that apply to this kind of widget are offered.
4. Choose the **Rule**. Only the rule types that apply to this property are offered.
5. Enter the **Variable**, as for any binding (§32.4.1), and fill in the rule's fields.
6. Save the screen.

Remove deletes a rule. Changing a rule's property or type starts that rule afresh and keeps its variable.

Property	Rule types	Notes
X position, Y position, Width, Height	Scale	Pixels on the screen.
Rotation, Opacity, Line width	Scale	Degrees; 0–1; pixels.
Fill colour, Line colour, Text colour	Bool, Map	
Visible	Cond	Hidden while the condition is FALSE.
Blink (1 Hz)	Cond	Blinks while the condition is TRUE.
Spin while true	Cond	Plant symbol, image, rectangle, ellipse, polygon only. The drawing turns continuously while the condition is TRUE — an impeller, fan, mixer or turbine running. Speed slow (4 s per turn), normal (2 s), fast (1 s), very fast (0.5 s); Direction clockwise or anticlockwise. The caption does not turn.

Property	Rule types	Notes
Impeller pulse while true	Cond	Same widgets. The drawing narrows to a line and widens again while the condition is TRUE — the look of a mixer blade turning seen from the side. Speed slow (2 s), normal (1.2 s), fast (0.8 s), very fast (0.5 s). If a widget has both, Spin wins.
Text	Format, Map	Labels, values, lamps, buttons, boxes, screen links, symbol captions, and text inside shapes.
Enabled (writes)	Cond	Buttons and input fields only; while FALSE they cannot be pressed or typed into.

Rule	Fields	Result
Scale	<i>From value</i> low, high; <i>To</i> low, high	Linear, and clamped to the <i>To</i> range: a level of 130 % draws a full tank, not one that runs off the screen. <i>To</i> may run backwards (high to low).
Bool	<i>When TRUE</i> , <i>When FALSE</i>	One of two colours.
Map	one or more <i>If value</i> rows (operator, value, result), <i>Otherwise</i>	The result of the first row whose comparison holds; else <i>Otherwise</i> . An empty <i>Otherwise</i> leaves the widget as drawn.
Cond	operator, value	TRUE or FALSE.
Format	format text	The value as text (below).

Operators are `==`, `!=`, `<`, `<=`, `>`, `>=`. A comparison value is a number, `TRUE` / `FALSE`, or a short text for STRING variables.

A **format** is literal text with one or more of these codes, each replaced by the variable's value:

Code	Shows	Example (value 23.456)
<code>%d</code> , <code>%i</code>	Whole number (truncated)	23
<code>%u</code>	Whole number, unsigned	23
<code>%x</code> , <code>%X</code>	Hexadecimal	17
<code>%f</code> , <code>%.Nf</code>	Fixed point, 6 or <i>N</i> decimals	<code>%.1f</code> → 23.5
<code>%e</code> , <code>%.Ne</code>	Exponent	<code>%.2e</code> → 2.35e+01
<code>%s</code>	The value as the controller reports it	23.456
<code>%%</code>	A percent sign	

Widths and flags (`%5.1f`, `%-d`) are not supported and are refused on save, rather than shown literally on the panel.

Unknown means grey. When the controller does not report a rule's variable — misspelt, renamed, not in the downloaded program, or the connection is lost — the widget is drawn grey: colours become grey, text shows **?**, a position or size stays as drawn, a writable widget is disabled, and a blinking, spinning or pulsing widget stops. A *Visible* rule does not hide a widget whose variable is unknown. The screen never shows the healthy colour for a value it does not have.

The screen asks the controller for every variable a rule uses, together with the widgets' own bindings.

Worked example — a tank, a pump, a temperature and an alarm colour. The program has `Tank1.Level` (REAL, 0–100 %), `P101.Running` (BOOL) and `TT101` (REAL, °C).

1. **Tank level bar.** Add a **Rectangle** for the tank outline at X 40, Y 60, 80 × 240, **Fill** empty, **Line colour** #8b949e . Add a second **Rectangle** for the liquid, same X and width, **Fill** #1e6fd9 , **Line width** 0. On the second rectangle add two rules on `Tank1.Level` :
 - **Height, Scale**, from 0 ... 100 to 0 ... 240;
 - **Y position, Scale**, from 0 ... 100 to 300 ... 60.
 The liquid grows upwards from the bottom of the outline (Y 300) as the level rises.
2. **Pump colour.** Add an **Ellipse** (60 × 60) with the rule **Fill colour, Bool** on `P101.Running` : *When TRUE* #22c55e , *When FALSE* #6b7280 .
3. **Temperature text.** Add a **Label** with the rule **Text, Format** on `TT101` : `%.1f °C` . It shows 23.5 °C .
4. **Alarm colour.** On the same label add **Text colour, Map** on `TT101` : `>= 90` → #ef4444 , `>= 70` → #f59e0b , *Otherwise* #22c55e . The rows are tried in order, so put the highest limit first.

In the screen file (`hmi/<name>.json`) the label reads:

```
{ "kind": "label", "text": "TT101", "x": 300, "y": 60, "w": 120, "h": 30,
  "dyn": {
    "text": { "bind": "TT101", "type": "format", "format": "%.1f °C" },
    "textColor": { "bind": "TT101", "type": "map", "ranges": [
      { "op": ">=", "value": 90, "result": "#ef4444" },
      { "op": ">=", "value": 70, "result": "#f59e0b" } ], "else": "#22c55e" } } }
```

Note: Rules change what the operator sees, never what the machine does. A rule cannot write a variable. Anything that must act on the machine belongs in the program.

32.4.9 Faceplates (templates)

A plant has many motors, valves and loops of the same kind. A **template** (faceplate) is drawn once for a kind of equipment, with its variables written against a **parameter** instead of a real instance — `$M.running` instead of `P101.running` . It is then placed as often as needed with a **Faceplate** widget, each time with its own argument (M = P101, M = P102), or opened as a **dialog** from a screen link. Change the template and every place it is used changes with it.

Term	Meaning
Template	A screen file (<code>hmi/<name>.json</code>) marked <code>"type": "template"</code> , with a list of parameters. It is drawn, saved and downloaded like a screen.
Parameter	A name the template's variables refer to: M , Motor , Area . Up to 16 per template; a letter or underscore, then letters, digits or underscores, up to 32 characters; not case-sensitive.
Reference	<code>\$M</code> or <code>\${M}</code> inside a variable field. Use the braces when more letters follow: <code>\${M}_speed</code> is M followed by <code>_speed</code> , while <code>\$M_speed</code> refers to a parameter called <code>M_speed</code> .
Argument	What a Faceplate widget or a dialog link puts in place of a parameter: a variable path such as <code>P101</code> , <code>Line2.Pump[3]</code> .

Where references are replaced. In every field that names a variable — the widget's **Variable**, a symbol's **Fault bit**, every alarm-list row, every dynamic rule's **Variable**, every action's **Variable** and dialog arguments, hotkey

actions, tab-page arguments, and the arguments of a Faceplate inside the template (a large faceplate can frame a smaller one: `M = $M.inlet_valve`). They are also replaced in display text — **Text**, **Confirm text**, alarm-row text and the template's **Title** — so a caption `$M` reads `P101` on the panel. A `$` in text that names no parameter is shown as written.

What is checked on save:

- In a template, every reference in a variable field must be one of its declared parameters. A `$` that is not a reference at all (`$ M`, `$1`) is refused.
- An ordinary screen has no parameters: a `$` reference in one of its variable fields is refused.
- A Faceplate must name a template that exists in the project (not an ordinary screen), give an argument for **every** parameter and none for a parameter the template does not have. Each argument must be a variable path.
- Faceplates may be nested (a template containing a Faceplate) at most **4** deep, and a chain that leads back to itself — template A framing B framing A — is refused as a loop.
- The IDE server refuses to delete a template that is still placed on a screen. If its file is deleted by hand, the Faceplates using it show a red dashed box (below).
- Adding a parameter to a template that is already placed is allowed; the save reports each screen whose Faceplates now lack that argument (in **Messages**), and those widgets show grey **?** until the argument is filled in.

Because of the first Faceplate check, create (and save) a template **before** the screens that use it.

Creating a template

1. Right-click **HMI** and choose **New screen...**
2. Enter a name, for example `Motor`, set **Kind** to **Template (faceplate)** and enter the first **Parameter** name (default `M`).
3. Click **Create**. The template opens as `<name> · Template`, 360 × 240 pixels, with the title `$M`.
4. With nothing selected, the properties panel lists the **Parameters**: edit a name or its *What it stands for* description, **Add parameter**, or remove one with **x**. Renaming a parameter renames every `$` reference to it in the template.
5. Draw the faceplate like any screen. In the **Variable** fields, the list offers `$M` first; if a parameter's description starts with a function-block type (`MOTOR_DOL instance`), it also offers `$M.<member>` for that block's inputs and outputs.
6. **Save screen**.

Size the template to its content: a Faceplate scales the whole template into its box.

Placing a template: the Faceplate widget

1. Open the screen, click **Add: Faceplate**. It takes the first template in the project; choose another under **Template**.
2. Under **Arguments**, enter a variable for each parameter (the list offers the controller's names).
3. **Fit: keep proportions** (default) scales the template uniformly and centres it in the box; **stretch to the box** fills the box, distorting it if the proportions differ.
4. Resize the box. The canvas shows the template's widgets as outlines, with the arguments already filled in, so each Faceplate shows which instance it is.

On the panel every widget inside a Faceplate is live: values, lamps, rules, buttons, input fields and confirmations work exactly as on a screen, with the argument in place of the parameter. A Faceplate whose template is missing

on the controller (not downloaded, or deleted by hand) is drawn as a red dashed box with the template's name and the reason, never as an empty box.

Opening a template as a dialog

A **Screen link** can open its target over the current screen instead of replacing it:

- 1. Select the screen link and choose the template (or screen) under **Opens screen**.
- 2. Set **Open as** to **dialog over this screen**.
- 3. If the target is a template, fill in its **Arguments** as for a Faceplate. A plain screen takes no arguments.

On the panel the dialog shows the target's title (with its arguments filled in) and a close button (✕); **Esc** closes the top dialog. A dialog may open another dialog, up to **3** stacked. The screen underneath keeps updating. A momentary button held in a dialog is released when the dialog closes.

Generate faceplate

Generate faceplate... (editor toolbar, or right-click **HMI**) creates a template from a function-block type — a block from the project library or one of the project's own function blocks:

- 1. Choose the **Function block**. The dialog shows how many lamps, values, buttons and fields the template will have.
- 2. Keep or change the **Template name** (default: the block's name).
- 3. Click **Generate**. The template opens for editing.

The template has one parameter, **I** (the instance), described as `<block> instance`, and a title label `$I – <block>`. The block's interface is laid out in a grid, 360 pixels wide:

Member	Widget	Details
BOOL output	Lamp	Two per row, labelled with the member name.
Numeric or STRING output	Label + Value	Unit guessed from the name (<code>_s</code> , <code>_ms</code> , <code>_h</code> / <code>_hours</code> , <code>_pct</code> ...) or a unit written in the declaration's comment (<code>[bar]</code> , <code>°C</code> , <code>%</code> ...); none otherwise. REAL values show one decimal.
BOOL input from the field	Lamp	Names starting <code>fb_</code> or <code>ext_</code> , or containing <code>feedback</code> , <code>running</code> , <code>trip</code> , <code>fault</code> or <code>interlock</code> : the plant drives these, so they are shown, never offered as buttons.
BOOL start/stop pair	Button ×2	When the block has a start-like input (a word start, run, on or open) and a stop-like one (stop, off or close), they become a latching pair: START runs Reset stop, Set start and asks <i>Start \$I?</i> ; STOP runs Reset start, Set stop (\$32.4.12).
other BOOL input	Button	Named like a command first. Momentary for names containing start, stop, reset or ack (start and stop without a partner ask <i>Start \$I? / Stop \$I?</i>); toggle for all others.
Numeric input	Label + Input field	

Members of other types (structures, arrays, other blocks) and the block's internal variables are not included. The result is an ordinary template: move, resize, delete and restyle anything, then save.

Note: Check that the generated buttons match how the block reads its inputs. `MOTOR_DOL` treats `start` and `stop` as *levels* (it runs while `start AND NOT stop`), so the generator makes them a latching pair: `START` resets `stop` and sets `start`, `STOP` resets `start` and sets `stop`. `reset` stays momentary (the block reacts to its rising edge). The field inputs `fb_running`, `ext_trip` and `interlock_ok` appear as lamps. A block that reads `start` and `stop` as *pulses* needs its buttons changed to momentary.

Worked example — two motors and a dialog

The program has two `MOTOR_DOL` instances, `P101` and `P102`, and a screen `main`.

1. **Create the template. New screen...**, name `Motor`, **Kind** *Template (faceplate)*, **Parameter** `M`. Set its description to `MOTOR_DOL instance`. Size 240×150 .
2. **Draw it.** A **Label** with text `$M` at the top; a **Plant symbol** *motor* with **Variable** `$M.running` and **Fault bit** `$M.fault`; a **Value** on `$M.run_hours`, unit `h`; a **Button** `START` on `$M.start`, **On press set**, **Confirm text** `Start $M?`; a **Button** `STOP` on `$M.start`, **On press reset**. **Save screen**.
3. **Place it twice.** Open `main`, **Add: Faceplate** twice. On the first, **Template** `Motor`, argument `M` = `P101`; on the second, `M` = `P102`. The canvas now shows `P101.running`, `P102.running` ... in the two boxes.
4. **A dialog.** Add a **Screen link** with text `P103...`, **Opens screen** `Motor`, **Open as dialog over this screen**, argument `M` = `P103`. **Save**.
5. **Download screens** (§32.7). Templates are files in `hmi/` like screens and are downloaded with them.

On the panel, `main` shows both motors live; pressing **P103...** opens the same faceplate for `P103` as a dialog, titled `P103`.

In `hmi/main.json` the Faceplate and the link read:

```
{ "id": "m1", "kind": "frame", "x": 40, "y": 40, "w": 240, "h": 150,
  "template": "Motor", "args": { "M": "P101" }, "fit": "contain" },
{ "id": "m2", "kind": "frame", "x": 300, "y": 40, "w": 240, "h": 150,
  "template": "Motor", "args": { "M": "P102" }, "fit": "contain" },
{ "id": "n1", "kind": "nav", "x": 40, "y": 220, "w": 120, "h": 40, "text": "P103...",
  "screen": "Motor", "popup": true, "args": { "M": "P103" } }
```

and `hmi/Motor.json` starts:

```
{ "name": "Motor", "title": "$M", "type": "template",
  "params": [ { "name": "M", "about": "MOTOR_DOL instance" } ],
  "width": 240, "height": 150,
  "widgets": [
    { "id": "sym", "kind": "symbol", "shape": "motor", "bind": "$M.running", "bind2": "$M.fault", ... },
```

To get a first version without drawing, use **Generate faceplate...** with `MOTOR_DOL` instead of steps 1–2: it creates a template `MOTOR_DOL` with parameter `I`, lamps for `run_cmd`, `running` and `fault`, values for `fault_code` and `run_hours` (unit `h`), lamps for the field inputs `interlock_ok`, `fb_running` and `ext_trip`, a latching `START/STOP` pair and a momentary `RESET` button, and input fields for `fb_timeout` and `dt_ms`. Place it with argument `I` = `P101`.

32.4.10 Input elements

Kind (toolbar)	Binding	Fields	What it does
Slider	Variable (number)	Orientation, Minimum, Maximum, Step, Decimals, Unit, Confirm text	Drag the thumb; the value is written once, when the finger or mouse is released. The value is shown beside the track. Arrow keys step by Step and write at once.
Knob	Variable (number)	Minimum, Maximum, Step, Decimals, Unit, Confirm text	A rotary control through 270°. Drag round it; written on release. Dragging through the gap at the bottom keeps the knob at the end it came from.
Meter	Variable (number)	Minimum, Maximum, Decimals, Unit, Colour (needle), coloured bands	A round gauge with a needle and the value below it. Bands (From, To, Colour) are drawn on the dial, for example 80 ... 100 in red.
7-segment	Variable (number)	Digits shown (1–16), Decimals, Colour	The value on a seven-segment display, right-aligned. A value too long for the display shows <code>-</code> in every digit rather than a number with its first digits cut off.
Checkbox	Variable (BOOL)	Text, Confirm text	A tick box: a tap writes the opposite of the current value.
Radio buttons	Variable	Orientation, Options, Confirm text	One labelled button per option; a tap writes that option's value.
Drop-down	Variable	Options, Confirm text	The same, as a drop-down list.
Array table	Variable (the ARRAY)	Columns, Decimals, Editable, Limits, Confirm text	One row per array element (§ below).
Tabs	per page	Pages	A row of tabs above an area that shows one page at a time (§ below).
Date/time	Variable, or none	Format	The panel's clock, or a date/time or TIME variable, formatted (§ below).

Options (radio buttons and drop-down) are a table of **Value written** and **Shown as**. A value is a number, `TRUE / FALSE`, or a text for a STRING variable; the same value may not be listed twice. The option whose value equals the variable's current value is shown selected. A value that matches no option — and a variable the controller does not report — shows **grey with ?**, never the first option as though it were selected.

Unknown is grey. Every input element whose variable the controller does not report is drawn grey with **?**: a slider or knob without its thumb, a meter without its needle, a checkbox with `?` in the box, a 7-segment display showing `?`.

Array table. Set **Variable** to the array itself, for example `Recipe.Temps`. The rows are the elements the controller reports as `Recipe.Temps[1]`, `Recipe.Temps[2]` ..., in index order (up to 500). For an array of structures, add one **Column** per member (**Member** `temp`, **Heading** `Temp °C`): the cell in row 3 of that column shows `Recipe.Steps[3].temp`. Without columns each element is one value. An element the controller does not report shows `?`; an array with no reported elements shows *no elements reported*. The table is read-only unless **Editable** is ticked; then a tap on a cell opens the keypad (§32.4.11) and writes that element (or member). With **Limits** ticked, a value outside **Minimum ... Maximum** is refused.

Tabs. Under **Pages, Add page**, give it a **Title** and choose what it **Shows**: another screen, or a template with an argument for each of its parameters (like a Faceplate). The page is scaled into the area below the tabs, keeping its

proportions. The first page is shown when the screen opens. All pages stay live while hidden, so switching tabs shows current values at once. On save, each page is checked like a Faceplate: the screen or template must exist, a template must get exactly its arguments, a template may not be used as a plain screen page, and a chain of pages and Faceplates may not lead back to itself or be nested more than 4 deep. A screen or template used as a page cannot be deleted.

Date/time. Leave **Variable** empty to show the panel's own clock (updated twice a second, even when the controller does not answer). A variable may hold:

Value	Shown as
a date and time: DT#2026-09-29-14:05:00 , 2026-09-29 14:05:00 , D#2026-09-29	that date and time, as the controller printed it (no time zone is applied)
a time of day: TOD#08:30:00	that time; year, month and day are empty
a TIME: T#1h2m3s , T#1.500s , T#250ms	a duration: d is days, H hours (all hours when the format has no d), m minutes, s seconds, SSS milliseconds
a number	seconds since 1970 (milliseconds for very large numbers), in the panel's local time

Format fields: yyyy (2026), yy (26), MM (09), M (9), dd , d , HH (00–23), H , mm , m , ss , s , SSS (milliseconds). Anything that is not a letter is printed as it stands; put letters in single quotes: dd.MM.yyyy 'at' HH:mm . Default yyyy-MM-dd HH:mm:ss . A letter that is not a field (YYYY , MMM) is refused on save. A value that is not a date or a duration shows ? .

Limits. A **Slider** or **Knob** always writes inside **Minimum ... Maximum** (which must be different, minimum first). On an **Input field** and an **Array table**, tick **Limits** to have values outside **Minimum ... Maximum** refused. The check is made on the panel **before** anything is written; the refusal is shown as a message at the bottom of the panel, for example *Not written - 95 is above the maximum 80 (allowed 0 ... 80)*, and nothing is sent. Limits are not ticked on fields saved before this version, so existing screens keep behaving as before.

Note: Limits on a panel are an operator aid, not protection. Anything that must never exceed a value is limited in the program.

Writes as (input field): *as the variable is* (default) writes a number to a numeric variable and text to a STRING; *a number or text (STRING)* fixes it. Limits apply to numbers only.

32.4.11 The on-screen keypad

An **Input field** (and an editable table cell) opens a keypad on the panel itself instead of relying on the device's keyboard. **Keypad:**

Setting	When the keypad opens
<i>on touch panels</i> (default, auto)	when the device has no fine pointer (a touch screen with no mouse)
<i>always</i>	always; the field cannot be typed into directly
<i>never</i>	never; type into the field and press Enter

The **numeric keypad** (for numbers) has 0–9 , . , - (changes the sign), ← (delete one character), **C** (clear), **Cancel** and **OK**, and shows the allowed range when **Limits** is ticked. The first key replaces the value shown. **OK** checks the value: a value that is not a number, or is outside the limits, is refused inside the keypad with the reason,

and the keypad stays open. The **text keypad** (for STRING variables) has digits, the letters in QWERTY layout, **.**, **_**, **Shift**, **Space**, **←**, **C**, **Cancel** and **OK**.

OK writes (after the **Confirm text**, if any). **Cancel**, **Esc** and a tap outside the keypad write nothing. A physical keyboard also works while the keypad is open: digits and letters, **Backspace**, **Enter** (OK) and **Esc** (Cancel). While the keypad is open, hotkeys do not act.

32.4.12 Actions

A **Button**, and any display element with **On click** ticked (Label, Value, Lamp, Box, Plant symbol, Rectangle, Ellipse, Polygon, Line, Image, Meter, 7-segment, Date/time), can run a list of **actions** when it is pressed. Each action is one declared step — there is no scripting:

Action	Fields	Effect
Set (write TRUE)	Variable	Writes TRUE.
Reset (write FALSE)	Variable	Writes FALSE.
Toggle a BOOL	Variable	Writes the opposite of the current value. Refused if the value is not known.
Write a constant	Variable, Value	Writes a number, TRUE/FALSE or a text.
Increase by a step / Decrease by a step	Variable, Step, Minimum, Maximum	Reads the current value, adds (subtracts) Step , keeps the result inside Minimum ... Maximum , writes it. Refused if the value is not known.
Go to screen	Screen	Opens another screen in place.
Open a dialog	Opens, arguments, Dialog title	Opens a screen, or a template with an argument for each parameter, as a dialog (§32.4.9).
Close this dialog	—	Closes the dialog the widget is in. Does nothing on a screen.

Edit the list in the **Actions** section of the properties panel: **Add action**, choose the action, fill in its fields, reorder with ↑ / ↓, remove with ×. Up to 16 actions.

How a list runs. The **Confirm text**, if any, is asked **once**, before the list. The actions then run in order. The list **stops at the first write the controller refuses** (the reason is shown at the bottom of the panel): a START that could not reset STOP does not go on to set START. A second press while a list is still running is ignored. An element disabled by an **Enabled** rule (§32.4.8) — or whose Enabled variable is not reported — does not run its actions. **Enabled** can be given to any element with **On click**.

Buttons. A button either has an **On press** (momentary, set, reset, toggle — §32.4.2) or an actions list, not both: adding the first action replaces **On press**, and removing the last one restores it (momentary). A screen file that has both is refused on save. Buttons saved before actions existed keep their **On press** unchanged.

What is checked on save: every action's variable is a variable path (or, in a template, uses its parameters); a constant is a number, TRUE/FALSE or a text; a step is greater than 0 and the minimum is less than the maximum; a **Go to screen** target exists and is a screen (a template needs **Open a dialog** with its arguments); an **Open a dialog** target exists, and a template gets exactly its arguments while a screen gets none. **On click** is refused on elements that already react to a press of their own (input fields, sliders, knobs, checkboxes, radio buttons, drop-downs, tables, tabs, Faceplates, screen links, trends, alarm lists), and on an element with no actions.

32.4.13 Hotkeys

With nothing selected, the properties panel lists the screen's **Hotkeys**. **Add hotkey**, choose the **Key** — **F1 ... F12** or **Esc** — tick **Shift** if wanted, and give it an actions list as in §32.4.12. A key (with or without Shift) may be used once per screen.

Hotkeys act only on the view on top: the top dialog while one is open (a template or screen opened as a dialog brings its own hotkeys), the screen otherwise. They do not act while someone is typing into a field, while a drop-down has focus, or while the keypad is open, and never with **Ctrl**, **Alt** or the Windows/Command key. Holding a key down runs its list once. **Esc** without an Escape hotkey still closes the top dialog. A key used as a hotkey no longer does what the browser would do with it (F1 help, F5 reload, F11 full screen).

Worked example — setpoint, mode, speed, START/STOP and a help key

The program has `TT101_SP` : REAL (a temperature setpoint, 20–80 °C), `Mode` : INT (0 Off, 1 Manual, 2 Auto), `Fan_Speed` : REAL (0–100 %) and a `MOTOR_DOL` instance `P101`. There is a screen `help` with a **Button Close** whose actions are **Close this dialog**.

1. **Setpoint with keypad and limits.** Add an **Input field**, **Variable** `TT101_SP`, **Decimals** 1, **Keypad on touch panels**, tick **Limits**, **Minimum** 20, **Maximum** 80, **Confirm text** `Change the setpoint?`. On a touch panel a tap opens the numeric keypad showing `20 ... 80`; 95 is refused in the keypad, 65 is written after the confirmation.
2. **Mode drop-down.** Add a **Drop-down**, **Variable** `Mode`, **Options**: 0 *Off*, 1 *Manual*, 2 *Auto*. If the program ever sets `Mode` to 7, the drop-down shows grey ?.
3. **Speed slider.** Add a **Slider**, **Variable** `Fan_Speed`, **Minimum** 0, **Maximum** 100, **Step** 5, **Unit** `%`. The value is written when the slider is released, rounded to a multiple of 5.
4. **START and STOP as a latching pair.** `MOTOR_DOL` reads `start` and `stop` as levels (the motor runs while `start AND NOT stop`). Add a **Button START**, **Variable** `P101.start`, **Confirm text** `Start P101?`, actions **Reset** `P101.stop`, **Set** `P101.start`. Add a **Button STOP**, **Variable** `P101.stop`, actions **Reset** `P101.start`, **Set** `P101.stop`.
5. **F1 opens help.** Select nothing, **Add hotkey**, **Key** F1, action **Open a dialog**, **Opens** `help`, **Dialog title** `Help`.
6. **Save screen** and **Download screens**.

In the screen file these read:

```
{ "id": "sp", "kind": "input", "x": 40, "y": 40, "w": 120, "h": 34, "bind": "TT101_SP",
  "digits": 1, "limits": true, "min": 20, "max": 80, "confirm": "Change the setpoint?" },
{ "id": "mode", "kind": "combo", "x": 40, "y": 90, "w": 160, "h": 34, "bind": "Mode",
  "options": [ { "value": 0, "text": "Off" }, { "value": 1, "text": "Manual" },
    { "value": 2, "text": "Auto" } ] },
{ "id": "spd", "kind": "slider", "x": 40, "y": 140, "w": 260, "h": 44, "bind": "Fan_Speed",
  "min": 0, "max": 100, "step": 5, "unit": "%", "orientation": "h" },
{ "id": "start", "kind": "button", "x": 40, "y": 200, "w": 120, "h": 44, "text": "START",
  "bind": "P101.start", "confirm": "Start P101?",
  "actions": [ { "op": "reset", "bind": "P101.stop" }, { "op": "set", "bind": "P101.start" } ] },
{ "id": "stop", "kind": "button", "x": 180, "y": 200, "w": 120, "h": 44, "text": "STOP",
  "bind": "P101.stop",
  "actions": [ { "op": "reset", "bind": "P101.start" }, { "op": "set", "bind": "P101.stop" } ] }
```

and, at screen level:

```
"hotkeys": [ { "key": "F1", "actions": [ { "op": "popup", "screen": "help", "title": "Help" } ] } ]
```

32.4.14 Alarm table

An **Alarm table** shows the **controller's** alarms — the ones defined in `controller.json` and raised, timestamped and journalled by the controller's alarm engine (Chapter 27). The screen does not decide what is an alarm: the program sets the condition bit, the controller keeps the state (ISA-18.2, §27.2) and the journal, and the table only shows them and passes on acknowledgements. It is a different thing from the **Alarm list** (§32.4.3), which shows screen-defined BOOLS and has no acknowledgement.

Field	Meaning
Text	The title in the table's header.
Show from	The lowest priority shown: <i>DIAGNOSTIC (all)</i> , <i>LOW</i> , <i>HIGH</i> or <i>CRITICAL</i> and above.
Alarms	Optional name filter, comma separated: an exact alarm name (<code>PT101_HIGH</code>) or a prefix ending in <code>*</code> (<code>TT_*</code>). Empty shows every alarm. Not case-sensitive.
Rows	At most this many rows (1–200, default 50).
Format	How times are written, as for Date/time (§32.4.10); default <code>yyyy-MM-dd HH:mm:ss</code> .
Columns	Which columns, in which order: Time (since when), Priority , State , Name , Message , Value (the condition bit), Acknowledged by . Default: Time, Priority, State, Message.
Colour per priority	Default CRITICAL red, HIGH orange, LOW yellow, DIAGNOSTIC grey.

Active tab. Every alarm that is not NORMAL — active, cleared but unacknowledged, shelved, suppressed or out of service. Unacknowledged alarms come first, then by priority (highest first), then newest first. Each row carries its priority colour; an **unacknowledged** row blinks in that colour (1 Hz) until someone acknowledges it, and only then — an acknowledged alarm that is still active stays coloured but still, so what moves is always what is new.

Acknowledge acknowledges the selected row (tap a row to select it). **Acknowledge all shown** asks for confirmation and acknowledges the unacknowledged rows **this table shows** — alarms hidden by its priority or name filter are not touched. Both need a user with the **operator** role or higher; the controller records each acknowledgement against the user (§27.2). When the panel knows the logged-in user is a viewer, the buttons are not shown; if the controller refuses an acknowledgement, its reason appears at the bottom of the screen.

History tab. The controller's alarm journal — its latest 200 events — newest first: time, event (RAISE, CLEAR, ACK, SHELVE...), alarm, priority, who, batch. The same filter applies.

Times. A time the controller stamped while its clock was not synchronised (a controller with no NTP and no RTC) is shown with ~ in front, and its tooltip reads *controller clock not synchronised*. In **Open as operator...** from the IDE, times are converted by the IDE from the controller's uptime (the tooltip says so), and the **Value** and **Acknowledged by** columns are not available there (shown as ? and empty); on the panel served by the controller all columns come from the controller.

The table is read once per screen poll, but never more often than once a second, and only while something on the screen needs it.

Alarm counts. Two names that are not program variables may be bound: `@alarms.unacked` (number of unacknowledged alarms) and `@alarms.active` (number of active alarms). They are offered in the **Variable** list and can be shown by a **Value**, **Lamp** (lit while the count is above zero), **Bar**, **Meter**, **7-segment** or **Trend**, or followed by any dynamic rule (§32.4.8). They are read-only: a button, field, action or alarm-list row cannot use them, and any other @ name is refused on save. When the controller's alarms cannot be read they are unknown (grey ?), never zero.

Worked example — alarm banner with a blinking header lamp

The controller has alarms `TT101_HIGH` (HIGH), `PT101_TRIP` (CRITICAL) and `LS201_LOW` (LOW) in `controller.json` (Chapter 27).

1. **Header lamp.** Add a **Lamp** at the top of the screen, **Variable** `@alarms.unacked`, **Text** `ALARM`, **Colour when ON** `#ef4444`. Under **Dynamic properties** add **Blink (1 Hz)**, rule *TRUE when a comparison holds*, variable `@alarms.unacked`, `>` `0`.
2. **Count.** Beside it a **Value**, **Variable** `@alarms.active`.
3. **Table.** Add an **Alarm table**, **Text** `Plant alarms`, keep the default columns, tick **Name** as well and move it up after **Priority**.
4. Optionally a second **Alarm table** for the shift supervisor: **Show from** *HIGH and above*, **Alarms** `PT*`, `TT*`.
5. **Save screen** and **Download screens**.

When `TT101_HIGH` is raised, the lamp lights and blinks, the count reads 1 and the row blinks orange at the top of the table. Select it and press **Acknowledge**: the row stops blinking (state *ACTIVE*), and the lamp goes dark once no alarm is left unacknowledged. When the condition goes away the row disappears; **History** shows RAISE, ACK and CLEAR with the user who acknowledged.

In the screen file:

```
{ "id": "hdr", "kind": "lamp", "x": 10, "y": 8, "w": 120, "h": 34, "bind": "@alarms.unacked",
  "text": "ALARM", "onColor": "#ef4444",
  "dyn": { "blink": { "bind": "@alarms.unacked", "type": "cond", "op": ">", "value": 0 } } },
{ "id": "cnt", "kind": "value", "x": 140, "y": 8, "w": 80, "h": 34, "bind": "@alarms.active" },
{ "id": "tbl", "kind": "alarmtable", "x": 10, "y": 60, "w": 900, "h": 300, "text": "Plant alarms",
  "columns": ["time", "priority", "name", "state", "message"],
  "minPriority": "DIAGNOSTIC", "rows": 50 },
{ "id": "sup", "kind": "alarmtable", "x": 10, "y": 370, "w": 900, "h": 160, "text": "Supervisor",
  "minPriority": "HIGH", "names": ["PT*", "TT*"] }
```

32.4.15 Text lists and languages

Texts an operator reads can come from **text lists** in several languages instead of being typed into each widget. The lists are one file per project, `hmi/texts.json`, edited with **Text lists...** in the screen editor's toolbar:

Field	Meaning
Languages	The language codes the panels offer, comma separated: <code>en, he, ar, de</code> (two or three lower-case letters, optionally <code>-XX</code>). Up to 16.
Default	The language a panel starts in, and the one used when a translation is missing (★ in the table).
List	A named list (<code>Modes</code> , <code>Run</code> , <code>Titles</code> — letters, digits, underscore). New list , Rename , Delete list .
entries	One row per key with a text per language. For a list shown by value, the key is the value: <code>0</code> , <code>1</code> , <code>2</code> , or <code>TRUE</code> / <code>FALSE</code> .

Export CSV writes every list as `list,key,en,he,...` (one row per entry) for a translator; **Import CSV...** reads such a file back (comma or semicolon separated, UTF-8). **Save** stores `texts.json`, checked like a screen: a bad language code, a translation in a language that is not listed, or a text longer than 200 characters is refused.

Using a list on a screen. In a text field write:

Text	Shows
@Titles.main	The entry <code>main</code> of list <code>Titles</code> , in the panel's language. Anywhere text is shown.
@Modes	The entry whose key is the widget's own value — on a Value , Lamp , Button , Checkbox or Plant symbol with a Variable . On a <code>Value</code> it replaces the number.
@@home	The literal text <code>@home</code> .
anything else	The text as written, as before.

`@List.key` may also stand in **Radio buttons** and **Drop-down** option texts, **Alarm list** row texts, **Tab** titles, dialog titles of a **popup** action, the screen title, and the results of a dynamic **Text** rule of type *map* (§32.4.8) — so a map can say `value ≥ 2 → @Modes.2`. A *format* rule's result is the value's own text and is not looked up. `@List` without a key on a label is refused on save (a label has no value to choose by).

A missing translation falls back to the **default** language, then to the **key** itself — never to an empty box. A value with no entry in the list shows the value; a variable the controller does not report shows **?** in grey. A reference to a list or key that `texts.json` does not have is a **warning** in **Messages** when the screen (or `texts.json`) is saved — the screen still saves, and the panel shows the key.

Choosing the language on the panel. Tick **Languages — language selector in the header** in the screen's properties (nothing selected). The panel then shows a language box in its header; the choice is remembered **by that panel** (the browser's local storage), not by the screen or the user. For Hebrew, Arabic, Persian and Urdu the text of text elements runs right to left; the layout is **not** mirrored — a pump stays left of its valve, and tables, trends and recipes keep their columns. `texts.json` goes to the controller with **Download screens**.

Worked example — a mode shown in two languages

1. **Text lists...** → **Languages** `en`, `he`, **Default** `en`. **New list** `Modes`; entries `0` `Off` / `1` `כבוי` `Manual` / `2` `ידני` `Auto` / `אוטומטי`. **New list** `Titles`; entry `main` Line 2 / 2 קו. **Save**.
2. On the screen: **Title** `@Titles.main`; tick **language selector in the header**.
3. Add a **Value**, **Variable** `Line2.Mode`, **Text** `@Modes`.
4. Add a **Drop-down**, **Variable** `Line2.Mode`, options `0` → `@Modes.0`, `1` → `@Modes.1`, `2` → `@Modes.2`.
5. **Save screen**, **Download screens**.

With `Line2.Mode = 1` the value reads *Manual*; choosing **HE** in the header shows *ידני*, and the drop-down lists the Hebrew texts. In the files:

```
{ "languages": ["en", "he"], "default": "en",
  "lists": { "Modes": { "0": { "en": "Off", "he": "כבוי" }, "1": { "en": "Manual", "he": "ידני" },
                     "2": { "en": "Auto", "he": "אוטומטי" } },
    "Titles": { "main": { "en": "Line 2", "he": "2 קו" } } }
```

```
{ "title": "@Titles.main", "langSelector": true, "widgets": [
  { "id": "mode", "kind": "value", "bind": "Line2.Mode", "text": "@Modes", "x": 20, "y": 60, "w": 180,
    "h": 34 },
  { "id": "sel", "kind": "combo", "bind": "Line2.Mode", "x": 220, "y": 60, "w": 180, "h": 34,
    "options": [ { "value": 0, "text": "@Modes.0" }, { "value": 1, "text": "@Modes.1" },
                  { "value": 2, "text": "@Modes.2" } ] ] }
```


32.4.16 Access per element

Every widget has an **Access** section:

Field	Meaning
Visible to	<i>everyone (viewer) (default), operator or engineer.</i> Below it, the element is not drawn at all.
Operated by	<i>operator (default) or engineer.</i> Below it, the element is drawn dimmed and locked: presses, drags, taps and the keypad do nothing.

The panel takes the role of whoever is logged in (`GET /api/whoami` on a panel; in **Open as operator...** the IDE's own session on the controller). Roles are the controller's accounts: viewer < operator < engineer (§32.6.2). An element that writes — a button, an input element, a recipe, an alarm table's acknowledge buttons — needs an **operator** even when **Operated by** is not set; display elements are never locked by default. **Access on a Faceplate or Tabs** applies to everything inside it.

Access is **display only**. The controller still decides every write by the user's role: a viewer is refused whatever a screen shows, and a screen cannot give anyone more than their account allows. Use access to keep engineering settings off an operator's screen, not as the only protection. When the page cannot tell the role — the controller has no accounts and **Screens may write** is on, or an older IDE — nothing is hidden or locked.

In the screen file: `"access": { "visible": "operator", "operate": "engineer" }`. An unknown role is refused on save.

32.4.17 Recorded trends

A **Trend** with **Recorded** ticked is recorded **on the controller**: the controller samples the variable into memory, and when the screen opens the panel draws that history at once and then continues live — the operator sees the last minutes even on a panel that was just switched on.

Field	Meaning
Recorded	Record this variable on the controller.
Sample ms	How often (100 ms to 1 h). Empty: 1000 ms.
Samples kept	How many (1 to 86 400). Empty: what the trend's Window shows (window ÷ sample period).

The recording is part of the controller configuration. When the configuration is downloaded (**Download hardware**, Deploy, starting the simulator), the IDE collects every recorded trend on every screen — through Faceplates, tab pages and dialogs, with each template's arguments filled in (a motor faceplate placed for P101 and P102 records `P101.speed` and `P102.speed`) — into `web.trends`. A variable recorded by several trends is recorded once, at the shortest period and the largest number of samples. An entry you wrote into `web.trends` by hand is kept and wins. So: **after adding or changing a recorded trend, Download hardware** (which restarts the runtime); a screen download alone does not start a recording.

Limits: at most **32** variables; at most 86 400 samples each; a period of at least 100 ms. What does not fit is left out and said in **Messages** (as a warning, naming the variables), never sent to be refused by the controller. The recording is memory only — never the SD card — so it starts again empty after a restart of the runtime. It is not a historian.

On the panel: missed samples are gaps in the line, not joined. When the controller's clock differs from the panel's by more than two sample periods (a controller without time synchronisation), the history is moved so its last sample is *now* and the trend says *controller clock differs - aligned*. A trend whose variable is not recorded on the

controller says *not recorded yet - Download hardware* and shows live values. In **Open as operator...** from the IDE there is no recorder to ask: the trend shows live values and says *history is on the controller*.

In the screen file: `{ "kind": "trend", "bind": "TT101", "span": 600, "recorded": true, "period_ms": 1000 } → web.trends: [{ "bind": "TT101", "period_ms": 1000, "depth": 600 }]`.

32.4.18 Recipes on the panel

A recipe is a set of values written to the controller together — a product's temperatures, times and speeds. The **definition** (which variables, in which order, with labels, units and limits) is part of the project; the **value sets** are made by operators and stored **on the controller**.

Defining a recipe. Recipes... in the editor's toolbar → **New...** (a name: letters, digits, dot, dash, underscore). For each parameter:

Column	Meaning
Variable	The variable written — pick it from the list of the controller's names, or type it (<code>Mix.Temp</code> , <code>Line2.Speed[3]</code>). Each once.
Label	What the operator reads. Empty: the variable name.
Unit	Shown beside the value.
Min, Max	Optional limits. A parameter with limits takes a number in range; one without takes any text the controller's write accepts (a BOOL, a STRING).

↑ / ↓ set the order, which is the order of the writes. **Save** stores `hmi/recipes/<name>.json`. A definition still shown by a Recipe widget cannot be deleted.

The Recipe widget. Add: Recipe, choose the **Recipe**. On the panel it shows the recipe's sets and a grid (parameter, value, unit, range, and the value **in the PLC** now — orange where it differs from the grid):

Button	Does	Needs
set list	Reads the chosen set from the controller into the grid.	—
Load from PLC	Reads the current values into the grid (a value the controller does not report stays —).	—
tap on a value	Opens the keypad; a value outside the limits is refused there.	—
Save	Stores the grid as the chosen set (with no set chosen, as Save as...).	operator
Save as...	Asks for a new set name (keypad), asks before replacing an existing set.	operator
Delete	Deletes the chosen set, after asking.	operator
Download to PLC	Checks every value first (none missing, all in range), asks, then writes them in the definition's order and stops at the first the controller refuses — the message says how many were written and which one stopped it.	operator

Every write is an ordinary write, audited against the user like a button press; the writes of a **Download to PLC** also carry the note `recipe <recipe>/<set>` in the controller's audit log, so one download's records read together (a controller older than this ignores the note). Sets are stored as `<recipe_dir>/<recipe>/<set>.json` on the controller (`web.recipe_dir` , default `/var/lib/zeus-plc/recipes`); set names follow the same rule as screen names. The definition goes with **Download screens**.

In **Open as operator...** from the IDE the sets are not reachable (they are the controller's): the list says *sets are on the controller*, and **Save**, **Save as...** and **Delete** are unavailable; **Load from PLC** and **Download to PLC** work.

Worked example — a mixing recipe

1. **Recipes...** → **New...** `Mix` . Parameters: `Mix.Temp` , *Temperature*, °C, 20, 90; `Mix.Time` , *Time*, s, 0, 3600; `Mix.Product` , *Product* (no limits — a STRING). **Save**.
2. On the screen **Add: Recipe**, **Recipe** `Mix` . Set its **Access** → **Operated by** to *engineer* if only engineers may change recipes.
3. **Save screen**, **Download screens**.
4. On the panel: **Load from PLC**, tap *Temperature* and enter 65, **Save as...** `Cola_1` . Later: choose `Cola_1` , **Download to PLC**, confirm — three writes, in order, each in the audit log.

```
{ "title": "Mixing", "params": [
  { "bind": "Mix.Temp", "label": "Temperature", "unit": "°C", "min": 20, "max": 90 },
  { "bind": "Mix.Time", "label": "Time", "unit": "s", "min": 0, "max": 3600 },
  { "bind": "Mix.Product", "label": "Product" } ] }
```

A stored set: { "def": "Mix", "values": { "Mix.Temp": "65", "Mix.Time": "300", "Mix.Product": "Cola" }, "saved_at": 1790000000000, "saved_by": "olga" } .

32.5 Binding rules

- Bind to variables of the **running** program. After renaming a variable, correct the screens as well: a screen refers to a variable only by its name, and a renamed variable shows as not reported.
- Bind writable widgets only to variables the program **reads** (setpoints, commands). A value the program assigns every scan reverts on the next scan. `%I` inputs cannot be written.
- `CONSTANT` variables cannot be written.
- Function-block members are bound by their full name: `P101.Running` , `TT_01_01.EU_Value` . SFC step flags work too: `Filling.X` , `Filling.T` .
- Prefer one clear variable per displayed item over deep paths into library blocks; it keeps screens stable when a block's internals change.

32.6 The screens server

32.6.1 The Screens Server page

Open **HMI** → **Server**.

Field	Config key (<code>"web"</code>)	Default	Meaning
Serve screens	<code>enabled</code>	off	Start the screens server with the runtime.
Listen on	<code>bind</code>	<code>127.0.0.1</code>	<code>127.0.0.1</code> – this controller only (tunnel to reach it) or <code>0.0.0.0</code> – any interface, reachable from the network .

Field	Config key (<code>web</code>)	Default	Meaning
Port	<code>port</code>	8080	1024...65535. Ports below 1024 are refused, because the runtime drops the privileges they need; forward 80 to 8080 in the firewall if a low port is required.
Screens may write	<code>allow_write</code>	no	no – screens are read-only or yes – buttons and inputs can change values .
Files	<code>root</code>	<code>/var/lib/zeus-plc/hmi</code>	Directory on the controller the screens are served from.
—	<code>cpu</code>	-1	CPU core for the server thread (edit <code>controller.json</code>).

Buttons: **Save**, **Download screens to controller** (§32.7) and **Users & Roles...**

To enable the screens server:

1. Create the accounts operators will use (**Users & Roles...**, Chapter 34) if the panel is on the network.
2. Open **HMI** → **Server**.
3. Set **Serve screens** to **on**, choose **Listen on**, **Port** and **Screens may write**.
4. Click **Save**.
5. Choose **Online** → **Download & Upload** → **Download hardware** to apply the configuration (the runtime restarts).

WARNING: A hardware download stops the control loop while the runtime restarts.

6. Download the screens (§32.7).

32.6.2 Users and roles on a panel

The screens server uses the **same accounts** as the IDE (**Users & Roles...**). There is no second user list for panels: one place to add someone, one place to remove them, one audit log.

Role	On a screen
viewer	Open screens and read values.
operator	Also press buttons and type values.
engineer	Same as operator.

When the project has accounts, **everything** is protected — screens, drawings, text lists and values alike, since the layout of a plant is itself information. The operator page shows its own log-in page (§32.6.3); other clients may use HTTP Basic authentication. A write by a viewer, or by nobody, is refused and audited (`web: DENIED write from ... - role is below operator`).

Listen on	Accounts	Result
<code>127.0.0.1</code>	none	Served to anyone with a login on the controller (or an SSH tunnel).
<code>127.0.0.1</code>	yes	Login required.

Listen on	Accounts	Result
0.0.0.0	none	The server refuses to start (web: refusing to serve screens on 0.0.0.0 with no accounts configured).
0.0.0.0	yes	Login required.

Important: HTTP is not encrypted. With **Listen on** 0.0.0.0, the password crosses the network in clear on every request. That is acceptable on a dedicated control segment; on any other network, keep the server on 127.0.0.1 and reach it through an SSH tunnel.

32.6.3 Logging in on a panel

A panel has no keyboard for the browser's password box, so when the controller has accounts the operator page draws its **own log-in page: User, Password and Log in**. On a touch panel a tap on either field opens the on-screen keypad (§32.4.11), the password shown as dots. The log-in is `POST /api/login`; the controller answers with a session cookie that the browser sends with every request after it, and the screen opens.

What happens	On the panel
Wrong name or password	<i>the user name or password is not correct</i> under the form (the same message for both).
5 failed log-ins from this panel's address	<i>Too many failed log-ins - try again in N s</i> , counting down; Log in is disabled until then (60 s).
Logged in	The header shows the user and role and, for a log-in made on this page, Log out .
Log out	Ends the session on the controller and returns to the log-in page.
The session ends later (12 h idle by default, 7 days at most, or the runtime restarted)	The next request is refused, the screen stops and the log-in page returns; after logging in, the same screen opens again.
The user's role may not write (a viewer pressing a button that is not locked)	<i>olga (viewer) may not change values on this controller</i> at the bottom of the screen; the user stays logged in.

A browser that already sends HTTP Basic credentials (a user who logged in through the browser's own dialog) is simply logged in, without **Log out** — those credentials live in the browser and end when it closes. Every request the page makes is marked as coming from the panel (`X-Zeus-Panel: 1`), so the browser's own password box never appears over the panel. The idle time is `web.session_idle_min` in `controller.json` (minutes, default 720 = 12 h), applied by **Download hardware**.

Important: the password crosses the network in clear when logging in (HTTP). Use a dedicated control segment, or keep the server on 127.0.0.1 and tunnel (§32.6.2).

What the page reads. The page asks the controller for exactly the variables the open screen and its dialogs show (`GET /api/symbols?names=...`, or `POST` when the list is long), not the whole symbol table. A variable the controller does not have is reported as unknown and drawn grey; an answer cut short by the controller's size limit is treated as missing, never as zero. An **Array table** reads the full list once when the screen opens, to learn which elements the array has.

32.7 Downloading screens

To download the screens:

1. Save every open screen.
2. Choose **Online** → **Download & Upload** → **Download screens** (or **Download screens to controller** on the Server page, or the context menu of **HMI**).
3. Read the messages: one line per screen, drawing and asset copied.

The download sends every screen in `hmi/`, every imported drawing, every picture in the image pool (`images/`), the text lists (`texts.json` , §32.4.15), the recipe definitions (`recipes/` , §32.4.18), and the operator page itself (`index.html` , `hmi.js` , `hmi-core.js` , `hmi-symbols.js` , `hmi.css`) to the **Files** directory. On a controller it is copied over SSH (engineer role and an SSH key are required, Chapter 34); on the simulator it is written to the simulator's own screen directory. The result names the address to open on the panel, for example `http://10.10.0.20:8080/` .

A screen download does **not** stop the control loop and needs no restart; a panel shows the new screens after a reload. **Recorded trends** (§32.4.17) are the exception: the recording is controller configuration, so it needs a **Download hardware**.

32.8 Opening screens

Where	How
In the IDE, for testing	Open as operator... in the screen editor. The page runs through the IDE's connection to the current target, so it needs the IDE to be logged in.
On a panel or any browser	<code>http://<controller>:<port>/</code> opens <code>main</code> ; <code>http://<controller>:<port>/?screen=<name></code> opens another screen.

It is the same operator page in both places: the screen an engineer tests is the screen the operator gets. Use a Chromium-based browser in kiosk mode on the panel.

When the controller cannot be reached, or a variable is not reported, the page says so rather than showing old values as current.

32.9 Availability on the simulator

The standalone Windows simulator of the IDE package (`bin\zeus-sim.exe`) has no screens server, so a panel cannot connect to it; **Open as operator...** still works through the IDE. When the IDE runs the full Linux runtime under WSL2, the simulator serves screens from its own directory after **Download screens**.

32.10 Troubleshooting

Symptom	Cause	Remedy
Panel cannot connect	Server off, not downloaded, bound to <code>127.0.0.1</code> , wrong port, firewall	§32.6; check the log for the web server's start line.

Symptom	Cause	Remedy
Server does not start; log says <i>refusing to serve screens ... with no accounts</i>	0.0.0.0 without accounts	Add users, Download hardware .
Page loads but shows <i>not found</i>	Screens not downloaded, or Files directory differs	Download screens .
Values show ? or grey	Variable not in the running program (renamed, not downloaded)	Correct the binding; download the program.
Button does nothing	Screens may write is off; user is a viewer; variable is assigned by the program	\$32.4.2, \$32.5, \$32.6.2.
Login prompt keeps coming back	Wrong password, or an account still on its initial password that was changed but not downloaded	Users & Roles... , Download hardware .
A screen node shows broken	Its JSON file cannot be read or refers to an unknown widget or shape	Fix or delete <code>hmi/<name>.json</code> .
Save screen is refused naming a widget and a dynamic property	A rule type not allowed for that property, a bad operator, colour or format, or a missing variable	Correct the rule the message names (\$32.4.8).
A widget is grey with ? though its own binding is fine	One of its rules follows a variable the controller does not report	Check each rule's Variable (\$32.4.8).
A Faceplate shows a red dashed box with the template's name	The template is not on the controller (screens not downloaded since it was created), or its file is broken	Download screens ; open the template and save it again.
Save screen is refused: <i>\$X is not a parameter of this template / is a template parameter, and this is a screen</i>	A \$ reference to an undeclared parameter, or a \$ in an ordinary screen	Declare the parameter, or correct the variable (\$32.4.9).
Save screen is refused: <i>there is no template ..., needs an argument for ..., makes a loop, nested more than 4 deep</i>	A Faceplate or dialog link that does not match the project	Save the template first; fill every argument; break the loop (\$32.4.9).
A text shows its key (<code>main</code> , <code>1</code>) instead of the translation	The list or key is not in <code>texts.json</code> , or the screens were not downloaded since	Text lists... (the save warning in Messages names it); Download screens .
Save screen is refused: <i>shows a list entry by value, and a "label" has no value</i>	<code>@List</code> without a key on a widget with no variable	Use <code>@List.key</code> , or put it on a Value, Lamp, Button, Checkbox or symbol with a variable (\$32.4.15).
An element is missing or dimmed for one user and not another	Its Access (\$32.4.16)	Check Visible to / Operated by ; the user's role is in Users & Roles....
A recorded trend says <i>not recorded yet - Download hardware</i>	<code>web.trends</code> on the controller does not have it	Download hardware (\$32.4.17).

Symptom	Cause	Remedy
Messages warns ... <i>variables are marked recorded; the controller records at most 32</i>	More than 32 recorded variables	Untick Recorded on trends that do not need history.
A Recipe shows a red dashed box	The definition is not on the controller, or its file is broken	Recipes... , then Download screens .
Recipe Save / Download to PLC are greyed	Not logged in as operator, locked by Access , or (Download) a value is missing	Log in; fill every value (§32.4.18).
The log-in page keeps returning	The session ended (idle, 7 days, runtime restarted)	Log in again; raise <code>web.session_idle_min</code> if needed (§32.6.3).
A symbol shows <i>missing drawing</i>	Its drawing was removed from <code>hmi/symbols/</code> , or the screens were not downloaded since it was inserted	Insert it again from Symbol library... or Import SVG... , then Download screens .
Save screen is refused: <i>"stateColor" applies to an imported drawing</i>	Colour with state set on a built-in shape (for example by editing the JSON)	Remove <code>stateColor</code> , or use an imported drawing (§32.4.4).
An Image widget shows an empty frame	The picture was removed from the pool, or the screens were not downloaded since it was imported	Images... , then Download screens .

32.11 Local display on the controller

The controller can show its own operator screens on a monitor or touch panel plugged into its **HDMI port** — no PC and no separate panel. It is the same operator page (§32.8), opened by a full-screen browser on the controller itself at `http://127.0.0.1:<port>/`.

Note: the local display has **not yet been bench-tested** on a controller with a display attached. Treat the first installation on each carrier board and display as a bench test: rotation, touch calibration and graphics may need adjusting (see `service/controller/kiosk/README.md`).

Requirements

Controller	IPC-CM400200 / IPC-CM500200 (Raspberry Pi CM4/CM5), Debian 12/13 or Raspberry Pi OS Lite, KMS graphics (the default). 2 GB RAM or more recommended — the browser uses 300–400 MB.
Screens server	Enabled (§32.6.1) and bound to <code>127.0.0.1</code> or <code>0.0.0.0</code> — the display always opens <code>127.0.0.1</code> . Download hardware after enabling it, then Download screens .
Installation	SSH key access and passwordless <code>sudo</code> for the target's SSH user (like Deploy, Chapter 34), and internet access on the controller during installation: the browser (about 250 MB) comes from the controller's package sources.
Display	Any HDMI monitor; a USB touch panel for operation. Connect it before the controller starts.

Installing. Select the controller as target, open **Controller Settings** (engineer role) and go to **Local display (HDMI)**:

1. Set the options below and press **Install on controller...** The IDE copies the display files to the controller and runs its installer with `sudo`: it installs the browser, creates the unprivileged `zeus-kiosk` account, installs the `zeus-kiosk` service and takes over the first console (its login prompt is turned off). This can take several minutes. The installer's output appears in **Messages**.
2. The project's settings are then applied: the display starts if **Enable local display** is ticked.

Run **Install on controller...** again to update the display files after an IDE update; settings on the controller are kept.

Settings

Setting	Meaning
Enable local display	On: the display service is enabled and started, and starts at every boot. Off: it is stopped and disabled, and the console login returns to the screen.
Start screen	The screen opened first — one of the project's screens (templates are not listed). (<i>default — main</i>) opens <code>main</code> .
Rotation	0°, 90°, 180° or 270° clockwise, for a panel mounted on its side. A touch panel may also need a touch calibration after rotating (README, <i>Touch is rotated</i>).
Hide cursor	Hides the mouse pointer — recommended for a touch-only panel.
Wait timeout (s)	5–600 s. How long the display waits for the screens server at start before showing <i>Waiting for the controller</i> ; it keeps trying and switches over as soon as the server answers.
Status on the controller	Whether the display is installed, enabled and running (<code>systemctl is-enabled / is-active zeus-kiosk</code>). Checked when the page opens with a controller connected, or with Check status .

The settings are stored in the project (`controller.json` , section `kiosk` ; not part of the runtime's configuration and not sent by **Download hardware**). **Apply...** saves them and writes them to the controller's `/etc/zeus-plc/kiosk.conf` — only the keys this page sets; `OUTPUT` and `EXTRA_FLAGS` , if a technician set them by hand, are kept — then enables and restarts the display, or turns it off. **Restart display...** restarts the browser, for example after **Download screens** or to end a log-in. Each asks first; none of them stops the control loop. On the **Simulator** target the section is unavailable: the local display is a controller feature, and the simulator's screens open in any browser (§32.8).

Logging in. With controller accounts, the display shows the panel's own log-in page (§32.6.3); on a touch panel the on-screen keypad opens on the **User** and **Password** fields, so no keyboard is needed. The log-in lasts as a panel session (`web.session_idle_min`) and is forgotten when the display restarts (reboot, power loss, **Restart display...**). Without accounts the screens open directly, writable if **Screens may write** is on.

Security

- The browser runs as `zeus-kiosk` , an account with no password, no shell and no `sudo` , and cannot read the runtime's configuration. It can open only `127.0.0.1 / localhost` ; developer tools, downloads, printing and file dialogs are disabled, and there is no desktop behind it. Closing it only makes the service start it again.
- Anyone at the panel can operate whatever the screens allow for the logged-in user. Use a dedicated **operator** or **viewer** account for the display rather than an engineer's, use **Access** (§32.4.16) and confirmations on what matters, and lock the cabinet — physical access to the controller (USB, SD card, Ethernet) is access to everything.

- The page talks to the controller over loopback only; the password never crosses the network from the display.

Troubleshooting

Symptom	Cause	Remedy
Install fails at the package step	No internet access on the controller, or no package source for the browser	Give the controller internet access for the installation; the Messages lines show apt's error.
<i>has no passwordless sudo</i>	The target's SSH user cannot run <code>sudo</code> without a password	Configure <code>sudo</code> for that account, or install by hand (README).
Black screen	KMS graphics off, display connected after boot, or the browser crashing	<code>journalctl -u zeus-kiosk -b</code> on the controller; README, <i>Black screen</i> .
<i>Waiting for the controller stays</i>	Screens server off, bound to one outside address, or wrong port	Messages warns about both after Apply... ; enable the server on <code>127.0.0.1 / 0.0.0.0</code> , Download hardware .
Touch is off by 90° after rotating	The touch coordinates are not rotated with the picture	Calibration matrix, README <i>Touch is rotated</i> .

33. Retain, power-fail and maintenance

This chapter covers what a controller keeps across a restart and a power cut, and the routine work of keeping a fielded controller healthy: retained variables and the retain store, SD-card wear, the power-fail save on the IPC carrier, CONFIG mode for hardware work, backup and restore, time synchronisation, and updating the controller's runtime. Read it before commissioning, and keep it at hand for maintenance visits.

33.1 What survives a restart

Data	Survives runtime restart / power cut	Notes
The program (boot program)	Yes	Stored on the controller at every successful download, with its signature, version and source map.
%M (the whole marker area)	Yes	Always retained.
Variables declared <code>RETAIN</code>	Yes	Program, global and function-block instance variables (§33.2).
Retained standard timers and counters	Their state (Q, CV, elapsed time)	The presets <code>PT</code> / <code>PV</code> come from the program text, never from the store.
All other variables	No	Start at their declared initial values.
%I	No	Read from the field at start.
%Q	No	Start at zero until the program writes them.
Forces	No	A restart releases every force.
Logins	No	Login is per connection; reconnecting starts anonymous.
Alarm journal, signature list in RAM	No	The audit trail on disk keeps operator actions and signatures (Chapters 27, 28).
Controller configuration	Yes	Changed only by Download hardware or Deploy .

33.2 RETAIN variables

33.2.1 Declaring retained data

Declare variables that must survive a power cut with `RETAIN` :

```
VAR_GLOBAL RETAIN
    batch_counter : DINT;          (* survives a power cut *)
    fill_setpoint : REAL := 500.0; (* initial value used only on a cold start *)
END_VAR
```

```

PROGRAM Totals
VAR RETAIN
    total_flow    : TOTALIZER;      (* its outputs are part of the retained image *)
    run_hours     : RUNTIME_HRS;
END_VAR
VAR
    flow_m3h      : REAL;
    pump_running  : BOOL;
    batch_done    : BOOL;
    done_edge     : R_TRIG;
END_VAR
total_flow(rate := flow_m3h, rate_base := 2, dt_ms := 10, run := pump_running);
run_hours(running := pump_running, dt_ms := 10);

done_edge(CLK := batch_done);
IF done_edge.Q THEN
    batch_counter := batch_counter + 1;
END_IF;
END_PROGRAM

```

A retained function-block instance retains all its data, including the standard timers and counters inside it. Retain only what the process needs after a restart: totals, counters, operating hours, setpoints entered by operators, recipe parameters. Do not retain sequence states or command bits blindly — a machine that restarts in the middle of a step it was in before a power cut, with the plant in a different state, is a hazard. Restart sequences deliberately from a known state.

Important: Keep `VAR_GLOBAL RETAIN` declarations in `globals.st`. Further global variable lists are concatenated in name order, so creating a list that sorts earlier moves every retained symbol after it; the controller then refuses the restore and resets every retained value (§33.3.3). The IDE warns about this on every save of a list that declares retained values (Chapter 19).

33.2.2 Initial values and cold starts

A retained variable takes its declared initial value only when there is nothing to restore: on the first start, after a refused restore, or after a cold start. A **cold start** discards the retained image on purpose; it is started from the controller's command line (`--cold`) by service staff, and is audited (`cold restart requested on the command line - retained data ...`).

A download does not touch retained values: online change carries every variable with the same name and type across (Chapter 20), retained or not.

33.3 The retain store

33.3.1 Where and how it is written

Setting	Config key	Default	IDE
Retain file	<code>process_image.retain_file</code>	<code>/var/lib/zeus-plc/retain.bin</code>	Controller Settings → Retain file
Periodic save interval	<code>process_image.retain_interval_ms</code>	30000 (30 s); 0 = save only on a clean stop	<code>controller.json</code>

Setting	Config key	Default	IDE
%M size	process_image.marker_bytes	4096	Controller Settings → %M marker bytes

The store is written:

- **periodically**, every `retain_interval_ms`, by a background thread — never by the scan;
- **on every clean stop** of the runtime (service stop, **Restart runtime...**, **Download hardware**, **Shut down controller...**, the power-fail save), as a final save that is audited (`retain: final image written ...`).

Each save writes a temporary file, flushes it to the card, renames it over the old file (atomic on the same filesystem) and flushes the directory. A reader therefore always finds either the complete old file or the complete new one. The file carries a CRC32 over its contents.

Note: Data written between the last periodic save and an **unclean** power loss is lost: up to `retain_interval_ms` of changes. That is why the power-fail save (§33.5) exists. Do not shorten the interval to a few seconds to compensate; see §33.4.

33.3.2 The layout signature

The store records a **layout signature**: a hash over the name, declared shape, type, storage and memory slot of every retained variable, the size of the program's data area, the size of `%M`, and the function-block types. At start, the stored image is restored only if its signature matches the program that is loaded.

The reason is the failure mode it prevents. Restoring by position into a program whose retained variables were reordered, retyped or resized would succeed byte for byte and silently put a batch counter into a totaliser — no error, wrong data. Matching on the signature makes that impossible.

33.3.3 When the restore is refused

Log/audit message	Cause	Result
<code>retain: restored N byte(s) from '...' (saved ..., save #n)</code>	Match.	Values restored.
<code>retain: no stored image (first run) - starting from initial values</code>	No file.	Initial values.
<code>retain: RESTORE REFUSED - the program's retained variables changed since this image was written</code>	The retained layout changed.	Retained data reset to initial values; audited.
<code>RESTORE REFUSED - the %M area size changed</code>	%M marker bytes changed.	As above.
<code>RESTORE REFUSED - the program's data area changed size</code>	The program's data layout changed.	As above.
<code>... written by a different runtime version</code>	Retain file format of an older runtime.	As above.
<code>retain: stored image REJECTED - CRC mismatch ... / file is truncated / bad magic</code>	File damaged.	Initial values; audited.

A refusal is deliberate and loud: an operator must know that retained process data was discarded.

When it happens in practice. After a download, the runtime's retain saver follows the new program at once and the next save writes the new layout. A refusal therefore occurs when the controller boots a program whose layout differs from the last saved image:

- the power was lost **after** a download that changed the program's data layout and **before** the next retain save;
- `%M` was resized, or retained globals were reordered (§33.2.1);
- the retain file was restored from a backup of a different program version.

To protect retained values across a program change:

1. Before the download, save a data snapshot (**Online** → **Save data snapshot...**, Chapter 23).
2. Download the program.
3. Wait at least one retain interval (default 30 s), or make a clean stop, before any power cycle.
4. After a restart, check the audit trail for the `retain:` line. If the restore was refused, restore the values that matter from the snapshot (**Online** → **Restore data snapshot...**).

33.4 SD-card wear

The IPC boots from an SD card. SD cards wear-level across a small pool of erase blocks; a card written every few seconds for years fails, typically by turning read-only in the middle of production, which looks like a software fault for the first hour of diagnosis.

What the runtime does:

- A retain save is **skipped** when no retained byte changed since the last save (the header's timestamp and counter are excluded from the comparison). Retained data changes when the process changes state, so on most plants most intervals write nothing. The log reports the ratio every hundred saves (`retain: N save(s), M skipped as unchanged`).
- The interval is configurable (`retain_interval_ms`). The default of 30 s is a good balance; do not go below about 10 s.

What the operating system does is handled by the script `scripts/harden-sdcard.sh`, run on the controller by service staff:

Command	Effect
<code>harden-sdcard.sh</code>	Report only; changes nothing.
<code>harden-sdcard.sh --apply</code>	Applies the safe measures (fewer OS writes). Printed and reversible.
<code>harden-sdcard.sh --apply --volatile-journal</code>	Also moves the system journal to RAM.

Important: The runtime log can be put on a RAM file system (`log.file`) and the audit trail on flash (`log.audit_file`). Do **not** move the audit trail into RAM, and do not use `--volatile-journal`, on a validated or security-assessed system unless the log is shipped off the controller first (`log.syslog_host`, Chapter 34): a volatile journal deletes the audit trail at every power cycle.

33.5 Power-fail save

33.5.1 The carrier

The IPC-CM400200 carrier has a UPS microcontroller and a supercapacitor. When the 9 V supply is removed, the board keeps running on the supercapacitor for a few seconds, and the UPS controller signals the loss on **GPIO25** with one clean, debounced falling edge (HIGH on mains, LOW on loss).

33.5.2 The power-fail monitor

The service `zeus-powerfail` watches GPIO25:

1. At start, while on mains, it checks that the pin reads the configured "power present" level. If it does not — wrong pin, wrong polarity, pin not set — it **refuses to arm** and logs why, rather than risk powering the controller off.
2. When armed it logs `armed: GPIO25 on gpiochip0, present=1, grace 5000ms` and waits for the loss edge in the kernel (edge-triggered, no polling, no CPU load, no scan jitter).
3. On loss it waits a **grace period** (default 5 s), re-checking the line. If power returns within it, the shutdown is cancelled (`power returned ... shutdown cancelled`) — a short dip rides through on the supercapacitor.
4. If power is still lost, it runs the emergency save: stop the runtime cleanly (the final retain save), flush the file systems, power off. The data is saved first, so even if the capacitor drains before the power-off completes, retained data is already on the card.

On the next power-up the runtime restores the saved image.

The configuration is `/opt/zeus-plc/config/powerfail.conf` :

Setting	Default	Meaning
<code>POWER_GPIO</code>	25	Power-fail line (BCM number).
<code>GPIO_CHIP</code>	<code>gpiochip0</code>	GPIO character device.
<code>POWER_PRESENT_LEVEL</code>	1	Level while mains is present.
<code>SHUTDOWN_DELAY_MS</code>	5000	Grace period. The supercapacitor must hold for this plus about 2–3 s for the save and power-off; measure the hold time before raising it.
<code>POLL_MS</code>	250	Re-check interval during the grace period.
<code>ACTION</code>	<code>/opt/zeus-plc/bin/zeus-emergency-save.sh</code>	What runs on a confirmed loss.

WARNING: Never read the carrier's GPIO lines with a rapid polling tool (for example a tight `pinctr1` loop) on this board. It disturbs the handshake between the UPS controller and the compute module and has caused an unattended shutdown and restart. The monitor uses edge events only.

33.5.3 Testing the power-fail save

To test, with the machine safe to stop:

1. Check that the monitor is armed: the system log shows `zeus-powerfail: armed ...`.
2. Remove the 9 V supply and keep it removed.

- 3. After about 5 s the log shows `EXTERNAL POWER STILL LOST ... emergency save + poweroff` ; the runtime stops and the controller powers off.
- 4. Restore power. The controller boots; the audit trail shows `retain: restored ...` .
- 5. Repeat with power restored within 5 s: the log shows `power returned ... shutdown cancelled` and the controller keeps running.

33.5.4 Clean shutdown from the IDE

Online → Controller → Shut down controller... (engineer, real controller only) saves retained data and halts the controller. It stays off until somebody restores power at the machine. Use it before disconnecting power for work on the panel. The carrier has no hardware shutdown button.

WARNING: Shut down stops the control loop and drops all outputs, and the controller does not come back by itself.

33.6 CONFIG mode

CONFIG mode halts the program for hardware work while inputs continue to be read, so channels can be watched while they are wired.

	RUNNING	CONFIG
Program	Executes	Halted
Inputs %I	Read	Read
Outputs	Driven by the program	No longer written; each fieldbus device's watchdog releases them to its fail-safe state
Leaving	—	Goes to PAUSED , never directly to RUNNING

To enter CONFIG mode (engineer):

- 1. Make the machine safe for hardware work.
- 2. Choose **Online → Controller → Config mode** (or the  rail button).
- 3. Confirm the dialog. If the machine is running it warns *THE MACHINE IS RUNNING*.

CONFIG can be entered from any live state, including **FAULT**: a fault is often why somebody is about to work on the hardware.

To leave: choose **Config mode** again and confirm. The controller goes to **PAUSED**, because after hardware work the program's output image may describe a machine that no longer exists. Check the IO in the online view, then press **Run**.

Entering and leaving are audited (`scan engine entered CONFIG at scan N`).

WARNING: CONFIG mode is not an isolation. Outputs fall to their fail-safe state through the fieldbus watchdogs, but the controller is still powered and can be put back into RUN from the IDE. Follow your site's lock-out procedure for work on field wiring.

33.7 Backup and restore

33.7.1 What to back up

Item	Where	How
The project (program, configuration, screens, library, lock)	Engineering PC	Project → Create archive... (Chapter 5); version control.
Automatic project copies	backups/ in the project	Written by the IDE before each download. Not included in archives.
Controller configuration	/opt/zeus-plc/config/active.json	Controller backup (below). Contains the monitor token or user hashes, the signing public key.
Boot program with signature, version, source map	/var/lib/zeus-plc/boot.st (+ .sig, .ver, .manifest)	Controller backup.
Retained data	/var/lib/zeus-plc/retain.bin	Controller backup; values also via Save data snapshot...
Audit trail	/var/lib/zeus-plc/audit.log	Controller backup; ship it continuously (Chapter 34).
OPC UA server certificate and key	as configured, e.g. /var/lib/zeus-plc/opcua-server-*.der	Controller backup.

Not backed up, by design: SSH host keys, `machine-id` and the hardware root of trust — per-device identities that must never be duplicated.

33.7.2 Controller backup

Service staff take a controller backup from a PC with SSH access to the controller:

```
bash scripts/zeus-backup.sh <user>@<controller-ip> ./backups ~/.ssh/id_rsa
```

It is read-only on the controller and produces `backups/zeus-backup-<host>-<UTC>.tar.gz` with the configuration, the state files, an identity record (host, `machine-id`, SSH host-key fingerprint, runtime binary SHA-256) and a `MANIFEST.sha256`. Verify an archive with `sha256sum -c MANIFEST.sha256` after unpacking.

Important: The backup contains secrets (monitor token or password hashes, the OPC UA private key). It is written with owner-only permissions; store it in an access-controlled, preferably encrypted location, not on a shared drive.

Take a backup before every key rotation, before switching program verification to `enforce`, before a runtime update, and on a schedule that matches how often the program and retained data change.

33.7.3 Restore — same unit only

```
bash scripts/zeus-restore.sh backups/zeus-backup-<...>.tar.gz <user>@<controller-ip> ~/.ssh/id_rsa
```

The script verifies the manifest, warns if the target's `machine-id` differs from the archive, asks you to type `YES` , stops the runtime, restores configuration, program, retained data and OPC UA identity with the correct ownership, and restarts. Then confirm: the program signature verifies, the scan runs, the `clock: audit` line is present.

The audit log is never overwritten: if the controller already has one, the archived copy is placed beside it as `audit.from-backup.<UTC>.log` .

Important: Do not restore a backup onto a **different** controller. It carries that unit's credentials, and two controllers sharing a token is a security defect. To replace a controller, install a fresh image, **Deploy** the runtime (which creates new credentials) and download the program.

33.8 Time synchronisation

33.8.1 Why it matters

Every audit record, alarm and signature is stamped with the controller's clock and with a flag saying whether that clock was synchronised. A Compute Module has no battery-backed clock of its own; a controller on an isolated network with no time source free-runs from whatever time it booted with and drifts — months behind was found on a bench unit. For a GxP system that is a defect: the records carry wrong times.

At start, and whenever the state changes, the runtime writes one audit line describing the clock:

Audit line	Meaning
<code>clock: system time SYNCHRONIZED to a time source - audit timestamps are trustworthy</code>	NTP-disciplined.
<code>clock: system time NOT NTP-disciplined, but a hardware RTC (<chip>) is present ...</code>	Held across reboots by a real-time clock, not disciplined.
<code>clock: system time NOT synchronized - no NTP source and no hardware RTC; audit timestamps are PROVISIONAL/free-running</code>	Neither. Records before the first <code>SYNCHRONIZED</code> line are provisional.
<code>... SYNCHRONIZED ... (recovered)</code>	The moment synchronisation was achieved; explains the step in the timestamps.

The runtime observes the clock; it never sets it.

33.8.2 What the site must provide

Configure one internal, controlled time source — typically the SCADA server, a domain controller or an NTP appliance on the management network. Do not point a controller at the public internet.

On the controller (service staff), in `/etc/systemd/timesyncd.conf.d/zeus-site.conf` :

```
[Time]
NTP=<site NTP server IP>
FallbackNTP=
```

then enable it with `sudo systemctl enable --now systemd-timesyncd` and `sudo timedatectl set-ntp true` , and check that `timedatectl` reports `System clock synchronized: yes` .

Note: If the source answers but the controller still does not synchronise and the time service logs *Server has too large root distance*, the server declares a poor accuracy (typical of a Windows time server). Adding `RootDistanceMaxSec=30` to the file above accepts it. That is a recorded decision to trust a source whose accuracy claim is poor; a stratum-1 appliance, a GPS clock or a Linux `chrony` server does not need it.

A battery-backed RTC (for example a DS3231 on I²C) holds the time across power cycles and network isolation. With one fitted and seeded, the controller comes up with the correct time with no network, and the audit line reports the RTC.

33.8.3 Set clock / time... in the IDE

Online → **Controller** → **Set clock / time...** (engineer) shows the controller's time, whether NTP is synchronised and whether a hardware RTC is present, and offers:

Button	Effect
Sync to this PC	Sets the controller clock to this PC's time.
Set by hand	Sets the time entered in Set by hand (UTC) .

Setting the clock turns NTP off on the controller (a time server would otherwise overwrite it) and writes the hardware RTC when one is present; without an RTC the setting is lost at power-off. Use it to correct a bench unit or a controller with no network; for a plant, configure NTP.

Audit files and syslog records are written in UTC regardless of the controller's time zone.

33.9 Restart, reboot and versions

Command (Online → Controller)	Effect	Role
Versions...	IDE and runtime versions.	any
Controller log...	The runtime's log without an SSH session.	viewer
Restart runtime...	Stops the control loop briefly, reconnects drivers, rebuilds the OPC UA address space, reloads and runs the boot program. On a controller it restarts the service over SSH.	engineer
Reboot controller (OS)...	Reboots the operating system; controller offline for about 30–60 s.	engineer
Shut down controller...	Saves retained data and powers off; does not come back by itself.	engineer

Reboot and shut down need SSH access and password-less `sudo` for the SSH user on the controller; otherwise they report that the controller cannot do it without a password.

WARNING: All three stop the control loop. Each asks for confirmation. Use them only when the machine is safe to interrupt.

33.10 Updating the controller runtime

Updating the runtime is a **Deploy**, a commissioning act separate from a program download. It installs the runtime binary, the configuration and the system service, and restarts the control loop.

To update the runtime:

1. Back up the controller (§33.7.2) and archive the project.
2. Make the machine safe to stop.
3. Choose **Target...** in the status bar. In the controller's row, click **Deploy runtime...**
4. Read the step report:

Step	Checks / does
probe	Reaches the controller over SSH (key authentication only), reads architecture, kernel and board. Refuses a non-aarch64 target.
board check	Warns if the declared type (IPC-CM400200 / IPC-CM500200) does not match the board.
sudo	Password-less <code>sudo</code> is required for capabilities and the service.
service account	The runtime runs as a dedicated system account without a shell.
directories, copy binary	Installs the new binary.
binary integrity	Compares the local and remote SHA-256; on any mismatch the deploy aborts and the running binary is left untouched. Writes a signed manifest beside the binary.
copy config, config permissions	Installs the configuration, readable only by the runtime.
install service, install + capabilities	Installs the service with its sandbox profile and grants only the needed capabilities.
start service	Starts the runtime and waits until it listens.

5. Reconnect, log in, and check **Versions...**, the run state and the audit trail (`signature` `VERIFIED` under program verification, the `clock:` line, the `retain:` line).

Important: A Deploy writes the controller configuration from the project and resets program verification to the project's value (default `warn`). If the controller was running with `enforce`, set it again (Chapter 34).

Keep the IDE and the runtime at the same version. A runtime older than the IDE reports missing features (for example *Connection time not reported by this runtime*); a newer IDE compiler may accept programs the older runtime refuses.

33.11 Maintenance checklist

Interval	Task
Every visit	Check the run state, forces (<code>FORCE</code> badge), overruns and jitter, the IO lamp; read the audit trail since the last visit.
Every visit	Check <code>timedatectl</code> / the <code>clock:</code> audit line: synchronised.

Interval	Task
After every program change	Wait for a retain save; check the <code>retain:</code> line after the next restart.
Monthly or per campaign	Controller backup; project archive.
Yearly	Test the power-fail save (§33.5.3); run <code>hardening-sdcard.sh</code> in report mode; review users and remove leavers.
Before a runtime update	Backup, archive, safe machine state.

34. Security

This chapter is a working summary of the controller's security controls for the engineer who commissions and maintains it: user accounts and roles, passwords and lockout, machine tokens, transport protection (SSH tunnel and TLS), program signing, the audit trail, the host firewall and process hardening. It states what each control protects against and what it does not. The full security architecture, threat model and requirement set are in documents 09, 10, 20 and 21 of the `docs` folder; this chapter tells you what to configure.

WARNING: The IDE can start, stop, force and download to a running machine from a desk, possibly in another room. Whoever can log in as an operator or engineer can move the machine. Treat controller credentials as you would the key to the control cabinet.

34.1 What is being protected

In order of consequence:

1. **The machine.** Outputs drive actuators; the worst outcome is motion nobody commanded.
2. **The control program.** A modified program is a machine that behaves wrongly and looks healthy.
3. **The process data.** Reading it may be commercially sensitive; writing it is item 1.
4. **The audit trail.** Without it none of the above can be shown to have happened or not.

Boundary	Protected by	Not protected against
IDE → controller (monitor protocol, TCP 5020)	Accounts or token, per-address lockout, loopback bind + SSH tunnel, optional TLS	A compromised engineering PC that holds valid credentials
IDE → controller (deploy, backup, reboot)	SSH with key authentication only	Anyone holding the SSH key
Controller → fieldbus (EtherCAT, Profinet, Modbus, EtherNet/IP scanner)	Physical and topological separation: a dedicated NIC and segment	Anyone on the fieldbus segment — the protocols have no authentication
Servers (OPC UA, EtherNet/IP tags, Modbus slave, screens)	Per-server settings, §34.8	Unauthenticated protocols (Modbus, EtherNet/IP) are as open as their network
Physical access	Panel lock	Anyone with the SD card: no disk encryption in this version

The IDE's own web server listens on the engineering PC's loopback address only and has no authentication of its own. Do not expose it to the network.

34.2 Users and roles

34.2.1 Roles

Every command a client sends to the controller is checked against the role of the logged-in user.

Role	May	May not
viewer	Read values, watch, browse symbols and IO, read the log, see forces, see alarms and signatures, read a trace, verify, upload, sign (with own password)	Change anything
operator	Everything a viewer may, plus write values, force and unforce, run, pause, resume, reset fault, reset statistics, acknowledge and shelve alarms; press buttons on operator screens	Download, scan a fieldbus, CONFIG mode, breakpoints, arm a trace, take alarms out of service
engineer	Everything, including download, scanning fieldbuses, Profinet commissioning, CONFIG mode, breakpoints and stepping, arming a trace, alarm out-of-service; restart, reboot, shut down, deploy (these also need SSH)	—

Two deliberate choices: seeing what is **forced** is a viewer right, because the person who most needs to know is often the one with the least authority to change it; and any command the controller does not recognise defaults to the engineer role.

34.2.2 Managing accounts

Accounts are part of the project (`controller.json` , `monitor.users`) and reach the controller with **Download hardware**. The same accounts are used by the IDE and by the operator screens (Chapter 32).

To add or change a user:

1. Choose **Online** → **Users & Roles...** (or the **Users & Roles** node in the device tree).
2. Under **Add or replace**, enter **User** (2–31 characters: letters, digits, dot, dash, underscore), **Password**, **Confirm** and **Role**.
3. Click **Save user**. A user that already exists is overwritten.
4. Choose **Online** → **Download & Upload** → **Download hardware** to put the accounts into effect.

WARNING: A hardware download restarts the runtime and stops the control loop briefly.

Until the download, the controller keeps running with the accounts it had. A controller you believe is protected and is not is worse than one you know is open.

To remove a user, click **Remove** on its row, confirm, and **Download hardware**. The IDE refuses to remove the last **engineer** while other accounts exist, because nobody could then download or add users.

Limit	Value
Accounts per controller	16
User name	2–31 characters [A-Za-z0-9._-]
Password	at least 8 characters; not <code>admin</code> , <code>password</code> , <code>12345678</code> , <code>zeus</code> , nor the user name

34.2.3 Password storage

The IDE derives a salted PBKDF2-SHA256 hash (200,000 iterations, 16-byte random salt) and stores only the salt and hash. The controller only verifies; a password is never stored, logged or shown, and cannot be read back — only replaced.

34.2.4 The default account

A new project contains one account, `admin / admin`, role engineer, marked as **still on its initial password**. It exists so that commissioning has a way in. For the same reason, **Users & Roles...** on a project with no accounts offers **Create the default admin account**.

Because the password is documented, it is not a password:

- the IDE asks for a new password the first time you log in with it;
- the controller writes an audit line at every start and every login while it is unchanged (`user "admin" still has its initial password ...`);
- an account on its initial password **cannot sign** electronically (Chapter 28);
- once a controller has accounts, the IDE refuses a download that would carry an account still on its initial password to it. Set a new password first.

34.2.5 Logging in

Online → **Log in...** logs in to the connected controller; **Log out** drops the session. The session badge in the status bar shows the user and role.

Login is **per connection**. When the connection drops — a controller restart, a hardware download, a network interruption — the new connection starts anonymous and you must log in again. The IDE carries the login across the restarts it performs itself (hardware download, restart runtime) where it can.

A controller with no accounts answers every connection; login is not possible and not needed. That is acceptable on a bench and wrong in a plant.

34.3 Lockout

Failed logins and failed token authentications are counted **per source address**. After **5** failures from one address, that address is refused for **60 seconds**; the lockout survives reconnection. A wrong user name and a wrong password give the same answer (*the user name or password is not correct*) and take the same time, so the account list cannot be discovered by trying names. Every failure and every lockout is audited with the address.

34.4 Machine tokens

A token authenticates a **program** — the IDE, a SCADA connector — rather than a person. It does not expire and nobody types it.

Configuration	Behaviour
<code>monitor.token</code> + <code>require_token</code>	One shared token; treated as an engineer named <code>default</code> .
<code>monitor.identities[]</code>	Up to 8 named tokens, each with its own role: <code>{ "name": "scada1", "token": "...", "role": "viewer" }</code> .
Monitor bound to a non-loopback address	A token of at least 16 characters is mandatory; without it the runtime refuses to start .

Accounts and a shared token are alternatives, not layers: the IDE attaches its token to every request, so a controller that had both would authenticate the IDE silently as the token's identity. **Deploy** therefore removes the token

when the project has accounts with at least one engineer, and generates a random 48-hex-character token when it is needed (non-loopback bind, no usable accounts), storing it in the target entry of the IDE.

Rotate tokens on staff change or suspected leak: change `monitor.token` in the controller configuration and in the IDE's target, restart both.

34.5 Protecting the connection

The monitor protocol is line-based JSON over TCP. **Without further measures it is not encrypted:** on a routable address the token and all process data cross the network in clear. The runtime says so in its audit log at every start when the monitor is bound to a routable address. There are two ways to protect it.

34.5.1 SSH tunnel (recommended)

The monitor binds to `127.0.0.1` on the controller, and the IDE reaches it through an SSH tunnel it opens and supervises itself. This gives modern transport security with no extra software in the process that drives the machine.

To configure a hardened target:

1. Make sure your SSH key is installed on the controller (`ssh-copy-id <user>@<controller>`); SSH uses key authentication only.
2. In the IDE's target file (`.zeus-ide-targets.json` beside the IDE), set `"tunnel": true` for the controller's entry, for example `{ "id": "line1", "host": "10.10.1.20", "user": "zeus", "port": 5020, "tunnel": true }`.
3. **Deploy** (Chapter 33). The deployer writes `"bind": "127.0.0.1"` into the controller's configuration, and the IDE connects through a local forwarded port from then on.

Set the flag and deploy together. With `tunnel: true` but no deploy, the IDE forwards to a loopback port on a controller whose monitor still listens only on its LAN address, and the connection is refused. Deploy itself always uses SSH directly, so it cannot lock you out.

Authentication stays required under a loopback bind: "reachable only from this host" is not the same as "reachable only by the engineer".

The same pattern protects the screens server (Chapter 32) and any other loopback-bound service: `ssh -L 8080:127.0.0.1:8080 <user>@<controller>`.

34.5.2 TLS

The monitor can also run TLS itself (TLS 1.2 or later):

```
"monitor": { "tls": true, "tls_cert": "/opt/zeus-plc/config/monitor.crt",
             "tls_key": "/opt/zeus-plc/config/monitor.key" }
```

Without a certificate the runtime generates a self-signed one and logs that it encrypts but does not prove the controller's identity. If TLS is enabled and cannot be initialised, the monitor does not start. The IDE uses TLS for a target whose entry has `"tls": true`. The Windows simulator does not support TLS.

34.6 Program signing

34.6.1 How it works

The IDE holds a site **Ed25519** signing key, generated on first use and stored beside the IDE (`.zeus-ide-signing.key` , owner-only permissions and an owner-only ACL on Windows). Every program the IDE downloads is signed with it, together with a monotonically increasing **version** number. The controller holds only the public key and verifies.

<code>security.program_verify</code>	Unsigned or badly signed program	Program with a version below the stored one
<code>off</code>	Runs	Runs
<code>warn</code> (set by Deploy on first provisioning)	Runs; audited	Audited
<code>enforce</code>	Refused	Refused (anti-rollback)

The signature is stored with the boot program and verified again at every start. The key's version counter (`.zeus-ide-signing.key.counter`) is the rollback floor: never delete it.

Deploy provisions `security.program_key` from the IDE's public key and sets `program_verify` to the project's value, `warn` if none is given. It never raises it to `enforce` by itself.

34.6.2 Moving to enforce

1. Confirm that every controller of the site logs `signature VERIFIED (vN)` after a download under `warn` .
2. Back up each controller (Chapter 33).
3. Set `"security": { "program_verify": "enforce" }` in the project and **Download hardware**.
4. Download the program again and confirm it runs and verifies.
5. After any later **Deploy**, check that `enforce` is still set.

Note: The Windows simulator cannot verify signatures and treats every program as unsigned. A simulator configured with `enforce` refuses every program — keep the simulator's project at `off` or `warn` .

34.6.3 Key rotation

Rotate the signing key if it may have been exposed or when staff with access to the engineering PC change. Keep the old key until every controller has migrated and keep the counter file in place, then per controller: set `warn` , provision the new public key, re-download (the version continues above the previous floor), set `enforce` again. The full procedure is in document 19.

34.7 The audit trail

Every action that changes the machine or its access is written to the controller's audit trail, with the user name (or token identity) and the peer address: logins and failed logins, lockouts, writes, forces, run-state changes, downloads (with online-change result), uploads, trace arming, breakpoints, alarm acknowledgements and shelving, signatures and refused signatures, writes from OPC UA, the Modbus slave and the operator screens, retain restores and refusals, clock-synchronisation changes, and the SHA-256 of the configuration at start.

Item	Configuration ("Log")
Runtime log	<code>file</code> (may be on RAM, \$33.4)
Audit trail file	<code>audit_file</code> — on flash; fsynced
Remote shipping	<code>syslog_host</code> , <code>syslog_port</code> (default 514), <code>syslog_level</code> (records at or above it are shipped; audit records are always shipped)

Read it with **Online** → **Controller** → **Controller log...**, or **HMI** → **Events** with **audit only** ticked. Audit records are stamped in UTC.

Important: The audit trail on the controller is lost with the SD card. For a regulated or security-assessed system, ship it continuously to a collector the controller cannot rewrite (`log.syslog_host`).

Not audited in this version: writes through the EtherNet/IP tag server (Chapter 30). Keep that server read-only where a complete trail is required.

34.8 Services and their exposure

Service	Default	Authentication	Encryption	Recommendation
Monitor (5020)	on, loopback	accounts / token	SSH tunnel or TLS	Loopback + tunnel.
OPC UA (4840)	off	anonymous (off) / users	yes (security policy)	Users + security policy; see Chapter 29 for writes.
Screens server (8080)	off, loopback	accounts (HTTP Basic)	no	Loopback + tunnel, or a dedicated segment.
Modbus slave (5502)	off, loopback, read-only	none	no	Read-only; bind to the OT address.
EtherNet/IP tags (44818)	off, read-only	none	no	Read-only; bind to the HMI network address.
SSH (22)	on	keys	yes	Keys only; no password or root login.

The **Network Interfaces** page of each port lists the services listening on it and highlights those bound to every interface, allowing writes or allowing anonymous access.

34.9 Host firewall

The controller is usually dual-homed: one port on the OT/fieldbus network, one on the management/SCADA network. A host firewall enforces that separation and prevents a later misconfiguration from silently re-exposing a service. The script `scripts/zeus-firewall.sh` (run on the controller by service staff) applies a default-deny rule set:

Port	Allowed on
TCP 5020 (monitor)	loopback only (reached through SSH)

Port	Allowed on
TCP 22 (SSH)	management network
TCP 4840 (OPC UA)	management network (tighten to the SCADA server)
UDP 34964 (Profinet CM)	OT port
Profinet DCP/RT, EtherCAT	Layer 2 — not affected by the IP firewall

Add rules for the Modbus slave, EtherNet/IP tags or screens server if you enable them.

To apply it without locking yourself out:

1. `sudo bash scripts/zeus-firewall.sh minimize` — disables unused services first.
2. `sudo bash scripts/zeus-firewall.sh apply` — loads the rules and arms an automatic rollback after 180 s.
3. From your PC, open a **new** SSH session and check that the IDE (through the tunnel) and OPC UA still work.
4. Only then: `sudo bash scripts/zeus-firewall.sh commit` — cancels the rollback and makes the rules permanent.

If step 3 fails, do nothing; the rules revert by themselves. `rollback` reverts at once; `status` shows the rules and the timer. Interface names and networks are set with `MGMT_IF`, `MGMT_NET`, `OT_IF` and related variables. Do it with someone near the controller the first time.

The controller does not route between its two networks (`ip_forward = 0`); keep it that way.

34.10 Process hardening

The runtime runs as a dedicated system account without a login shell, under a restrictive systemd profile installed by **Deploy**:

- only the capabilities it needs: real-time scheduling, memory locking, and raw sockets only when the binary contains EtherCAT;
- a read-only system, no new privileges, no writable-and-executable memory, restricted address families and namespaces, a system-call allow-list, private `/proc` view, owner-only file creation;
- the runtime cannot rewrite its own binary or configuration (the install directory belongs to the deploying user; the configuration is readable only by the runtime).

Some of this cannot be tightened further without breaking a real need of a real-time fieldbus controller — real-time priority, locked memory, raw network access, access to hardware devices — and these are documented as accepted residual risks in document 21.

34.11 Known limits

Stated so they are not mistaken for oversights:

- No secure boot and no disk encryption in this version: physical access to the controller or its SD card is full access. Lock the panel.
- A compromised engineering PC holding the SSH key, the signing key or an engineer password can do anything an engineer can. Protect the engineering workstation.
- Fieldbus protocols have no authentication; the fieldbus segment must carry nothing else.

- Modbus TCP and EtherNet/IP have no authentication; writes through the EtherNet/IP tag server are not audited.
- The OPC UA server accepts any client certificate; client authentication rests on users and passwords.

34.12 Hardening checklist for a production panel

1. Every person has their own account; no shared logins; no account on its initial password.
2. Monitor bound to `127.0.0.1`, reached through the SSH tunnel (`tunnel: true` , deployed).
3. SSH: key authentication only, no password login, no root login.
4. `program_verify` set to `enforce` once all downloads verify.
5. OPC UA with users and a security policy; anonymous off.
6. Modbus slave and EtherNet/IP tag server off, or read-only and bound to the OT address.
7. Screens server on loopback, or on a dedicated segment with accounts.
8. Host firewall applied and committed.
9. Audit trail shipped to a collector (`log.syslog_host`).
10. Controller clock synchronised to the site time source (Chapter 33).
11. A dedicated NIC and segment for the fieldbus.
12. Record the runtime binary's SHA-256 (in the deploy manifest) at commissioning and check it after site visits.

35. Troubleshooting

This chapter collects symptoms seen in the field, their causes and remedies, grouped by area: connection and login, build and download, run state and performance, IO and fieldbuses, the integration servers, operator screens, alarms and batch, retained data and power, and the simulator. Start with §35.1, which lists the places the controller tells you what is wrong — most problems are diagnosed by reading one message there.

35.1 Where to look first

Source	How to open it	What it tells you
Messages pane	Bottom of the IDE window	Build errors (click to jump to the line), download reports, online-change results, step reports of deploy and simulator start.
Controller log	Online → Controller → Controller log...	The runtime's own log: driver states, server start lines, configuration problems, retain restore, clock state.
Audit trail	HMI → Events , tick audit only	Who did what, from where: logins, writes, forces, downloads, lockouts.
Status bar	Top right	Target, run state, session, <code>FORCE</code> , <code>IN SYNC</code> / <code>PROGRAM CHANGED</code> , connection dot.
Device pages	Device tree → IO device	State (<code>OPERATIONAL</code> , <code>DEGRADED</code> , <code>FAULT</code> , ...), reconnects, up-time, cycle time, data age (Chapter 25).
Panel lamps	On the controller	RUN: solid = running, slow blink = paused/config, fast blink = running with forces, off = stopped/faulted. IO: solid = all devices exchanging, blink = a device not exchanging.
Validate	Controller Settings → Validate with the runtime	The runtime's verdict on the project's <code>controller.json</code> , before it is sent.

Tip: Read the runtime's own sentence before changing anything. Messages name the setting, the device and the consequence; they were written to be the diagnosis.

35.2 Connection and login

Symptom	Cause	Remedy
Connection dot red, <i>Controller not connected</i>	Wrong host/port in Target... ; runtime not running; network or tunnel down	Check Target... ; ping the controller; check the service (<code>systemctl status zeus-plc</code> over SSH).
Connection refused from a controller that answers ping	Monitor bound to <code>127.0.0.1</code> and the target is not set to tunnel; or tunnel flag set without a deploy	Use the SSH tunnel (Chapter 34, §34.5.1): set <code>tunnel: true</code> and deploy.

Symptom	Cause	Remedy
Runtime refuses to start: <i>monitor.bind</i> is <code>"..."</code> (not loopback) but no token is set	Routable monitor bind without a token	Deploy (it generates one) or bind <code>127.0.0.1</code> and tunnel.
<i>monitor.token</i> must be at least 16 characters	Token too short	Use a longer token.
Live values blank; Monitor switches itself off	Not logged in; the controller refuses to be read by nobody when it has accounts	Online → Log in... , then turn Monitor on again.
Logged out after a restart, download or network drop	Login is per connection	Log in again.
<i>the user name or password is not correct</i>	Wrong credential, or account not yet on the controller	Retype; check Users & Roles... ; accounts reach the controller only with Download hardware .
<i>this controller has no accounts configured, so there is nothing to log in to</i>	Accounts exist in the project but were never downloaded	Download hardware , then log in.
Refused for 60 s after several attempts	Lockout after 5 failures from your address	Wait 60 s; find out who else is trying from that address (audit trail).
IDE insists on a new password at login	Account still on its initial password (admin / admin)	Set a new password; Download hardware .
Download refused: <i>"admin" is still on the password the account was created with</i>	Seeded account would be sent to a protected controller	Set a new password first.
<i>...needs a login</i> on an action	Role too low	Log in as a user with the required role (Chapter 34, §34.2.1).
Type mismatch error on connect (CM4 vs CM5)	Target type does not match the board	Correct the type in Target... — on a plant floor this mismatch is the moment before a program goes to the wrong line.
TLS target never connects	Controller monitor not configured for TLS, or vice versa	Match <code>"tls"</code> in the target with <code>monitor.tls</code> on the controller.

35.3 Build and download

Symptom	Cause	Remedy
<code>PROGRAM CHANGED</code> after editing	The program open here differs from the running one	Download it, or Upload to see what runs. It is the normal state while editing.
Download refused: the controller cannot compile the program	Program compiles in the IDE against a newer compiler than the controller's runtime	Keep IDE and runtime at the same version; deploy the runtime (Chapter 33).
'X' is not declared for another program's variable	Programs cannot read each other's local variables	Declare the variable in <code>globals.st</code> (Chapter 19).

Symptom	Cause	Remedy
A variable shows a value the program disagrees with	A local declared with the same name as a global: the program uses the local, the displays resolve to the global	The build warns; Where used (Shift+F12) lists both declarations first; rename one.
A timer never runs though its contacts read TRUE	Same cause (shadowing)	As above.
<i>not in static storage</i> when writing a variable	Name collides with a library block member (<code>start</code> , <code>stop</code> , <code>reset</code> , <code>enable</code> ...) or the declaring program is not the one running	Use a distinct name; download the right program.
Unknown type <code>PID</code> or another library block	The block is not in the project's <code>lib/</code> , or the library shrank	Library → Versions... : REMOVED shows it; Add to this project for missing default files.
Ladder declaration change undone	The change was made in the text, not in the ladder's declarations pane; the ladder header rewrites declarations when the diagram is saved	Edit declarations in the Ladder view (Chapter 16).
<i>not a whole multiple of the task period</i>	A program period must divide the task period	Use one of the two periods the message offers.
Program refused under <code>enforce</code>	Unsigned (IDE without key), signed by another key, or version below the stored one	Check the signing key and counter (Chapter 34, §34.6).
Online-change report shows many variables <i>reset</i>	Types changed, or variables renamed	Expected for changed declarations; restore values from a data snapshot if needed.
Hardware download stops the machine	By design: the runtime restarts to apply the configuration	Plan it; use Download program for program-only changes.

35.4 Run state, faults and performance

Symptom	Cause	Remedy
Controller in <code>FAULT</code>	Major fault (range error, stack fault, instruction budget), overrun limit, or a critical device lost	Read the controller log for the reason; fix; Reset fault .
Arithmetic wrong but machine keeps running	Minor fault (division by zero gives a defined result)	Check the minor-fault count and first message in the controller status (Chapter 21).
<code>FAULT</code> with <i>instruction budget</i>	Endless or very long loop in one scan (budget 200,000 instructions per scan)	Bound the loop; split work across scans.
Overruns increasing, jitter high	Scan period too short for the program; controller not prepared for real time; heavy service load on the scan core	Lengthen the period, stagger slow programs with offsets (Chapter 18), pin services to other cores; check <code>zeus-plcd --board</code> for <code>PREEMPT_RT</code> , isolated core and <code>performance</code> governor.
Jitter figures bad on a bench	Not a real-time-prepared controller	Do not quote them; measure on a prepared controller.

Symptom	Cause	Remedy
IO read jitter on one port	Fieldbus on a USB network adapter	Put fieldbuses on the native port.
In PAUSED an input moves but nothing else	By design: IO keeps running in PAUSED	Press Run .
After CONFIG mode the machine does not run	Leaving CONFIG goes to PAUSED	Check IO, press Run .
FORCE badge lit / RUN lamp blinks fast	Points are forced	Debug → Force / write table... ; read the list before Clear forces .
A breakpoint froze the machine	Breakpoints halt mid-scan	Continue (F5); use breakpoints only on a machine safe to freeze.

35.5 IO and fieldbuses

35.5.1 General

Symptom	Cause	Remedy
IO lamp blinks, wiring looks correct	A configured device is not exchanging	Find the device not OPERATIONAL ; inhibit it if not installed yet (Chapter 7).
Controller faults when one card is missing	Device marked critical (or <code>module_policy = fault</code> with critical)	Intended. Clear the fault when the device is back.
Device DEGRADED	A module missing or different from the configuration, <code>module_policy = degrade</code>	Compare the rack with the configuration; the log names the difference.
Badge shows SIM IO though hardware is fitted	Device configured with the simulation driver	Switch to the real driver; Download hardware .
Analogue channel reads counts, not engineering units	No signal range set on that channel	Set the range on the device page; Download hardware (Chapter 11).
Analogue value plausible but wrong	Wrong range, or wrong raw full scale	Check the range and raw limits against the card's manual (raw full scale 27700 on the reference cards).
Outputs go to zero on STOP/FAULT , or hold	Per-device output policy	Set the device's output behaviour on stop and on fault (clear or hold last) on the device page (Chapter 21).
Address of downstream cards changed	A card was inserted before them	Check addresses after any rack change; aliases follow the card.
Scan finds nothing on the simulator	The simulator has no fieldbus	Scan from a controller on the real segment.

35.5.2 Modbus TCP devices

Symptom	Cause	Remedy
Device never connects	Wrong IP/port/unit id; coupler on another subnet	Ping; check the device page; keep OT and plant ports on different subnets.
Values shifted by 8 bits or one register	Offset or count wrong in the device's register map	Compare with the coupler's register table (Chapter 8).
Device reconnects every few seconds	Coupler watchdog or TCP limit; cable	Check reconnects and up-time on the device page; check the coupler's own watchdog setting.

35.5.3 EtherCAT

Symptom	Cause	Remedy
<i>cannot open eth1 / raw socket error</i>	Binary lacks the raw-socket capability, or wrong interface name	Deploy (grants it when the binary has EtherCAT); check <code>ifname</code> .
No slaves answer	Cable in the wrong port (must enter IN of the first slave); segment not powered; NIC down	Check cabling and power; <code>ip link</code> .
Stuck below OPERATIONAL	Slave refuses the configuration; the log prints the slave, its state and the decoded AL status	Read that line — it is the diagnosis.
<i>slave refused the module list (0xF030:00) / refused PDO assignment (0x1C12/0x1C13) on the modular coupler</i>	The coupler does not accept these objects	Expected on this coupler; the process image comes from its ESI (<code>override_*_bytes</code>) — keep the ESI-derived configuration (Chapter 9).
<i>refused required start-up SDO</i>	A <code>startup_sdo</code> from the ESI or the configuration is rejected	Check the object and value; mark it optional if it is not essential.
<i>working counter N < expected M - segment degraded</i>	A slave not exchanging	Check the slave and cable; on a stable segment, a cycle too short for the controller.
<i>SYNC mode is not supported / only one EtherCAT master instance / cycle below 125 µs</i>	Refused configurations	Use async mode, one master, a longer cycle.
<i>unknown driver "ethercat" on the simulator</i>	Simulator binary without EtherCAT	The IDE simulator replaces fieldbus devices with the sim driver; start it from the Simulator node.

35.5.4 Profinet

Symptom	Cause	Remedy
<i>no device answers to station name 'X' on ethN</i>	Coupler has no or a different station name, or is on another port	Scan Profinet network , then Set the name (Chapter 10).
<i>DCP Set-IP ... failed</i>	Device refused the address, or address in use	Check for duplicates; set it manually from the scan dialog.

Symptom	Cause	Remedy
<i>the device's rack differs from the configured one</i>	Modules in the coupler differ from the GSDML rack	The message says what will happen (degrade/fault/ignore); correct the rack or the configuration.
<i>no input frame for a full data-hold time - reconnecting</i>	Cyclic data stopped: cable, switch, or controller overloaded	Check the link and switch; check the scan's jitter; the driver reconnects by itself.
<i>the device never reached data exchange</i>	Connect accepted but no cyclic data	Check the GSDML module selection and the firewall (UDP 34964 on the OT port).
<i>only async mode is supported</i>	Profinet device configured <code>sync</code>	Set <code>mode</code> to <code>async</code> .

35.5.5 IO-Link

Symptom	Cause	Remedy
Every compile fails after adding the IO-Link library	Controller runtime older than the library's built-in functions	Deploy the current runtime before adding <code>70_iolink.st</code> to a project.
Port shows no device	Port mode not set to IO-Link by the start-up SDOs, or sensor not connected	Check the IO-Link ports page and the ESI start-up commands (Chapter 12).

Note: IO-Link support has not yet been validated on real hardware. Treat unexpected behaviour as a possible product defect and report it.

35.6 OPC UA

Symptom	Cause	Remedy
Nothing listens on 4840	Server disabled or not downloaded; runtime built without OPC UA (Windows simulator)	Controller log: <code>opcua: listening ...</code> ; Chapter 29 §29.2.
<i>enabled in config but this build has no OPC UA server</i>	Runtime binary without OPC UA	Use the controller runtime or the full Linux simulator.
Client on the PC reaches a different server	Local discovery server holds 4840	Stop it or give the project another port (Chapter 29, §29.7).
Nobody can log in	Anonymous off and no <code>users</code> ; or user login on the <code>None</code> endpoint	Add <code>users</code> ; use an encrypted endpoint.
<code>users</code> vanished	Apply on the OPC UA page rewrote the section	Re-add in <code>controller.json</code> .
Secure channel rejected	Application URI does not match the certificate	Chapter 29, §29.5.4.
Certificate changes after every restart	Certificate paths empty	Set the paths.
<code>Symbols</code> folder empty	Empty publish list; no program at start	Chapter 29, §29.8.
New variable not visible	Address space built at start	Apply , restart the runtime, reconnect.

Symptom	Cause	Remedy
Writes rejected	Require security policy on (writes disabled), or variable not marked write, or a <code>CONSTANT</code>	Chapter 29, §29.5.3.
Array, structure or STRING node reads Bad	Not an elementary scalar	Publish elementary variables.
Tag reads <code>BadNodeIdUnknown</code>	Renamed or removed by a download	Update the publish list and the client.

35.7 EtherNet/IP tag server

Symptom	Cause	Remedy
HMI cannot connect	Server disabled or not downloaded; bind to another interface; firewall; Windows simulator without tag server	Controller log: <code>enip-tags: listening ...</code> ; Chapter 30 §30.2.
On a PC running the simulator, the client connects but finds no Zeus tags or a foreign device	Other EtherNet/IP software on the PC already listens on port 44818 , so the connection reaches it instead of the simulator	<code>`netstat -ano</code> <code>findstr :44818</code> ; stop that software's service while testing, test against a controller, or give the simulator another port (Chapter 30, §30.8).
<code>enip-tags: bind ...:44818: Address in use</code>	Another process on the runtime host uses the port	Stop it or change the port.
Controller shows as unknown device	EDS not registered or for the wrong board or revision	Create EDS file... ; register it (Chapter 30, §30.6).
Not found in a network browse	UDP discovery unavailable, bind on another network, or simulator behind WSL2	Connect by IP address.
Tag <i>path destination unknown</i>	Wrong name, TIME/STRING, or whole instance/array	Chapter 30, §30.3.
Nested members or arrays missing from the browse	Not listed, by design	Type the name.
Writes refused	Allow writes off; <code>CONSTANT</code> ; out of range	Chapter 30, §30.5.
64-bit or LREAL values wrong on the HMI	Truncated to DINT / rounded to REAL	Scale into a DINT/REAL variable.
Extra HMIs refused	<code>max_clients</code> (default 4) reached	Raise it, up to 16.

35.8 Modbus TCP slave

Symptom	Cause	Remedy
Runtime refuses to start: <i>enabled but maps nothing</i>	All counts 0	Map at least one space.
Permission denied on port 502	Privileged port	Use 5502 (Chapter 31, §31.2.3).
Connection refused	Bound to 127.0.0.1, or not downloaded	Set the bind address; Download hardware .
No answer, no error	Unit id mismatch	Set Unit id or 0.
ILLEGAL DATA ADDRESS	Unmapped space or beyond count; master counts from 1	Chapter 31, §31.3.2.
ILLEGAL FUNCTION on writes	Read-only	Chapter 31, §31.5.
Written value springs back	Mapped onto an output the program assigns	Map writes to %M.

35.9 Operator screens

Symptom	Cause	Remedy
Screens server does not start (<i>no accounts configured</i>)	Bound to 0.0.0.0 without accounts	Add users or bind loopback.
Panel shows <i>not found</i>	Screens not downloaded	Download screens .
Values grey or ?	Variable not reported by the running program	Fix the binding; download the program.
Button has no effect	Writes disabled; user is a viewer; program overwrites the variable	Chapter 32, §32.10.
Screen node broken	Screen file unreadable or invalid	Fix or delete the file.

35.10 Alarms and batch

Symptom	Cause	Remedy
No alarms are configured on this controller	No "alarms" section, or not downloaded	Chapter 27, §27.4.
Runtime refuses to start: <i>alarms[n]...</i>	Invalid alarm definition	Correct the entry named; validate.
An alarm never appears	Bit never set; suppressed; out of service; on-delay longer than the condition	The alarm page shows its state; check the bit in the online view.
A latched alarm stays ACK_ACTIVE after its condition cleared	It was acknowledged while still active	Chapter 27, §27.4.3.

Symptom	Cause	Remedy
Journal shows <i>event(s) were overwritten</i>	Reader fell more than 1,024 events behind	Read the journal more often from the SCADA.
Times marked (<i>clock not synced</i>)	Controller clock not synchronised	Chapter 33, §33.8.
A phase will not start	Not in IDLE, <code>permissive</code> or <code>interlock_ok</code> FALSE	COMPLETE/STOPPED/ABORTED need RESET first.
A phase went to HELD by itself	Interlock dropped while running	Intended; clear the interlock, RESTART.
A batch never gets the unit or line	Another batch owns it; release presented with 0	The queue shows the owner; release with the owning batch id (Chapter 28, §28.4.2).
Mode change refused	<code>refused_code</code> 1 phase active, 2 not allowed, 3 no key, 4 unknown	Chapter 28, §28.4.3.
Records show — in the Batch column	No batch allocated, or <code>batch_context</code> not configured	Chapter 28, §28.6.
Signature refused: initial password	Account never changed its password	Change it; Download hardware .

35.11 Retained data, power and time

Symptom	Cause	Remedy
Retained values back at initial values after a restart	Restore refused: layout changed, %M resized, or power lost before the next save after a download	Audit line <code>retain: RESTORE REFUSED - ...</code> ; restore from a data snapshot (Chapter 33, §33.3.3).
<i>stored image REJECTED - CRC mismatch</i>	Retain file damaged (power loss during a card failure, SD card wear)	Check the card; restore from backup or snapshot.
Last minutes of values lost after a power cut	Unclean power loss between periodic saves	Arm the power-fail monitor (Chapter 33, §33.5).
Power-fail monitor <i>refusing to arm</i>	Wrong pin or polarity, or pin not set	Fix <code>powerfail.conf</code> ; restart the service.
Controller shut down and restarted unexpectedly	Rapid GPIO polling disturbed the UPS handshake	Never poll the carrier's GPIO lines in a loop.
Controller clock months wrong	No time source and no RTC	Configure NTP or fit an RTC (Chapter 33, §33.8).
NTP answers but <i>Server has too large root distance</i>	Time server declares poor accuracy	<code>RootDistanceMaxSec=30</code> (recorded decision) or a better source.
SD card read-only	Card worn out	Replace; apply <code>harden-sdcard.sh</code> ; restore from backup.
Restoring a data snapshot is slow	Old runtime writes one value per request	Deploy the current runtime.

Symptom	Cause	Remedy
<i>Connection time not reported by this runtime</i>	Runtime older than the IDE	Deploy the current runtime.

35.12 Simulator

Symptom	Cause	Remedy
Simulator reports <i>HTTP 500</i> or <i>did not come up</i>	Start script checked out with Windows line endings, or runtime refused the configuration	The start report shows the runtime's own sentence; see Chapter 26.
After a restart, the simulator runs no program	The simulator may come back without the downloaded program	Download again after starting it.
No OPC UA, Modbus slave, EtherNet/IP tags or screens server	The Windows simulator (<code>bin\zeus-sim.exe</code>) has none of them	The start report says so; use a controller, or the full Linux runtime under WSL2 on a development PC.
OPC UA client reaches another server on 4840	Local discovery server	Chapter 29, §29.7.
EtherNet/IP client reaches other software on 44818	Other EtherNet/IP software on the PC	Chapter 30, §30.8.
Simulator refuses every program	Configured with <code>program_verify: enforce</code> ; the simulator cannot verify signatures	Use <code>off</code> or <code>warn</code> on the simulator.
<i>the simulator binary is older than the source</i> warning	Stale build on a development PC	Rebuild, or accept knowingly: a stale simulator may disagree with the controller about what compiles.
Timing figures differ from the controller	Windows has no real-time scheduling; scan floor 1 ms	Use the simulator for logic, not timing.
Scan for remote IO reports <i>this is a simulator with no network fieldbus</i>	By design	Scan from a controller.

35.13 When to call support

Collect before calling:

1. IDE and runtime versions (**Online** → **Controller** → **Versions...**).
2. The controller log around the event and the audit trail (**HMI** → **Events**).
3. A project archive (**Project** → **Create archive...**).
4. The device page of any affected IO device (state, reconnects, up-time).
5. A description of what the machine did, when, and what was done just before.

36. Practices for validated systems

This chapter collects the working habits that the records of a GxP installation (pharmaceutical, biotech, food under 21 CFR Part 11 or EU GMP Annex 11) depend on. The platform provides controls — named accounts, an audit trail, signatures, a retain layout signature, program signing, library locking, alarm timestamps at source — but a control only produces evidence if it is used the same way every time. Use this chapter as the basis of your site's operating procedure for Zeus controllers, and read it together with Chapter 28 (validation documents and responsibilities) and Chapter 34 (security).

WARNING: Validation does not make the controller a safety system. It carries no SIL or PL rating. Emergency stop, guard interlocking and any function that protects people belong in a certified safety relay or safety PLC wired independently of this controller.

36.1 Responsibilities

Layer	Validated by	Your deliverables
Platform: runtime, compiler, IDE, standard library, servers	Supplier (documents 27–31, 25)	Supplier assessment; acceptance of the platform documents.
Application: your program, IO configuration, phases, alarm definitions, screens, recipes	You	URS/FS for the application, IQ/OQ execution on your installation, PQ, process validation, SOPs.

The boundary is the download. The IQ and OQ protocols (document 30) ship **unexecuted**; execute them on your hardware, with your configuration, and record deviations.

36.2 People and access

1. **Give every person their own account.** The audit trail and every signature record *who*; a shared login makes that record worthless. Never share the engineer account.
2. **Change the default account before anything else.** A controller must never enter service with `admin / admin`; the IDE refuses to download it to a protected controller, and signing is impossible with it.
3. **Grant the lowest role that does the job.** Operators need `operator`; only people who download or commission need `engineer`. Review the list at least yearly and remove leavers the same day — a removed account takes effect only after **Download hardware**.
4. **Keep tokens for programs, accounts for people.** A token identity (`identities[]`) is recorded by its name; give each connected system (SCADA, historian) its own token and the viewer role unless it must write.
5. **Protect the engineering PC.** It holds the SSH key and the program-signing key. Restrict it to named users; lock it when unattended.

36.3 Configuration management

1. **Lock the library (Library → Versions...)** when it is the one you validate against, and say why in the note. Treat any **CHANGED**, **added** or **REMOVED** state as a change requiring change control before download.
2. **Put the project under version control** — every file is plain text or JSON so it can be diffed and reviewed: `programs`, `controller.json`, `globals.st`, `dut/`, `lib/`, `library.lock.json`, `hmi/`.

3. **Archive the project at every milestone** (**Project** → **Create archive...**). The archive carries a SHA-256 per file; a copied folder carries nothing. Keep the archive with the change record.
4. **Enforce program signing.** Move `program_verify` to `enforce` once all downloads verify (Chapter 34, §34.6). A controller then runs only programs signed by the site key, and a download below the stored version is refused. Re-check `enforce` after every **Deploy**.
5. **Verify what runs.** Before a batch campaign and after any intervention, use **Debug** → **Verify** and **Online** → **Download & Upload** → **Compare...**: the controller is asked what it is running rather than trusting a file on somebody's laptop.
6. **Record the configuration hash.** The runtime writes the SHA-256 of its configuration to the audit trail at every start; record it at qualification and compare it at periodic review.
7. **Keep IDE and runtime versions matched**, and treat a runtime update as a change to a validated system (IQ re-executed, determinism re-measured where required).

36.4 Downloads and online changes

1. **Read the online-change report** after every download that matters. It is the only place that says which variables were carried over, reset or dropped. Keep it with the batch record when data was carried across a download during a campaign.
2. **Save a data snapshot before and after changing setpoints or downloading** (**Online** → **Save data snapshot...**). It is the fallback if a restore of retained data is refused (Chapter 33, §33.3.3).
3. **Wait for a retain save after a download** that changes declarations (default 30 s) before any power cycle.
4. **Use Download hardware deliberately.** It restarts the runtime and stops the machine; schedule it outside production and record it.
5. **Never download to the wrong line.** Check the target badge before every download; keep each controller's target type (CM4/CM5) correct so a mismatch is refused.

36.5 Forces, writes and debugging

1. **Never clear forces without reading the list first.** **Debug** → **Force / write table...** is a safety document: a force hides the real state of a point. Record why each force was set and who removed it.
2. **No breakpoints or single-stepping on a machine in GxP production.** A breakpoint halts the program mid-scan and the process is not controlled while it is halted.
3. **Use the trace, not forcing, to investigate.** A trace changes nothing and is recorded with the scan number.
4. **Do not use CONFIG mode as isolation.** Follow the site's lock-out procedure.

36.6 Records

1. **Keep the controller clock synchronised** to the site's controlled time source (Chapter 33, §33.8). Every audit record, alarm and signature carries a clock-trust flag; records written while it is false are provisional. Check `timedatectl` and the `clock:` audit line at every visit.
2. **Ship the audit trail off the controller** (`log.syslog_host`) to a collector the controller cannot rewrite. The copy on the SD card is lost with the card, and a volatile journal deletes it at every power cycle.
3. **Collect the alarm journal continuously.** The controller keeps 1,024 alarm events in RAM; `RAISE` and `CLEAR` are nowhere else. Have the SCADA or historian read it with the cursor, and investigate any *event(s) were overwritten* report as a data-integrity deviation.

4. **Collect signatures.** The audit trail holds every signature; the in-memory list holds the last 256. Export them to the batch record system.
5. **Stamp records with the batch.** Configure `batch_context` and `S88_CONTEXT` (Chapter 28, §28.6) so alarms and signatures say which batch, unit and phase they belong to, instead of relying on correlating two clocks.
6. **The controller is a source of records, not their archive.** Your retention periods are met by the historian and batch system, not by the controller.

36.7 Electronic signatures

1. Sign **what** you are responsible for, with its **SHA-256**, so that anybody holding the document can prove it is the one signed.
2. Choose the **meaning** deliberately (`approved` , `reviewed` , `released` , `verified` , `authored` , `rejected`) and state the **reason** in words a reviewer understands.
3. Sign only as yourself. The controller asks for the signer's password at the moment of signing; never type another person's password.
4. Check the signature number returned and whether the controller's clock was synchronised; if not, record the deviation.

36.8 Alarms

1. Define alarms that must be acknowledged or investigated as **controller alarms** (Chapter 27), not only as screen alarm lists.
2. Rationalise the alarm set: every alarm has a consequence, a priority and an operator response. Few `CRITICAL` alarms.
3. Shelving is time-limited and recorded; define in the SOP who may shelve, for how long, and when shelving must be reported.
4. Taking an alarm out of service removes a protection: engineer only, under change control, with the reason recorded outside the controller.
5. Suppression by design is part of the alarm definition and is qualified with it.

36.9 Integration servers

1. Prefer **OPC UA** with users and a security policy for SCADA, historian and batch engine connections. Every OPC UA write is audited. Note that in this version variables are writable only when the security policy is not required (Chapter 29, §29.5.3); document that decision and the compensating network controls.
2. Keep the **EtherNet/IP tag server read-only**: its writes are not audited in this version (Chapter 30, §30.5).
3. Keep the **Modbus slave read-only** unless a master genuinely has to command the controller; its writes are audited with the master's address but not with a person.
4. **Operator screens**: named accounts, writes by operators only; every write is audited with the user name.
5. Restrict every server with the host firewall (Chapter 34, §34.9).

36.10 Retained data and power

1. Retain only what the process needs after a restart. Restart sequences from a known state.
2. Arm and test the **power-fail save** (Chapter 33, §33.5) at commissioning and yearly.

3. After every restart, check the `retain:` audit line. A **RESTORE REFUSED** is a data-integrity event: record it and restore values from the latest snapshot under procedure.
4. Keep `VAR_GLOBAL RETAIN` in `globals.st`.

36.11 Backups and recovery

1. Back up each controller (Chapter 33, §33.7) before every change and on a schedule; store the archive in an access-controlled location — it contains credentials and keys.
2. Restore only onto the **same** controller. Replace a failed controller by installing a fresh image, deploying and downloading, then re-executing the affected IQ steps.
3. Test a restore at least once, on a spare controller, as part of qualification.

36.12 Measurements and qualification evidence

1. Take determinism figures (jitter, overruns) only on a controller prepared for real time: `PREEMPT_RT` kernel, isolated scan core, `performance` governor. A figure from an unprepared controller measures the operating system, not the product.
2. Do not use the **simulator** for qualification evidence. It runs the real scan engine and compiler, but has no real-time scheduling and no fieldbus; use it to develop and rehearse, not to qualify.
3. IO-Link support is not yet validated on real hardware; qualify it on your hardware before relying on it.

36.13 Change control and periodic review

Change	Minimum response
Program change	Change request; review of the diff; download with online-change report kept; affected tests re-run.
Library change	As above, plus library lock renewed with a reason.
Configuration change (IO, alarms, servers, accounts)	Change request; Validate with the runtime ; Download hardware scheduled; configuration SHA-256 recorded.
Runtime update (Deploy)	Change request; backup; IQ steps re-executed; determinism re-measured if the scan or kernel changed; <code>enforce</code> re-checked.
Operating system or kernel update	IQ re-executed; determinism figures retaken.
Security event (lost key, leaver with access, suspected intrusion)	Rotate the affected credential (token, passwords, signing key — Chapter 34); review the audit trail.

Review each controller at least yearly: audit trail completeness, forces left in place, accounts and roles, clock synchronisation history, retain refusals, alarm shelving and out-of-service history, open deviations.

36.14 Quick checklist before a controller enters GxP service

1. No account on its initial password; one account per person; roles reviewed.
2. Monitor on loopback with the SSH tunnel; host firewall committed.
3. `program_verify` = `enforce` ; program downloaded and verified.

4. Library locked with a reason; project archived; archive hash recorded.
5. Clock synchronised; `clock: ... SYNCHRONIZED` in the audit trail.
6. Audit trail shipped to the site collector.
7. Alarm journal and signatures collected by the SCADA/historian.
8. `batch_context` configured if batches are run.
9. Power-fail save armed and tested.
10. Controller backup taken and stored.
11. IQ and OQ executed, deviations closed.

37. Licensing and demo time

The Zeus PLC IDE is free to install and use. What is licensed is the **runtime**: the software that runs the program on the controller. Every Zeus controller — **IPC-CM400200** (Raspberry Pi CM4) and **IPC-CM500200** (CM5) — leaves the factory with a license and runs without any limit. The same runtime on other hardware, and the simulator on an engineering PC, run in **demo mode**.

This chapter explains what demo mode allows, what happens when its time runs out, and how a license is read, requested and installed.

37.1 What runs where

Runtime	Simulated IO only	Real IO
Zeus controller (licensed)	No limit	No limit
Simulator on a PC	No limit	Demo: 2 hours , then the program stops
Runtime on any other Raspberry Pi / CM4 / CM5	No limit	Demo: 2 hours , then the program stops

Real IO means either of:

- an enabled IO device whose driver is not the simulator: Profinet, EtherCAT, EtherNet/IP or a Modbus TCP RIO - except a Modbus TCP device **on the same computer** (`127.0.0.1` , `localhost`), which is a process simulation such as Factory I/O (26.6a), not a machine;
- the **Modbus TCP slave** switched on, which lets a SCADA or another device read and write the controller over the network.

With simulated IO only — the IO devices of the project replaced by the simulator's signal generators — nothing physical is controlled, so there is no limit at all: learning, testing logic, training and demonstrations cost nothing.

The OPC UA server, the EtherNet/IP tag server and the operator screens do not count as real IO on their own.

Note: The simulator in the released IDE package always simulates the IO of every device, and has no Modbus slave. It therefore never reaches the limit. The limit matters when the runtime is put on non-Zeus hardware with a fieldbus, or when a development simulator with real IO or the Modbus slave is used.

37.2 When the demo time runs out

The demo clock starts when the runtime starts. The IDE shows the time left as a pill beside the run state, for example **DEMO 1:42:10**, and:

- at **10 minutes** left, a warning in the Messages panel and a notice;
- at **1 minute** left, a notice;
- when the time is up, the pill reads **DEMO ENDED** and a window explains what happened.

At that moment the runtime **stops the program** by putting the controller into **CONFIG** (Chapter 21):

- the program no longer runs;

- outputs are no longer driven — each fieldbus releases its outputs to the state its own watchdog defines, as in CONFIG and FAULT;
- inputs are still read, so values can still be watched;
- **Run, Resume, Reset fault** and leaving config mode are refused with the message *"Demo time ended: this runtime has no Zeus license and uses real IO..."*;
- the event is written to the audit trail.

To run again, either **restart the runtime** (Online → Controller → Restart runtime...), which starts another 2 hours, or **install a license** (section 37.4), which releases the controller at once.

WARNING: Demo mode is for evaluation, commissioning practice and testing. It is not for production: the program stops after 2 hours whatever the machine is doing. Production needs a Zeus controller.

37.3 Checking the license

Online → Controller → License... shows, for the connected controller or simulator:

Field	Meaning
Status	Licensed (with the model), Demo mode with the time left, Demo time ended , or Not licensed - simulated IO only, no time limit
Customer, Issued	From the license, when there is one
Hardware id	This machine's identity: <code>rpi:<CPU serial></code> on a Raspberry Pi / CM, <code>host:<machine id></code> on a PC. Copy puts it on the clipboard.
Detail	Why: for example <i>no license file</i> , <i>the license signature is not valid</i> , or <i>this license is for rpi:..., but this machine is rpi:...</i>

A runtime older than 0.1.0.10 has no license support; the dialog says so.

37.4 Getting and installing a license

1. Open **Online → Controller → License...** while connected to the machine, and press **Copy** beside the hardware id.
2. Send the hardware id to Zeus Automation, with the customer name.
3. You receive a small file (`...json`). In the same dialog press **Install license file...** and choose it.
4. The runtime checks the file and, if it is valid for this machine, saves it and adopts it at once. A controller whose demo had ended goes to **PAUSED**; press **Run**.

The runtime refuses — and saves nothing — a file that:

- was changed after it was issued, or was not issued by Zeus Automation (*the license signature is not valid*);
- was issued for another machine (*this license is for ..., but this machine is ...*). A license cannot be copied from one controller to another.

The license is stored on the controller in `/var/lib/zeus-plc/license.json` and read at every start. Replacing the controller's SD card or reinstalling its image does not change its CPU serial: re-install the same license file.

Replacing the controller itself needs a new license.

Note: Every audit record of a license — installed, refused, demo ended — names the address it came from (Chapter 34).

37.5 New and existing Zeus controllers

Zeus controllers are licensed during production, as a step of the master image procedure (Controller Master Image Guide, chapter 6). Controllers delivered before licensing was introduced are licensed the same way when their runtime is updated to 0.1.0.10 or later: Zeus Automation issues the license for the controller's CPU serial; it is installed over the network or with **Install license file....**

Install the license **before or together with** the runtime update. A Zeus controller with real IO that is updated without one runs in demo mode until the license is installed.

ZEUS PLC IDE · OPERATION MANUAL

Appendices

Appendix A. Built-in functions

These are all the built-in functions of the Zeus Structured Text compiler - the list is generated from the compiler's own table, so a name not listed here does not exist. Built-in functions take **positional** arguments (`MAX(a, b)`), not `MAX(IN1 := a)`). Function blocks such as TON and CTU are not functions; they are in Appendix B, and the library blocks (PID, SCALE, ...) in Appendix C.

Note: There are no generic `<TYPE>_TO_<TYPE>` conversions beyond those listed. Integer types convert implicitly on assignment (a narrowing assignment truncates, and the compiler warns when a REAL is assigned to an integer).

Function	Arguments	Returns
ABS	1	LREAL
SQRT	1	LREAL
LN	1	LREAL
LOG	1	LREAL
EXP	1	LREAL
SIN	1	LREAL
COS	1	LREAL
TAN	1	LREAL
ASIN	1	LREAL
ACOS	1	LREAL
ATAN	1	LREAL
MIN	2 or more	type of the arguments (promoted)
MAX	2 or more	type of the arguments (promoted)
LIMIT	3	type of the arguments (promoted)
SEL	3	type of a
MUX	2 or more	type of v0
SHL	2	DINT
SHR	2	DINT
ROL	2	DINT
ROR	2	DINT
INT_TO_REAL	1	LREAL
DINT_TO_REAL	1	LREAL
REAL_TO_INT	1	DINT

Function	Arguments	Returns
REAL_TO_DINT	1	DINT
BOOL_TO_INT	1	DINT
INT_TO_BOOL	1	BOOL
TIME_TO_INT	1	DINT
INT_TO_TIME	1	TIME
TRUNC	1	DINT
ROUND	1	DINT
TIME_NOW	0	TIME
PI_IN_BYTE	1	DINT
PI_OUT_BYTE	1	DINT
PI_SET_OUT_BYTE	2	BOOL
DWORD_AS_REAL	1	REAL

A.1 Arithmetic and mathematics

Call	Description	Returns
ABS(x)	Absolute value.	LREAL
SQRT(x)	Square root. A negative argument gives NaN.	LREAL
LN(x)	Natural logarithm.	LREAL
LOG(x)	Base-10 logarithm.	LREAL
EXP(x)	e raised to x.	LREAL
SIN(x)	Sine, x in radians.	LREAL
COS(x)	Cosine, x in radians.	LREAL
TAN(x)	Tangent, x in radians.	LREAL
ASIN(x)	Arc sine, result in radians.	LREAL
ACOS(x)	Arc cosine, result in radians.	LREAL
ATAN(x)	Arc tangent, result in radians.	LREAL

A.2 Selection and limits

Call	Description	Returns
MIN(a, b, ...)	Smallest of two or more values.	type of the arguments (promoted)

Call	Description	Returns
<code>MAX(a, b, ...)</code>	Largest of two or more values.	type of the arguments (promoted)
<code>LIMIT(mn, x, mx)</code>	x clamped to the range mn..mx.	type of the arguments (promoted)
<code>SEL(g, a, b)</code>	a when g is FALSE, b when g is TRUE.	type of a
<code>MUX(k, v0, v1, ...)</code>	The value selected by the index k (0 = v0).	type of v0

A.3 Bit shift and rotate

Call	Description	Returns
<code>SHL(x, n)</code>	Shift x left by n bits.	DINT
<code>SHR(x, n)</code>	Shift x right by n bits.	DINT
<code>ROL(x, n)</code>	Rotate x left by n bits.	DINT
<code>ROR(x, n)</code>	Rotate x right by n bits.	DINT

A.4 Type conversion

Call	Description	Returns
<code>INT_TO_REAL(i)</code>	Integer to floating point.	LREAL
<code>DINT_TO_REAL(i)</code>	Integer to floating point (same as INT_TO_REAL).	LREAL
<code>REAL_TO_INT(r)</code>	Floating point to integer, rounded to the nearest (half away from zero).	DINT
<code>REAL_TO_DINT(r)</code>	Same as REAL_TO_INT.	DINT
<code>TRUNC(r)</code>	Floating point to integer, fraction discarded (towards zero).	DINT
<code>ROUND(r)</code>	Round to the nearest integer (half away from zero).	DINT
<code>BOOL_TO_INT(b)</code>	TRUE -> 1, FALSE -> 0.	DINT
<code>INT_TO_BOOL(i)</code>	Non-zero -> TRUE.	BOOL
<code>TIME_TO_INT(t)</code>	Duration in milliseconds.	DINT
<code>INT_TO_TIME(ms)</code>	Milliseconds to a TIME value.	TIME

A.5 Time

Call	Description	Returns
<code>TIME_NOW()</code>	Monotonic controller time, sampled once per scan - every call in one scan returns the same value.	TIME

A.6 Process image and raw data

Call	Description	Returns
PI_IN_BYTE(off)	Byte off of the input image %I; -1 if outside the image.	DINT
PI_OUT_BYTE(off)	Byte off of the output image %Q; -1 if outside the image.	DINT
PI_SET_OUT_BYTE(off, v)	Write the low 8 bits of v to byte off of %Q. FALSE if outside the image.	BOOL
DWORD_AS_REAL(bits)	Reinterpret a 32-bit IEEE-754 bit pattern as REAL (Float32 values from IO-Link or Modbus).	REAL

A.7 Examples

```

PROGRAM FnDemo
VAR
    raw      : INT := 13824;
    pct      : REAL;
    limited  : DINT;
    bigger   : DINT;
    pick     : DINT;
    ms       : DINT;
    t        : TIME;
    bits     : DINT;
END_VAR
pct      := INT_TO_REAL(raw) / 276.48;      (* 0..27648 counts -> 0..100 % *)
limited   := LIMIT(0, REAL_TO_INT(pct), 100);
bigger    := MAX(raw, 20000, 5);
pick     := MUX(2, 10, 20, 30);            (* 30 *)
t        := INT_TO_TIME(1500);             (* T#1.5s *)
ms       := TIME_TO_INT(t);
bits     := SHL(1, 4) OR 16#01;            (* 16#11 *)
END_PROGRAM

```

Appendix B. Standard function blocks

The ten IEC 61131-3 standard function blocks built into the runtime. Declare an instance (`t1 : TON;`), call it with **named** parameters, and read its outputs as `t1.Q`. Inputs are bound with `:=`, outputs may be bound in the call with `=>` (`t1(IN := go, PT := T#5s, Q => done);`).

Every instance keeps its own state - including a timer declared inside a user function block, which is separate for every instance of that block and readable online as `Pump1.Tmr.ET`.

B.1 TON

On-delay timer. Q goes TRUE when IN has been TRUE for PT; ET counts up to PT.

Pin	Type	Direction
IN	BOOL	input
PT	TIME	input
Q	BOOL	output
ET	TIME	output

B.2 TOF

Off-delay timer. Q follows IN on, and stays TRUE for PT after IN goes FALSE.

Pin	Type	Direction
IN	BOOL	input
PT	TIME	input
Q	BOOL	output
ET	TIME	output

B.3 TP

Pulse timer. A rising edge on IN gives a Q pulse of exactly PT, not retriggrable.

Pin	Type	Direction
IN	BOOL	input
PT	TIME	input
Q	BOOL	output
ET	TIME	output

B.4 CTU

Up counter. Each rising edge of CU adds 1 to CV; Q = CV >= PV; R resets CV to 0.

Pin	Type	Direction
CU	BOOL	input
R	BOOL	input
PV	INT	input
Q	BOOL	output
CV	INT	output

B.5 CTD

Down counter. Each rising edge of CD subtracts 1; Q = CV <= 0; LD loads PV into CV.

Pin	Type	Direction
CD	BOOL	input
LD	BOOL	input
PV	INT	input
Q	BOOL	output
CV	INT	output

B.6 CTUD

Up/down counter. QU = CV >= PV, QD = CV <= 0; R resets, LD loads PV.

Pin	Type	Direction
CU	BOOL	input
CD	BOOL	input
R	BOOL	input
LD	BOOL	input
PV	INT	input
QU	BOOL	output
QD	BOOL	output
CV	INT	output

B.7 R_TRIG

Rising-edge detector: Q is TRUE for one scan when CLK goes FALSE -> TRUE.

Pin	Type	Direction
CLK	BOOL	input
Q	BOOL	output

B.8 F_TRIG

Falling-edge detector: Q is TRUE for one scan when CLK goes TRUE -> FALSE.

Pin	Type	Direction
CLK	BOOL	input
Q	BOOL	output

B.9 RS

Reset-dominant latch: Q1 set by S, reset by R1; R1 wins when both are TRUE. Q is an alias of Q1.

Pin	Type	Direction
S	BOOL	input
R1	BOOL	input
Q1	BOOL	output
Q	BOOL	output

B.10 SR

Set-dominant latch: Q1 set by S1, reset by R; S1 wins when both are TRUE. Q is an alias of Q1.

Pin	Type	Direction
S1	BOOL	input
R	BOOL	input
Q1	BOOL	output
Q	BOOL	output

B.11 Example

```
PROGRAM FbDemo
VAR
    start_pb : BOOL;
    part_seen : BOOL;
    reset_pb : BOOL;
    delay : TON;
    edge : R_TRIG;
    parts : CTU;
    latch : SR;
    run : BOOL;
    batch_ok : BOOL;
END_VAR
    edge(CLK := start_pb);
    latch(S1 := edge.Q, R := reset_pb);
    run := latch.Q1;

    delay(IN := run, PT := T#2s); (* run for 2 s before counting *)
    parts(CU := part_seen AND delay.Q, R := reset_pb, PV := 50);
    batch_ok := parts.Q; (* 50 parts counted *)
END_PROGRAM
```

Appendix C. Library reference

Every block of the function-block library that ships with the IDE (`stdlib\`), generated from the library source. A new project gets its own copy in `lib\` ; see Chapter 19 for managing it. Inputs have the default shown when the call does not set them.

Important: Blocks that integrate or differentiate over time take `dt_ms` , and it must equal the interval of the task that calls them. A wrong value does not fail - it quietly detunes the block.

C.1 10_signal.st — zeus-signal 1.0.0

Signal conditioning.

C.1.1 SCALE

Raw analogue count to engineering units, with out-of-range detection.

The range check is on the RAW value, before clamping: a 4-20 mA sensor reading below its low count is a broken wire, and that has to stay distinguishable from a real reading that happens to sit at the bottom.

Inputs

Name	Type	Default	Description
<code>raw</code>	INT		from the card, e.g. %IW2
<code>raw_min</code>	INT	<code>0</code>	
<code>raw_max</code>	INT	<code>27648</code>	common full-scale count
<code>eu_min</code>	REAL	<code>0.0</code>	
<code>eu_max</code>	REAL	<code>100.0</code>	
<code>clamp</code>	BOOL	<code>TRUE</code>	hold output inside eu_min..eu_max

Outputs

Name	Type	Default	Description
<code>eu</code>	REAL		engineering units
<code>under</code>	BOOL		raw below raw_min - suspect wiring
<code>over</code>	BOOL		raw above raw_max

C.1.2 FILTER_PT1

First-order lag filter.

The standard way to stop a noisy transmitter driving a control loop. tau is the time to reach 63% of a step.

tau shorter than the scan period is meaningless and would produce a gain above 1 - i.e. amplification and ringing - so it is bounded here rather than left to surprise someone.

Inputs

Name	Type	Default	Description
x	REAL		
tau_ms	DINT	1000	
dt_ms	DINT	10	MUST equal the task interval
reset	BOOL	FALSE	

Outputs

Name	Type	Default	Description
y	REAL		

C.1.3 RATE_LIMIT

Rate limiter - the output may not move faster than up/down per second.

Used to protect an actuator from a step, and to keep a setpoint change from being seen by the process as a disturbance.

Inputs

Name	Type	Default	Description
x	REAL		
up_per_s	REAL	10.0	
dn_per_s	REAL	10.0	
dt_ms	DINT	10	
reset	BOOL	FALSE	

Outputs

Name	Type	Default	Description
y	REAL		
limited	BOOL		TRUE while the rate is being held

C.1.4 ALARM_AN

Four-level analogue alarm with hysteresis and an on-delay.

Hysteresis and delay are both here because either alone still chatters: a value sitting exactly on a limit toggles without hysteresis, and a noisy spike trips without a delay. The pair is what makes an alarm list readable.

Hysteresis applies on the RETURN only, which is the convention operators expect - an alarm appears the moment the limit is crossed.

Inputs

Name	Type	Default	Description
pv	REAL		
hh	REAL	95.0	
h	REAL	90.0	
l	REAL	10.0	
ll	REAL	5.0	
hyst	REAL	1.0	
delay	TIME	T#1s	
enable	BOOL	TRUE	

Outputs

Name	Type	Default	Description
hh_alm	BOOL		
h_alm	BOOL		
l_alm	BOOL		
ll_alm	BOOL		
any	BOOL		

C.1.5 TOTALIZER

Totaliser - integrate a rate into a total.

rate is per the unit named by per_s/per_min/per_hour, chosen by rate_base: 0 = per second, 1 = per minute, 2 = per hour.

Declare the instance's total as RETAIN in the calling program if the count must survive a restart; this block cannot declare that for you.

Inputs

Name	Type	Default	Description
rate	REAL		
rate_base	INT	2	0=/s 1=/min 2=/hour
dt_ms	DINT	10	
run	BOOL	TRUE	
reset	BOOL	FALSE	
cutoff	REAL	0.0	ignore rates below this - stops drift

Outputs

Name	Type	Default	Description
total	LREAL		

C.1.6 AVERAGE_N

Moving average over the last n samples.

Chosen over FILTER_PT1 when the noise is a burst rather than a hiss: a first order filter smears one bad sample across its whole time constant, while an average of n dilutes it by n and then drops it entirely.

THE WINDOW IS IN SAMPLES, NOT SECONDS, so its length in time depends on the task interval. Moving this block to a slower task lengthens the average without changing a parameter - which is a real trap when a loop is retuned by moving it between tasks.

n is clamped to the buffer. Asking for more samples than the buffer holds would otherwise average whatever memory follows it.

Inputs

Name	Type	Default	Description
pv	REAL		
n	INT	8	1..32
reset	BOOL	FALSE	

Outputs

Name	Type	Default	Description
out	REAL		
ready	BOOL	FALSE until n samples have arrived	

C.1.7 MEDIAN3

Median of the last three samples - a despiker.

The one filter that removes a single-sample spike WITHOUT rounding off the edges of a real step, which is what every averaging filter does. Worth putting ahead of an average, not instead of one: it kills the outlier, the average then handles the hiss.

Costs two scans of delay on a genuine step. On a fast loop that is a real phase lag and has to be counted against the derivative term.

Inputs

Name	Type	Default	Description
pv	REAL		

Outputs

Name	Type	Default	Description
out	REAL		

C.1.8 MINMAX_HOLD

Highest and lowest a value has reached since the last reset.

For shift reports, commissioning records and "what did it peak at while nobody was watching". Kept separate from TOTALIZER because the reset points differ: a total is reset per batch, a peak per shift.

Inputs

Name	Type	Default	Description
pv	REAL		
reset	BOOL	FALSE	
enable	BOOL	TRUE	

Outputs

Name	Type	Default	Description
min_val	REAL		
max_val	REAL		
span	REAL		

C.1.9 DERIV

Rate of change, in engineering units per second.

For rate alarms - a tank emptying faster than a pump can explain, a temperature running away - and for feed-forward.

DIFFERENTIATING RAW MEASUREMENT AMPLIFIES NOISE, always. A signal with +/-0.1 units of noise on a 100 ms task produces +/-1 unit/s of pure invention. filter_s smooths the RESULT rather than the input, so a step in the measurement still shows as a burst of rate rather than being hidden; set it to zero only if the input is already filtered.

Inputs

Name	Type	Default	Description
pv	REAL		
filter_s	REAL	1.0	0 = no smoothing
dt_ms	DINT	100	

Outputs

Name	Type	Default	Description
rate	REAL		units per second

Name	Type	Default	Description
raw_rate	REAL		

C.1.10 LOOKUP16

Piecewise-linear characterisation from a table of up to 16 points.

For a non-linear sensor or valve: a tank whose volume is not proportional to level, a thermocouple over a wide span, an equal-percentage valve being linearised for a loop that assumes it is not.

X MUST INCREASE. The search walks the table forwards and stops at the first point beyond the input, so an out-of-order table produces a silently wrong answer rather than an error - there is no way to tell a mistake from an intentionally steep segment. n is the number of points in use.

Outside the table the end value is HELD, not extrapolated. Extrapolating a characterisation curve past its last measured point invents readings, and the place that happens is exactly where a sensor has failed.

Inputs

Name	Type	Default	Description
x	REAL		
n	INT	2	points in use, 2..16

In/out

Name	Type	Default	Description
xs	ARRAY[1..16] OF REAL		must increase
ys	ARRAY[1..16] OF REAL		

Outputs

Name	Type	Default	Description
y	REAL		
clamped	BOOL		input was outside the table

C.2 20_control.st — zeus-control 1.0.0

Closed-loop control.

C.2.1 PID

PID with anti-windup, bumpless transfer and derivative on the measurement.

Three choices worth stating, because they are what separates a PID that can be commissioned from one that cannot:

1. DERIVATIVE ON THE MEASUREMENT, not on the error. Differentiating the error means a setpoint step differentiates to a spike, and the output slams. Operators respond by turning derivative off, which loses it everywhere. Acting on -d(pv) gives the same damping with no kick.

2. ANTI-WINDUP BY CLAMPING THE INTEGRAL, not the output alone. If the integral is allowed to keep growing while the valve is already shut, the loop stays saturated long after the process has come back - the classic "it overshoots massively after a long low period".
3. BUMPLESS AUTO/MANUAL. On the scan that returns to auto, the integral is back-calculated so the output starts from where manual left it. Without that, going to auto steps the valve to whatever the integral had drifted to while nobody was looking.

Gains are in the parallel (ISA) form: $out = K_p \cdot e + K_i \cdot integral + K_d \cdot deriv$, with t_i/t_d in seconds. $t_i = 0$ disables integral, $t_d = 0$ disables derivative.

Inputs

Name	Type	Default	Description
pv	REAL		process value
sp	REAL		setpoint
kp	REAL	1.0	
ti_s	REAL	10.0	integral time, s. 0 = off
td_s	REAL	0.0	derivative time, s. 0 = off
dt_ms	DINT	100	
out_min	REAL	0.0	
out_max	REAL	100.0	
reverse	BOOL	FALSE	TRUE when more output lowers pv
man	BOOL	FALSE	TRUE = manual
man_out	REAL	0.0	
enable	BOOL	TRUE	

Outputs

Name	Type	Default	Description
out	REAL		
err	REAL		
sat	BOOL		output is against a limit
integral	REAL		exposed for tuning and diagnosis

C.2.2 RAMP

Ramp a value towards a target at a fixed rate.

Distinct from RATE_LIMIT: that one follows a moving input, this one is told a destination and walks to it, reporting when it arrives. Used for setpoint changes an operator should see complete.

Inputs

Name	Type	Default	Description
target	REAL		
rate_per_s	REAL	1.0	
dt_ms	DINT	100	
hold	BOOL	FALSE	
preset	BOOL	FALSE	jump to target, no ramp

Outputs

Name	Type	Default	Description
out	REAL		
ramping	BOOL		
at_target	BOOL		

C.2.3 HYSTERESIS

On/off control with a deadband - a thermostat.

Separate from ALARM_AN because the intent differs: this drives an actuator and its deadband is chosen from how often the plant can bear to cycle, not from how the value should be reported.

Inputs

Name	Type	Default	Description
pv	REAL		
on_at	REAL	20.0	
off_at	REAL	25.0	off_at > on_at = heating
enable	BOOL	TRUE	

Outputs

Name	Type	Default	Description
q	BOOL		

C.2.4 PWM

Time-proportional output: a controller's 0..100 % onto a relay or SSR.

The usual pairing for heaters, where the actuator has no analogue input and the only control available is what fraction of a period it is switched on.

PERIOD IS A COMPROMISE THIS BLOCK WILL NOT HIDE. Too long and the process ripples visibly between pulses; too short and a mechanical contactor is worn out in a season. Seconds suit an SSR, tens of seconds a contactor driving a heater bank. min_on_ms is for the contactor case: 0.4 % of a 20 s period is an 80 ms pulse, which wears the contacts without heating anything, so anything shorter is dropped to zero and the deficit left to the loop.

THE WINDOW IS NOT RESTARTED WHEN THE DEMAND CHANGES. Re-basing it on every change lets a loop that dithers hold the output on continuously, at arbitrary power - a failure that stays invisible until something overheats.

Inputs

Name	Type	Default	Description
duty	REAL		0..100 %
period_ms	DINT	2000	
min_on_ms	DINT	0	shorter pulses are not worth making
dt_ms	DINT	100	
enable	BOOL	TRUE	

Outputs

Name	Type	Default	Description
q	BOOL		
on_ms	DINT		what this window actually asked for
phase_ms	DINT		position in the window, for tracing

C.2.5 DEADBAND

Ignore movement smaller than a band, without losing where the value is.

For chatter on a value that feeds an alarm, a log or an operator display. The output tracks the input only once it has moved further than band from the last value reported - so the output is always a value the input really took, not a filtered approximation of one.

Inputs

Name	Type	Default	Description
pv	REAL		
band	REAL	1.0	

Outputs

Name	Type	Default	Description
out	REAL		
moved	BOOL		TRUE on the scan the output changed

C.2.6 SPLIT_RANGE

One control output driving two opposed actuators - heat and cool, or fill and drain.

The controller works in a single 0..100 % variable. Below the changeover the first actuator is driven, above it the second, and each is scaled across its own share of the range so both see a full 0..100 %.

A NEUTRAL ZONE IS NOT FREE. Widening dead_pct stops the two actuators fighting across the changeover, at the cost of a band where nothing responds at all. On a tightly tuned loop that band becomes a limit cycle: the loop integrates through the dead zone, overshoots, and hunts. Start at zero and open it only if the actuators are visibly fighting.

Inputs

Name	Type	Default	Description
ctrl	REAL		0..100 % from the controller
split_pct	REAL	50.0	changeover point
dead_pct	REAL	0.0	neutral band, centred on the split

Outputs

Name	Type	Default	Description
a_out	REAL		driven below the split, 0..100 %
b_out	REAL		driven above the split, 0..100 %
in_dead	BOOL		

C.2.7 THREE_POINT

Drive a motorised valve that has only open and close contacts.

There is no position feedback, so position is ESTIMATED by integrating how long each contact has been energised against the valve travel time. That estimate drifts - every start and stop loses a little to backlash - which is why the block drives hard to a limit whenever the demand reaches 0 or 100 % and re-datums there. A valve that never sees either end of its range loses registration slowly, and the fix is a periodic full stroke, not a cleverer estimator. Treat pos_pct as an indication, never as a measurement.

NEVER ENERGISES BOTH CONTACTS. That is a wiring-level fault in most actuators, so it is refused here as well as in the panel.

Inputs

Name	Type	Default	Description
demand	REAL		0..100 % wanted
travel_s	REAL	30.0	time for a full stroke
band_pct	REAL	2.0	stop within this much of the demand
dt_ms	DINT	100	
enable	BOOL	TRUE	

Outputs

Name	Type	Default	Description
open_cmd	BOOL		

Name	Type	Default	Description
close_cmd	BOOL		
pos_pct	REAL		ESTIMATED, not measured
at_limit	BOOL		

C.2.8 RATIO

Hold one flow in proportion to another.

The wild stream is measured, not controlled; this produces the setpoint for the loop that controls the second one. Blending, dosing and air/fuel all take this shape.

THE OUTPUT IS FORCED TO ZERO BELOW min_wild. Ratio on a dead flow means dosing into a pipe with nothing moving in it, which is how a batch is ruined quietly: the meter reads noise, the ratio multiplies it, and the dosing pump runs against a closed line.

Inputs

Name	Type	Default	Description
wild	REAL		measured, uncontrolled flow
ratio	REAL	1.0	
bias	REAL	0.0	added after the ratio
min_wild	REAL	0.0	below this the output is zero
sp_max	REAL	100.0	

Outputs

Name	Type	Default	Description
sp	REAL		setpoint for the controlled loop
active	BOOL		
clamped	BOOL		

C.3 22_autotune.st — zeus-autotune 1.0.0

Relay-feedback PID auto-tuner (Astrom-Hagglund).

C.3.1 PID_AutoTune

Relay-feedback PID auto-tuner (Astrom-Hagglund).

Sorts after 20_control.st but does not depend on it - it PRODUCES tuning, it does not run a PID. Wire its out to the actuator during a tune; when done goes true, copy Kp , Ti_s , Td_s into your PID and hand control back.

How it works: instead of the loop's PID, a RELAY drives the output between out_bias+amp and out_bias-amp based on the sign of the error. That forces a steady oscillation at the process's ultimate frequency. From the

oscillation amplitude a and period T_u : $K_u = 4 \text{ amp} / (\pi a)$ (ultimate gain) and a tuning rule turns (K_u, T_u) into PID gains. Tyreus-Luyben is the default because it is far less oscillatory than Ziegler-Nichols on real plant.

Inputs

Name	Type	Default	Description
Start	BOOL		rising edge starts a tune
Abort	BOOL		cancel and return to idle
pv	REAL		process value
sp	REAL		setpoint to oscillate around
reverse	BOOL	FALSE	TRUE when more output LOWERS pv (cooling)
out_bias	REAL	50.0	nominal output the relay steps around, %
relay_amp	REAL	10.0	relay step each side of the bias, %
relay_hyst	REAL	0.5	pv noise band - switch only outside it
rule	INT	1	0 = Ziegler-Nichols, 1 = Tyreus-Luyben
settle_cycles	INT	2	initial cycles to ignore (transient)
tune_cycles	INT	4	good cycles to average
timeout_s	REAL	600.0	give up if no clean oscillation
dt_ms	DINT	100	

Outputs

Name	Type	Default	Description
out	REAL		drive the actuator with this
tuning	BOOL		
done	BOOL		
failed	BOOL		
progress	INT		good cycles collected so far
Ku	REAL		ultimate gain (diagnostic)
Tu	REAL		ultimate period (diagnostic), s
Kp	REAL		-> PID.kp
Ti_s	REAL		-> PID.ti_s
Td_s	REAL		-> PID.td_s

C.4 30_equipement.st — zeus-equipment 1.0.0

Equipment drivers - motors and valves.

C.4.1 MOTOR_DOL

Direct-on-line motor with run feedback and a start/stop timeout.

The timeouts are the point. Commanding a motor and never checking it actually ran is how a line runs "successfully" with a tripped starter, and the first thing anyone knows is a full tank downstream.

Inputs

Name	Type	Default	Description
start	BOOL		command - level, not a pulse
stop	BOOL		overrides start
interlock_ok	BOOL	TRUE	FALSE stops and blocks starting
fb_running	BOOL		from the starter contact
ext_trip	BOOL	FALSE	overload / thermistor
fb_timeout	TIME	T#3s	
reset	BOOL	FALSE	
dt_ms	DINT	10	

Outputs

Name	Type	Default	Description
run_cmd	BOOL		to the contactor
running	BOOL		commanded AND confirmed
fault	BOOL		
run_hours	REAL		

C.4.2 VALVE_2POS

Two-position valve with limit switches and a travel timeout.

Handles the four states an operator actually needs to see: open, closed, travelling, and "neither switch made after the travel time", which is the one that means something is wrong.

Both limits made at once is a wiring or switch failure and is reported separately - it is not a state a valve can physically be in.

Inputs

Name	Type	Default	Description
open_cmd	BOOL		
interlock_ok	BOOL	TRUE	
fb_opened	BOOL		
fb_closed	BOOL		

Name	Type	Default	Description
travel_time	TIME	T#10s	
has_limits	BOOL	TRUE	FALSE = no switches fitted
reset	BOOL	FALSE	

Outputs

Name	Type	Default	Description
out_open	BOOL		to the solenoid
is_open	BOOL		
is_closed	BOOL		
travelling	BOOL		
fault	BOOL		

C.4.3 DEBOUNCE

Debounce a noisy discrete input.

The input must hold its new state for `stable_time` before the output moves. A limit switch that chatters produces one clean transition instead of nine.

Inputs

Name	Type	Default	Description
x	BOOL		
stable_time	TIME	T#50ms	

Outputs

Name	Type	Default	Description
q	BOOL		

C.4.4 BLINK

Square-wave generator, for lamps and heartbeats.

A separate on and off time so a heartbeat can be a short flash rather than a 50% square, which reads as a fault indication on a panel.

Inputs

Name	Type	Default	Description
enable	BOOL	TRUE	
on_time	TIME	T#500ms	
off_time	TIME	T#500ms	

Outputs

Name	Type	Default	Description
q	BOOL		

C.4.5 RUNTIME_HRS

Hour meter with a service interval.

Runtime drives maintenance on nearly every rotating machine, and the number is otherwise kept on a clipboard that is lost when the panel is replaced.

COUNTS SECONDS INTERNALLY, not hours. Accumulating hours directly means adding roughly 0.00003 to a REAL every scan; once the total passes a few thousand hours the increment falls below the precision of a 32-bit REAL and the meter STOPS ADVANCING while still looking healthy. Seconds keep the running total three orders of magnitude smaller, and hours are derived on the way out.

ack clears the due flag and starts the next interval from the current reading. total_h is never reset by the block - it is the life of the machine, and a service does not undo it.

Inputs

Name	Type	Default	Description
running	BOOL		
service_h	REAL	2000.0	interval; 0 disables the due flag
ack	BOOL	FALSE	service done, start the next one
dt_ms	DINT	100	

Outputs

Name	Type	Default	Description
total_h	REAL		life of the machine
since_h	REAL		since the last acknowledged service
due	BOOL		
remain_h	REAL		

C.4.6 PUMP_ALT2

Duty/standby alternation for two pumps.

Alternating on RUNTIME rather than on starts is what keeps a pair wearing evenly. Alternating per start sounds equivalent and is not: a duty pump that runs for eight hours and a standby that is bumped for a minute during a test each count one start, and the pair diverges until one fails years early.

THE SWAP ONLY HAPPENS WHEN DEMAND FALLS AWAY. Changing pumps under load means an open valve, a stopped pump and a pressure transient; the block therefore waits for a moment with no demand, which on a duty/standby set arrives often enough. If demand never falls, the pair does not alternate, and that is visible in the runtime figures rather than hidden.

NOT A LEAD/LAG CONTROLLER. Only one pump runs at a time, except during the changeover overlap, which exists so the header does not lose pressure.

Inputs

Name	Type	Default	Description
demand	BOOL		one pump should be running
a_avail	BOOL	TRUE	healthy and in auto
b_avail	BOOL	TRUE	
a_hours	REAL		from each pump's RUNTIME_HRS
b_hours	REAL		
lead_gap_h	REAL	50.0	hours of imbalance before a swap
overlap	TIME	T#2s	
dt_ms	DINT	100	

Outputs

Name	Type	Default	Description
a_run	BOOL		
b_run	BOOL		
lead_is_b	BOOL		which pump is currently the duty
none_avail	BOOL		

C.4.7 TRIP_FIRST_OUT

Trip latch that remembers WHICH input tripped first.

When a machine trips, every interlock downstream trips a few milliseconds later, and by the time anyone looks at the panel all eight lamps are lit. The first cause is the only one worth knowing, and it is the one the scan order destroys. This latches it.

INPUTS ARE HEALTHY-WHEN-TRUE, matching a normally-closed safety loop: a broken wire reads FALSE and trips. Passing an active-high fault signal straight in gives a machine that runs happily with a cut cable.

NOT A SAFETY FUNCTION. This is a process interlock in a soft PLC. Anything protecting people belongs in a rated safety relay or safety controller, in hardware, independent of this program.

Inputs

Name	Type	Default	Description
ok_1	BOOL	TRUE	TRUE = healthy
ok_2	BOOL	TRUE	
ok_3	BOOL	TRUE	
ok_4	BOOL	TRUE	

Name	Type	Default	Description
ok_5	BOOL	TRUE	
ok_6	BOOL	TRUE	
ok_7	BOOL	TRUE	
ok_8	BOOL	TRUE	
reset	BOOL	FALSE	

Outputs

Name	Type	Default	Description
tripped	BOOL		
first_out	INT		1..8, or 0 if healthy
healthy	BOOL		all inputs good right now

C.4.8 SOFT_START

Ramp a drive reference up and down, holding a minimum while it runs.

RATE_LIMIT already ramps a number. This adds the two things a drive needs: the output snaps to min_pct on start rather than crawling up from zero through a speed range where the motor is stalled and not cooling itself, and it drops cleanly to zero on stop rather than ramping into a deadband where the drive is energised and doing nothing.

Inputs

Name	Type	Default	Description
run	BOOL		
demand	REAL		0..100 % wanted while running
min_pct	REAL	20.0	lowest speed the motor may run at
up_s	REAL	5.0	0 to 100 % in this many seconds
down_s	REAL	5.0	
dt_ms	DINT	100	

Outputs

Name	Type	Default	Description
out	REAL		
at_speed	BOOL		within 1 % of the demand
ramping	BOOL		

C.5 40_hvac.st — zeus-hvac 1.0.0

HVAC air-handler control blocks.

C.5.1 TempHumidityCoolReheat

HVAC air-handler control blocks.

Built on the library PID (20_control.st), so this file must sort AFTER it - hence "40_". Two blocks:

TempHumidityCoolReheat - temperature + humidity where humidity can only be REDUCED (cooling coil + heating coil). Cool-with-reheat, energy- interlocked so it never cools-and-reheats unless it is actually drying.

AhuTempHumidity - the full loop: temperature AND humidity held AT setpoint from BOTH sides, adding a HUMIDIFIER so humidity below setpoint is raised, not just humidity above setpoint lowered.

A commissioning note that matters: for the DEHUMIDIFY side, control DEWPOINT (or absolute humidity), not %RH. Cooling air raises its %RH until it reaches the dewpoint and condenses, so a "more cooling -> less %RH" loop can push the wrong way above the dewpoint. Dewpoint moves monotonically with moisture, so the reverse-acting loop is always right.

Inputs

Name	Type	Default	Description
Name	STRING		
Enable	BOOL	TRUE	FALSE closes both coils
SetPoint_Temp	REAL		temperature setpoint, deg C
PresentValue_Temp	REAL		measured temperature
SetPoint_Hum	REAL		humidity setpoint (%RH) or dewpoint degC
PresentValue_Hum	REAL		measured humidity / dewpoint
Deadband	REAL	4.0	
Manual	BOOL		
ManualCool	REAL	0.0	cooling coil %, in Manual
ManualHeat	REAL	0.0	heating coil %, in Manual
Fail_Safe	BOOL		
FailSafeCool	REAL	0.0	safe cooling % (0 = closed)
FailSafeHeat	REAL	0.0	safe heating % (0 = closed)
dt_ms	DINT	100	
Ao_low	INT	0	
Ao_high	INT	27648	

Outputs

Name	Type	Default	Description
Cool_Eu	REAL		cooling coil command, 0..100 %
Cool_Ao	INT		cooling coil, scaled raw AO
Heat_Eu	REAL		heating coil command, 0..100 %
Heat_Ao	INT		heating coil, scaled raw AO
Out_Temp	REAL		temperature split-range output, 0..100
Out_Hum	REAL		dehumidification cooling demand, 0..100
Dehumidifying	BOOL		the humidity loop is actively drying
Mode	INT		0 off/satisfied 1 heat 2 cool 3 dehum(reheat) 4 manual 5 fail-safe

C.5.2 AhuTempHumidity

Inputs

Name	Type	Default	Description
Name	STRING		
Enable	BOOL	TRUE	
SetPoint_Temp	REAL		
PresentValue_Temp	REAL		
SetPoint_Hum	REAL		
PresentValue_Hum	REAL		
Deadband	REAL	4.0	temperature heat/cool changeover band, %
Manual	BOOL		
ManualCool	REAL	0.0	
ManualHeat	REAL	0.0	
ManualHumid	REAL	0.0	humidifier %, in Manual
Fail_Safe	BOOL		
FailSafeCool	REAL	0.0	
FailSafeHeat	REAL	0.0	
FailSafeHumid	REAL	0.0	
dt_ms	DINT	100	
Ao_low	INT	0	
Ao_high	INT	27648	

Outputs

Name	Type	Default	Description
Out_Temp	REAL		temperature split-range output
Dehum_Demand	REAL		dehumidify cooling demand
Humid_Demand	REAL		humidify demand
Dehumidifying	BOOL		
Humidifying	BOOL		
Mode	INT		0 satisfied 1 heat 2 cool 3 dehum(reheat) 4 manual 5 fail-safe 6 humidify

C.6 50_s88.st — zeus-s88 1.0.0

ISA-88 batch control: the layer between a recipe and a machine.

C.6.1 S88_PHASE

Inputs

Name	Type	Default	Description
cmd	INT		command from the batch engine
owner_req	DINT		batch or unit id issuing it
logic_done	BOOL		the work of the current state is done
fail	BOOL		the phase cannot continue
fail_code	INT		
interlock_ok	BOOL	TRUE	FALSE while RUNNING forces a HOLD
permissive	BOOL	TRUE	FALSE blocks starting, does not hold
auto_transient	BOOL	TRUE	
ack	BOOL		operator acknowledged a prompt
dt_ms	DINT	10	

Outputs

Name	Type	Default	Description
state	INT	1	see the table above
owner	DINT		who holds it; 0 = nobody
run_logic	BOOL		RUNNING or RESTARTING
hold_logic	BOOL		HOLDING
stop_logic	BOOL		STOPPING

Name	Type	Default	Description
abort_logic	BOOL		ABORTING
reset_logic	BOOL		RESETTING
is_idle	BOOL	TRUE	
is_running	BOOL		
is_held	BOOL		
is_complete	BOOL		
is_stopped	BOOL		
is_aborted	BOOL		
is_safe	BOOL	TRUE	IDLE, HELD, STOPPED, ABORTED, COMPLETE
t_state	TIME		how long in this state
t_running	TIME		total time spent RUNNING this run
last_fail	INT		fail_code that ended the last run
prompt	BOOL		waiting for an operator acknowledgement

C.6.2 S88_MODE

Inputs

Name	Type	Default	Description
req_mode	INT		0 = no request
req_owner	INT		
phase_active	BOOL		a phase is running on this equipment
allow_manual	BOOL	TRUE	site policy: may it go to MANUAL
manual_key	BOOL	TRUE	a physical key switch, if fitted
reset	BOOL		

Outputs

Name	Type	Default	Description
mode	INT	1	
owner	INT	1	
program_may_write	BOOL	TRUE	
operator_may_write	BOOL		
refused	BOOL		the last request was not allowed

C.6.3 S88_RESOURCE

Inputs

Name	Type	Default	Description
request	BOOL		requester wants it
requester	DINT		who is asking; 0 is not valid
release	BOOL		the owner is finished with it
heartbeat	BOOL	TRUE	the owner is still alive
timeout	TIME	T#0s	0 = never time an owner out
dt_ms	DINT	10	

Outputs

Name	Type	Default	Description
granted	BOOL		TRUE for the requester that holds it
owner	DINT		0 = free
queue_len	INT		
busy	BOOL		

C.6.4 S88_UNIT

Inputs

Name	Type	Default	Description
batch_id	DINT		0 = no batch allocated
acquire	BOOL		a batch wants this unit
release	BOOL		
phase_state	INT		the running phase's state, or 0
req_mode	INT		
req_owner	INT		
allow_manual	BOOL	TRUE	
dt_ms	DINT	10	

Outputs

Name	Type	Default	Description
allocated	BOOL		
batch	DINT		the batch this unit is making
mode	INT	1	
owner	INT	1	
phase_active	BOOL		
program_may_write	BOOL	TRUE	
t_batch	TIME		how long this batch has held it
refused	BOOL		
refused_code	INT		

C.6.5 S88_CONTEXT

Inputs

Name	Type	Default	Description
batch_id	DINT		0 = no batch allocated
phase_id	INT		0 = no phase active

Outputs

Name	Type	Default	Description
published	BOOL		TRUE while a batch is being stamped

C.7 70_iolink.st — zeus-iolink 1.0.0

IO-Link through an ODOT IP-EC-8IA (EtherCAT) or IP-PN-8IA (Profinet) master.

C.7.1 IOL_IN_INT (function, returns INT)

16-bit signed, MSB first, starting at %I byte `off`.

Inputs

Name	Type	Default	Description
off	DINT		

C.7.2 IOL_IN_UINT (function, returns DINT)

16-bit unsigned, MSB first.

Inputs

Name	Type	Default	Description
off	DINT		

C.7.3 IOL_IN_DINT (function, returns DINT)

32-bit signed, MSB first.

Inputs

Name	Type	Default	Description
off	DINT		

C.7.4 IOL_IN_REAL (function, returns REAL)

32-bit IEEE float, MSB first - the usual Float32 of IO-Link sensors.

Inputs

Name	Type	Default	Description
off	DINT		

C.7.5 IOL_OUT_INT (function, returns BOOL)

16-bit value to %Q byte off, MSB first. FALSE if outside the image.

Inputs

Name	Type	Default	Description
off	DINT		
val	DINT		

C.7.6 IOL_ISDU

One ISDU read or write through the master's ISDU module.

PLCopen-style: a rising edge on Execute starts one transfer; Done or Error then stays TRUE until Execute is dropped (or for one scan if it already was). Inputs are latched on the edge - changing them mid-transfer does nothing.

ONE TRANSFER PER MODULE AT A TIME. The module has one request slot. Several instances may share it, but the program must not start one while another is Busy. Consecutive ones are safe: the operation ID is taken from what is currently in the module, not from this instance.

ErrorID: 1 timeout - no reply with our operation ID 2 read answered with zero bytes - index/subindex refused 3 bad argument - Port not 1..8, Length not 0..32, Index not 0..65535 4 InBase/OutBase outside the process image

Inputs

Name	Type	Default	Description
Execute	BOOL		
InBase	DINT		%I byte offset of the ISDU module

Name	Type	Default	Description
OutBase	DINT		%Q byte offset of the ISDU module
Port	INT	1	1..8
Index	DINT		ISDU index, 0..65535
SubIndex	INT	0	
WriteReq	BOOL	FALSE	FALSE read, TRUE write
Length	INT	0	bytes of Data to write, 0..32
Data	ARRAY[1..32] OF BYTE		
Timeout	TIME	T#2s	

Outputs

Name	Type	Default	Description
Done	BOOL		
Busy	BOOL		
Error	BOOL		
ErrorID	INT		
RxLength	INT		bytes returned, 0..32
RxData	ARRAY[1..32] OF BYTE		

C.7.7 IOL_ReadParam

Read one ISDU parameter. The value is in RxData[1..RxLength], MSB first: a 16-bit parameter is RxData[1] * 256 + RxData[2].

Inputs

Name	Type	Default	Description
Execute	BOOL		
InBase	DINT		
OutBase	DINT		
Port	INT	1	
Index	DINT		
SubIndex	INT	0	
Timeout	TIME	T#2s	

Outputs

Name	Type	Default	Description
Done	BOOL		
Busy	BOOL		
Error	BOOL		
ErrorID	INT		
RxLength	INT		
RxData	ARRAY[1..32] OF BYTE		
Value	DINT		first 1/2/4 bytes as a big-endian number

C.7.8 IOL_WriteParam

Write one ISDU parameter. Put the bytes in Data[1..Length], MSB first - or, for a plain number, set Length to 1, 2 or 4 and pass it in Value.

Inputs

Name	Type	Default	Description
Execute	BOOL		
InBase	DINT		
OutBase	DINT		
Port	INT	1	
Index	DINT		
SubIndex	INT	0	
Length	INT	0	
Data	ARRAY[1..32] OF BYTE		
Value	DINT		used when UseValue: Length 1/2/4, MSB first
UseValue	BOOL	FALSE	
Timeout	TIME	T#2s	

Outputs

Name	Type	Default	Description
Done	BOOL		
Busy	BOOL		
Error	BOOL		
ErrorID	INT		

C.7.9 IOL_Events

Decode an "IO-Link Events/N" module and acknowledge it.

Each event is 4 bytes: type in the high nibble of byte 0 (1 message, 2 warning, 3 error), source port in the low nibble, byte 1 reserved, and a 16-bit event code MSB first. A slot with code 0 is empty.

Ack (rising edge) clears every event currently shown: the module's output has one "clear" bit per event and, at bit N, a toggle that must CHANGE for the master to act on the clear bits - so repeated acks keep working.

Master codes worth knowing: 16#1800 port communication lost, 16#1805 over-temperature, 16#1807 port power overcurrent, 16#1813 CQ overcurrent, 16#180E/16#180F UA under/over voltage, 16#180A/16#180B US under/over voltage. Other codes come from the device - see its IODD.

Inputs

Name	Type	Default	Description
InBase	DINT		
OutBase	DINT		
Count	INT	1	events in the module: 1, 2, 4, 6, 10, 15
Ack	BOOL		

Outputs

Name	Type	Default	Description
Active	INT		slots holding an event
AnyError	BOOL		
AnyWarn	BOOL		
EvType	ARRAY[1..15] OF INT		
EvPort	ARRAY[1..15] OF INT		
EvCode	ARRAY[1..15] OF DINT		
FirstCode	DINT		first slot holding an event, else 0
FirstPort	INT		
FirstType	INT		
Fault	BOOL		InBase/OutBase outside the image

Appendix D. Limits and capacities

The hard limits of the Zeus runtime and compiler, version 0.1.0.13. A limit marked *dynamic* has no fixed ceiling: the table grows with the program and is bounded only by the controller's memory. The IPC-CM400200 and IPC-CM500200 run the same software, so the figures are the same on both.

Note: On a real machine the first limit you meet is usually **scan time**, not any of the numbers below. The program is executed by an interpreter on a non-real-time Linux kernel; watch the scan statistics (Chapter 25) as the program grows and keep the execution time well inside the task interval.

D.1 Program and language

Item	Limit	Notes
Variables (globals, locals, FB members)	dynamic	
Function block instances (standard and user)	dynamic	
POUs in one download unit	256	Library blocks + FBs + methods + functions + programs, all counted
Members per STRUCT or per FB	128	
Parameters per FB or FUNCTION	64	
Call depth (nested FB/FUNCTION calls)	32	
Enumerator names, whole project	256	One namespace for all enumerations
STRING length	254 characters	STRING[n] , default 80
Array dimensions	3	
Operand stack depth	256	Only reached by extremely deep expressions
Compiler diagnostics reported per build	64	Further errors are not listed until the first ones are fixed
Breakpoints	32	
Programs in the task call list	16	

D.2 Process image and I/O

Item	Limit	Notes
IO devices (EtherCAT master, each Profinet or Modbus TCP device)	16	Each is one entry under IO Devices
%I and %Q size	sized from the configured devices	
%M size	4096 bytes by default	Configurable (process_image.marker_bytes)

Item	Limit	Notes
EtherCAT masters	1	
EtherCAT process image (in + out)	4096 bytes	
Modular EtherCAT couplers per segment	1	Module list, PDO assignment and start-up SDOs apply to the coupler at position 1
Profinet submodules per device	64	
Modbus TCP blocks per device	64	Up to 125 registers or 2000 bits per block
Signal-range scalings per device	32	

D.3 Communication and services

Item	Limit	Notes
Online monitor clients	32	IDE sessions and tools connected at once
Variable list sent to the IDE	12 MB per poll	Grows as needed; a list cut short is reported in the Messages panel
OPC UA instance folders	1,000	Further instances are published flat under Symbols
OPC UA users	8	
EtherNet/IP tag server clients	16	
Modbus TCP slave clients	8	
Controller users (monitor login)	16	

D.4 Diagnostics and alarms

Item	Limit	Notes
Alarms	10,000	
Trace channels	8	
Trace samples per channel	20,000	
Clock-memory pulses (user-defined)	8	In addition to the clock-memory byte

D.5 What is not a limit

- The number of variables and function block instances: they grow dynamically.
- The number of global variable lists and DUT files.
- The number of screens in the operator HMI (bounded by the controller's storage).

Appendix E. Keyboard shortcuts

Shortcuts are taken from the IDE's own key handling. Several keys mean different things depending on where the keyboard focus is — most importantly **F7**, which builds in the code editor but forces values in an online variable table.

E.1 Anywhere in the IDE

Keys	Action
Ctrl+S	Save the project
Ctrl+Shift+S	Save As
F7	Build (check) the active POU — when no online variable table is on screen
Ctrl+F7	Build all — when no online variable table is on screen
F8	Download the active program
Ctrl+Space	Browse IO (outside the code editor's body pane, where it opens completion instead)
Ctrl+F	Find — in a POU the find bar, elsewhere the tree/table search
F3 / Shift+F3	Next / previous find match
Esc	Close the open dialog

E.2 Code editor (Structured Text)

Keys	Action
Ctrl+Z	Undo (multi-step)
Ctrl+Y or Ctrl+Shift+Z	Redo
Ctrl+Space	Complete the word at the caret (body pane only; it does not also open Browse IO)
F2	Rename the variable at the caret everywhere it is used
Shift+F12	Where used (cross-reference of the name at the caret)
Alt+drag	Column (block) selection
Ctrl+C / Ctrl+X / Delete	Copy / cut / delete a column selection
Esc	Close the find bar / cancel a column selection
Tab / Shift+Tab	Indent / outdent

E.3 Online variable table

Keys	Action
Ctrl+F7	Write every prepared value once
F7	Force the prepared values of %-addressed points (held every scan)
Alt+F7	Release the forces on the rows shown
Enter (in a Prepared value box)	Move to the next box — Enter never writes
Esc (in a Prepared value box)	Clear that prepared value
Right-click a row	Write all prepared values (N) / Write only this one / Force / Release

Important: Writing to a running machine is always a deliberate action — Ctrl+F7, F7 or the buttons. Pressing Enter in a value box only prepares the value. While a variable table is on screen, F7, Ctrl+F7 and Alt+F7 act only on the table; they do not also start **Build** or **Build All**.

E.4 Debugging (while halted at a breakpoint)

Keys	Action
F5	Continue (only while halted; otherwise F5 reloads the page)
F10	Step over
F11	Step into
Shift+F11	Step out

Appendix F. Glossary

Term	Meaning
Alias	A name given to an IO channel in the device tree; the program uses the name and the IDE binds it to the address when it links the program
Boot application	The program the controller reloads at power-up — the last one downloaded
Call list	The ordered list of programs a task runs every scan (Task Configuration)
CONFIG	Run state for working on the hardware: the program is halted and no outputs are driven, but inputs are still read so channels can be watched as they are wired
Critical device	An IO device whose loss faults the controller
DEGRADED	Running, with a device not exchanging data — a real state, not a fault
DUT	Data Unit Type — a user data type: structure, enumeration or alias
ESI	EtherCAT Slave Information file (XML) describing an EtherCAT device and its modules
Force	Holding a %-addressed point at a value regardless of the program, until released
GSDML	The XML file describing a Profinet device and its modules
GVL	Global variable list — <code>globals.st</code> or a further list under <code>gv1/</code>
Inhibit	A device that is configured and addressed but deliberately not started
ISDU	IO-Link acyclic parameter access (index / subindex)
Jitter	Difference between the intended and the actual start of a scan
Link (the program)	Assembling DUTs, globals, aliases, library, blocks and programs into the one unit the controller compiles
Method	A procedure belonging to a function block, called by its name from inside that block (Chapter 14)
Minor fault	An operation that could not be done (for example division by zero); the scan continues
Online change	A download that keeps the running variables' values
POU	Program Organisation Unit — a program, function block, function or method
PRG / FB / FC	Program, function block, function
Prepared value	A value typed into the online variable table, sent only when you Write or Force
Process image	The controller's copy of IO: %I inputs, %Q outputs, %M markers
RETAIN	Data that survives a restart or a power cut
RIO	Remote IO — an IO coupler on a fieldbus
Scan	One cycle: read inputs, run the programs, write outputs
SFC	Sequential Function Chart
Signal range	Per-channel conversion of raw card counts to engineering values
Simulator	The real Zeus runtime running on the engineering PC

Term	Meaning
Start-up SDO	A CoE parameter written to an EtherCAT device while it starts
Step flag	Step.X (step active), Step.T (time in step)
Target	A controller (or the simulator) the IDE connects to
Watch / variable table	The online table of variables with live values
Write	Sending a value to a variable once; the program may change it on the next scan

Appendix G. Worked example projects

This appendix contains eight complete applications. Each one is a small, realistic piece of plant control. They are written the way a commissioning engineer would want to receive them: with the I/O list, the data types, the complete source of every POU, the reasons for the design choices, the Task Configuration entry, a test procedure for the simulator, and the usual variations. Use them as starting points, as training material, or as a reference for how the function-block library is meant to be wired.

Ready-made projects. You do not have to type them in: every example ships with the IDE as a project folder under `projects\examples\` (G1-TankFill, G2-Conveyor, G2b-ConveyorDOL, G3-TemperaturePID, G4-DutyStandbyPumps, G5-DeviceTable, G6-BatchMixerSFC, G7-MultiTask, G8-IOLinkSensor). Open one with **Project** → **Open an example project...** Each has one simulated IO device, *DeskIO*, large enough for every address the example declares, so it downloads to the simulator and runs as it is; G3 is set up for its desk test (the plant model in the task, the I/O as plain globals, G.3.7). On a controller, replace DeskIO with the real IO devices. The projects are generated from this appendix, and every release checks that each one compiles, downloads and runs. One more example, **F1-FactoryIO-Conveyor**, drives a 3D Factory I/O scene from the simulator (section 26.6a).

Every listing in this appendix was compiled with the controller's own compiler (`zeus-check --check`). Each example was compiled as one unit, in the order the IDE links a project: data types, then global variables, then the function-block library, then the project's own function blocks and programs.

WARNING: These examples show control logic, not safety functions. Emergency stops, guard interlocks, over-temperature cut-outs and high-high level trips that protect people or equipment belong in rated safety hardware, wired independently of the controller. Where an example reads such a signal, it does so only to stop the process in an orderly way and to tell the operator what happened.

G.0 How to use these examples

G.0.1 What each example contains

Part	Where it goes in the project	IDE tree node
I/O list	The IO device configuration (Chapter 7) and <code>globals.st</code>	IO Devices, Global Variables
Data types	<code>dut/<Name>.st</code> , one type per file	Data Types (DUT)
Global variables	<code>globals.st</code>	Global Variables
Function blocks and functions	<code>pou/<Name>.st</code> , one POU per file	POUs
Programs	<code>pou/<Name>.st</code> , one POU per file	POUs
Task Configuration	<code>controller.json</code> → task	Task Configuration

The file name above each listing (for example `pou/TankFill.st`) is the file the IDE writes when you create that object. One file holds exactly one POU or one data type. The IDE reuses a function-block file in other compiles only when it declares a single POU, so do not paste two POUs into one file.

G.0.2 Entering an example

To enter an example into a project:

1. Create a project for it, or make a copy of a scratch project with **Project** → **Save project as....** Do not paste examples into a project that runs a real machine.
2. For each data type: right-click **Data Types (DUT)** → **New DUT...**, enter the name, choose the kind (**Structure (STRUCT)**, **Enumeration** or **Alias**), click **Create**, replace the template text with the listing, and click **Save**. Click **Check** to compile the type on its own.
3. Open **Global Variables** and paste the global-variable listing. If the project already declares some of these names, merge them by hand; a name declared twice is a duplicate-symbol error.
4. For each function block, function and program: right-click **POUs** → **New POU...**, enter the name, choose the type (**Program (PRG)** — **ST**, **Function Block (FB)** — **ST**, **Function (FC)** — **ST** or **Program** — **SFC (sequence)**), and click **Create**. Replace the generated text with the listing.
5. Enter the Task Configuration shown in the example (Chapter 18). An example with one program needs only that program in MainTask's call list.
6. **Debug** → **Build All** (Ctrl+F7). The **Messages** pane must show 0 errors.

Note: The examples use the standard library in `lib/`: `SCALE`, `ALARM_AN`, `PID`, `MOTOR_DOL` and so on. A new project gets a copy of the library automatically. If **Library** in the tree is empty, add the library first (Chapter 19). G.8 also needs `70_ioLink.st`. Chapter 12 explains when you can add that file safely.

G.0.3 Addresses: real hardware or a desk test

The examples declare their I/O at fixed addresses (`LT101_Raw AT %IW2 : INT;`). Before a program is started, the controller checks that each address exists in the process image. If no IO device provides the address, the download is refused with a message like this:

'LT101_Raw' is at %I2, which needs %I to be at least 4 bytes, but it is 0 - add the IO device this address belongs to, or drop the AT clause

You can handle this in one of three ways:

Situation	What to do
The IO hardware is configured	Change the addresses to match the IO map of your modules (open each module in IO Devices to see where its channels are), or bind the names as channel aliases (Chapter 11).
You want to test with the simulator and the hardware is configured	Nothing. The simulator replaces each device with the <code>sim</code> driver and keeps its exact <code>%I</code> / <code>%Q</code> sizes, so every address stays valid.
You want a desk test with no hardware at all	Delete the <code>AT %...</code> part of each I/O declaration. The variables become ordinary globals. You can then write and force them directly, and the program cannot tell the difference.

`%M` addresses (the alarm bits in G.1) always exist: the marker area is 4096 bytes by default.

The analogue examples assume the input channel's **signal range** is set in the card configuration (Chapter 11). With the range set to **4–20 mA**, the process image holds microamps, 4000 to 20000. With **0–10 V** it holds millivolts, 0 to 10000. This is why `SCALE` is called with `raw_min := 4000`, `raw_max := 20000`. If you leave a channel at **Raw**, use the card's count range instead.

G.0.4 Testing in the simulator

The same procedure applies to every example. Each example then lists the values to force and the results to check.

1. Open the **Simulator** node. Clear **Generate moving input signals** so that inputs stay at 0 until you drive them. Click **Start / Restart**.
2. Click **Target...** and select the simulator.
3. **Online** → **Log in...** with an Engineer account.
4. **Online** → **Download & Upload** → **Download program** (F8). The status bar must read **RUNNING** and **IN SYNC**.
5. Open the program's **Variables** node, or **Global Variables**, and switch on **Online** → **Monitor**.
6. To drive an input, type a value in the **Prepared value** column. Click **Force values** (F7) to hold the value against the I/O. For a variable that has no **AT** address, you can also click **Write values** (Ctrl+F7). **Release force** (Alt+F7) returns the variable to its real value.
7. For fast signals such as timers, step times and PID output, use **Online** → **Trace / scope...** rather than watching the table.

Important: Release every force before you leave the controller. The amber **FORCE** indicator in the status bar shows that something is still forced, and clicking it releases everything.

G.0.5 Conventions used in the listings

- Inputs that signal a healthy condition are named `..._OK` and are wired normally-closed. **TRUE** means healthy, so a broken wire reads as a fault. The library's `TRIP_FIRST_OUT` uses the same convention.
- A fault **latches** and is cleared only by a **rising edge** on a reset. The library blocks behave the same way.
- Every block that integrates or differentiates has a `dt_ms` input. It is given a named constant equal to the interval of the task that runs the program. If you move the program to another task, change the constant too. Nothing detects a wrong `dt_ms`: the block keeps running with the wrong time base and behaves wrongly.
- Operator settings are declared `VAR_GLOBAL RETAIN` so that they survive a power cycle. Keep them in `globals.st`, not in a separate global variable list (Chapter 19 explains why).

G.1 Tank fill with level control, safety timeouts and alarms

G.1.1 Task

Tank T-101 is filled through an on/off inlet valve XV-101. The valve has open and closed limit switches. A 4–20 mA transmitter LT-101 measures the level over 0–100 %. The requirements are:

- In AUTO, after the operator presses START, keep the level between 40 % and 80 %: open the valve at or below 40 % and close it at or above 80 %. The two setpoints are adjustable from the HMI and are retained.
- Report level alarms HH 95 %, H 90 %, L 10 % and LL 5 %, each with hysteresis and a 2 s delay. Send the alarms to the controller's alarm engine.
- Stop filling immediately on any of these conditions: STOP pressed, selector moved out of AUTO, the independent high-high switch LSHH-101 tripped, the HH alarm, or a transmitter out of range (wire break).
- Detect a failed fill in two ways:
 - The level has not risen by at least 0.5 % within any 60 s while the valve is open (burst line, closed manual valve, empty supply).
 - The whole fill has taken longer than 30 minutes.

- Close the valve on these faults and hold them until the operator presses RESET.
- Supervise the valve. If it does not reach position within 10 s, or both limit switches are made at once, raise a fault.

G.1.2 I/O list

Address	Name	Type	Description
%IW2	LT101_Raw	INT	Level transmitter, range 4–20 mA (4000–20000 µA) = 0–100 %
%IX0.0	LSHH101_OK	BOOL	High-high level switch, NC. TRUE = level below the switch
%IX0.1	Auto_Sel	BOOL	Selector switch, TRUE = AUTO
%IX0.2	Start_PB	BOOL	Start push button, NO
%IX0.3	Stop_PB_OK	BOOL	Stop push button, NC. TRUE = not pressed
%IX0.4	Reset_PB	BOOL	Fault reset push button, NO
%IX0.5	XV101_ZSO	BOOL	Inlet valve open limit switch
%IX0.6	XV101_ZSC	BOOL	Inlet valve closed limit switch
%QX0.0	XV101_Sol	BOOL	Inlet valve solenoid, energise to open
%QX0.1	Horn	BOOL	Alarm horn
%QX0.2	Lamp_Filling	BOOL	Panel lamp: filling
%QX0.3	Lamp_Fault	BOOL	Panel lamp: fault, flashing
%MX100.0 – %MX101.0	ALM_...	BOOL	Alarm bits for the alarm engine (see G.1.6)

G.1.3 Global variables

globals.st :

```

(* ---- Tank T-101: field IO ----- *)
VAR_GLOBAL
    (* Inputs *)
    LT101_Raw      AT %IW2   : INT;    (* Level transmitter, channel range 4-20 mA = 4000..20000 uA *)
    LSHH101_OK     AT %IX0.0 : BOOL;    (* high-high level switch, NC: TRUE = Level below switch *)
    Auto_Sel       AT %IX0.1 : BOOL;    (* selector switch: TRUE = AUTO *)
    Start_PB       AT %IX0.2 : BOOL;    (* start push button, NO *)
    Stop_PB_OK     AT %IX0.3 : BOOL;    (* stop push button, NC: TRUE = not pressed *)
    Reset_PB       AT %IX0.4 : BOOL;    (* fault reset push button, NO *)
    XV101_ZSO      AT %IX0.5 : BOOL;    (* inlet valve open limit switch *)
    XV101_ZSC      AT %IX0.6 : BOOL;    (* inlet valve closed limit switch *)

    (* Outputs *)
    XV101_Sol      AT %QX0.0 : BOOL;    (* inlet valve solenoid, energise to open *)
    Horn           AT %QX0.1 : BOOL;    (* alarm horn *)
    Lamp_Filling   AT %QX0.2 : BOOL;    (* panel lamp: filling *)
    Lamp_Fault     AT %QX0.3 : BOOL;    (* panel lamp: fault, flashes until reset *)
END_VAR

(* ---- Alarm bits watched by the controller's alarm engine ----- *)
VAR_GLOBAL
    ALM_LT101_HH   AT %MX100.0 : BOOL;
    ALM_LT101_H    AT %MX100.1 : BOOL;
    ALM_LT101_L    AT %MX100.2 : BOOL;
    ALM_LT101_LL   AT %MX100.3 : BOOL;
    ALM_LT101_Signal AT %MX100.4 : BOOL;    (* transmitter out of range / wire break *)
    ALM_LSHH101    AT %MX100.5 : BOOL;    (* high-high switch tripped *)
    ALM_NoProgress  AT %MX100.6 : BOOL;    (* valve open but Level not rising *)
    ALM_FillTimeout AT %MX100.7 : BOOL;    (* fill took longer than the maximum *)
    ALM_XV101_Fault AT %MX101.0 : BOOL;    (* valve did not reach position *)
END_VAR

(* ---- Operator settings (written from the HMI / SCADA) ----- *)
VAR_GLOBAL RETAIN
    SP_FillStart_Pct : REAL := 40.0;    (* start filling at or below *)
    SP_FillStop_Pct  : REAL := 80.0;    (* stop filling at or above *)
    SP_MaxFillTime   : TIME := T#30m;   (* whole-fill safety timeout *)
    SP_ProgressWindow : TIME := T#60s;   (* Level must rise within this *)
    SP_MinRise_Pct   : REAL := 0.5;     (* ... by at least this much *)
END_VAR

```

G.1.4 Program

pou/TankFill.st :

```

PROGRAM TankFill
VAR
    (* Measurement *)
    Level      : SCALE;      (* LT-101 raw -> 0..100 % *)
    LevelAlm   : ALARM_AN;   (* HH / H / L / LL with hysteresis and delay *)
    Level_Pct  : REAL;

    (* Control *)
    FillDemand : HYSTERESIS; (* on at SP_FillStart, off at SP_FillStop *)
    Inlet       : VALVE_2POS; (* XV-101 with limit switches *)
    Running     : BOOL;      (* operator has started the tank in AUTO *)
    Filling     : BOOL;      (* valve commanded open *)
    Permissive  : BOOL;      (* nothing prevents filling *)

    (* Supervision *)
    tFillMax    : TON;       (* whole-fill timeout *)
    tWindow     : TON;       (* progress window *)
    WindowDone  : BOOL;
    LevelAtStart : REAL;     (* level at the start of the current window *)
    FaultLatched : BOOL;     (* any trip that needs a reset *)
    rReset      : R_TRIG;
    rStart      : R_TRIG;
    FaultLamp   : BLINK;
END_VAR

    (* ---- 1. Measurement ----- *)
    Level(raw := LT101_Raw, raw_min := 4000, raw_max := 20000,
          eu_min := 0.0, eu_max := 100.0, clamp := TRUE);
    Level_Pct := Level.eu;

    LevelAlm(pv := Level_Pct, hh := 95.0, h := 90.0, l := 10.0, ll := 5.0,
            hyst := 1.0, delay := T#2s, enable := NOT (Level.under OR Level.over));

    ALM_LT101_HH := LevelAlm.hh_alm;
    ALM_LT101_H  := LevelAlm.h_alm;
    ALM_LT101_L  := LevelAlm.l_alm;
    ALM_LT101_LL := LevelAlm.ll_alm;
    ALM_LT101_Signal := Level.under OR Level.over;

    (* ---- 2. Operator commands ----- *)
    rStart(CLK := Start_PB);
    rReset(CLK := Reset_PB);

    IF rStart.Q AND Auto_Sel AND Stop_PB_OK THEN
        Running := TRUE;
    END_IF;
    IF NOT Stop_PB_OK OR NOT Auto_Sel THEN
        Running := FALSE;
    END_IF;

    (* ---- 3. Trips: Latch here, clear only on a reset edge ----- *)
    IF NOT LSHH101_OK THEN
        ALM_LSHH101 := TRUE;
    END_IF;
    IF Inlet.fault THEN
        ALM_XV101_Fault := TRUE;
    END_IF;

    FaultLatched := ALM_LSHH101 OR ALM_NoProgress OR ALM_FillTimeout

```

```

        OR ALM_XV101_Fault;

IF rReset.Q THEN
    (* A reset clears only what has actually gone away. *)
    IF LSHH101_OK THEN ALM_LSHH101 := FALSE; END_IF;
    ALM_NoProgress := FALSE;
    ALM_FillTimeout := FALSE;
    ALM_XV101_Fault := FALSE;
END_IF;

(* ---- 4. Fill control ----- *)
Permissive := Running
    AND LSHH101_OK
    AND NOT ALM_LT101_Signal
    AND NOT LevelAlm.hh_alm
    AND NOT FaultLatched;

FillDemand(pv := Level_Pct, on_at := SP_FillStart_Pct, off_at := SP_FillStop_Pct,
    enable := Permissive);
Filling := FillDemand.q AND Permissive;

Inlet(open_cmd := Filling, interlock_ok := Permissive,
    fb_opened := XV101_ZSO, fb_closed := XV101_ZSC,
    travel_time := T#10s, has_limits := TRUE, reset := rReset.Q);

(* ---- 5. Safety timeouts ----- *)
(* The whole fill must finish within SP_MaxFillTime. *)
tFillMax(IN := Filling, PT := SP_MaxFillTime);
IF tFillMax.Q THEN
    ALM_FillTimeout := TRUE;
END_IF;

(* And the Level must keep rising: every SP_ProgressWindow it must have
    gained SP_MinRise_Pct. A burst pipe or a closed manual valve shows up
    here within a minute instead of at the end of a 30-minute timeout. *)
tWindow(IN := Inlet.is_open AND NOT WindowDone, PT := SP_ProgressWindow);
WindowDone := tWindow.Q;
IF WindowDone THEN
    IF (Level_Pct - LevelAtStart) < SP_MinRise_Pct THEN
        ALM_NoProgress := TRUE;
    END_IF;
    LevelAtStart := Level_Pct;
END_IF;
IF NOT Inlet.is_open THEN
    LevelAtStart := Level_Pct;
END_IF;

(* ---- 6. Outputs ----- *)
XV101_Sol := Inlet.out_open;
Lamp_Filling := Inlet.out_open;

FaultLamp(enable := FaultLatched OR ALM_LT101_Signal,
    on_time := T#500ms, off_time := T#500ms);
Lamp_Fault := FaultLamp.q;

Horn := FaultLatched OR LevelAlm.hh_alm OR LevelAlm.ll_alm;
END_PROGRAM

```

G.1.5 Design notes

- **Measurement before control.** `SCALE` checks the *raw* value against its range before it clamps. A wire break reads below 4 mA, which gives `under = TRUE` and not simply 0 %. The program treats that as `ALM_LT101_Signal` and blocks filling. It also disables `ALARM_AN`, so a dead transmitter does not additionally raise a false LL alarm.
- **Two layers of high-level protection.** The software HH alarm and the independent LSHH-101 switch both block filling. The switch trip is latched (`ALM_LSHH101`). The HH alarm is not latched: it clears when the level falls below 94 % (95 % minus the 1 % hysteresis). The switch must *also* be hard-wired to de-energise the solenoid if it is a protective function (see the WARNING at the start of this appendix).
- **HYSTERESIS as the controller.** With `off_at > on_at` the block works in the heating sense: it switches on at or below `on_at` and off at or above `off_at` . A tank started with the level between 40 % and 80 % therefore does not fill until the level falls to 40 %. That is the specified behaviour. G.1.9 shows how to fill immediately instead.
- **Two independent timeouts.** The whole-fill timeout (`tFillMax`) catches a very slow fill. The progress window (`tWindow`) catches a fill that has stopped, within one minute instead of thirty. The window restarts itself: `WindowDone` holds the timer's input low for one call, which resets it, so the timer measures consecutive windows. `LevelAtStart` is re-sampled at each window boundary and whenever the valve is not open.
- **The valve block does the valve supervision.** `VALVE_2POS` times the travel, detects both limits made at once (code 2) and a valve stuck between its limits (code 3), and latches its own fault. `reset := rReset.Q` passes it the same edge as the program's own reset.
- **Reset clears only what has gone.** `ALM_LSHH101` is cleared only while the switch is healthy. A reset pressed while the switch is still tripped does nothing, and filling stays blocked.
- **Alarm bits in %M.** The alarm engine watches bits, not expressions. The program decides *what* the alarm means, and the configuration decides *how* it is presented (G.1.6). `ALARM_AN` already provides the on-delay and the hysteresis, so the alarm-engine entries use no further delay.

G.1.6 Task Configuration and alarm configuration

Use MainTask at 10 ms with `TankFill` in its call list. None of the blocks used has a `dt_ms` input, so the program does not depend on the interval.

```
"task": {
  "period_ms": 10,
  "calls": [ "TankFill.st" ]
}
```

The alarm engine entries (Chapter 27). The byte offset is the `%M` byte and the bit is 0–7:

```

"alarms": [
  { "id": 101, "name": "LT101_HH",      "message": "T-101 level high-high",      "priority":
"critical", "area": "M", "offset": 100, "bit": 0 },
  { "id": 102, "name": "LT101_H",      "message": "T-101 level high",      "priority":
"high",     "area": "M", "offset": 100, "bit": 1 },
  { "id": 103, "name": "LT101_L",      "message": "T-101 level low",      "priority": "low",
"area": "M", "offset": 100, "bit": 2 },
  { "id": 104, "name": "LT101_LL",     "message": "T-101 level low-low",   "priority":
"high",     "area": "M", "offset": 100, "bit": 3 },
  { "id": 105, "name": "LT101_SIGNAL", "message": "LT-101 signal out of range", "priority":
"high",     "area": "M", "offset": 100, "bit": 4 },
  { "id": 106, "name": "LSHH101",      "message": "T-101 high-high switch tripped", "priority":
"critical", "area": "M", "offset": 100, "bit": 5, "latched": true },
  { "id": 107, "name": "T101_NOPROG",  "message": "T-101 filling but level not rising", "priority":
"high",     "area": "M", "offset": 100, "bit": 6 },
  { "id": 108, "name": "T101_TIMEOUT", "message": "T-101 fill time exceeded",      "priority":
"high",     "area": "M", "offset": 100, "bit": 7 },
  { "id": 109, "name": "XV101_FAULT",  "message": "XV-101 did not reach position", "priority":
"high",     "area": "M", "offset": 101, "bit": 0 }
]

```

G.1.7 Testing in the simulator

Follow G.0.4. Then force the inputs as below and check the results.

Step	Force	Expected result
1	LT101_Raw = 3000	ALM_LT101_Signal TRUE, Lamp_Fault flashing, ALARM_AN outputs all FALSE
2	LT101_Raw = 8800 (30 %), LSHH101_OK = TRUE, Stop_PB_OK = TRUE, Auto_Se1 = TRUE, XV101_ZSC = TRUE	Healthy, but not running
3	Start_PB TRUE, then FALSE	Running TRUE, XV101_So1 TRUE, Lamp_Filling TRUE
4	XV101_ZSC = FALSE, XV101_ZSO = TRUE within 10 s	Inlet.is_open TRUE, no valve fault
5	Leave LT101_Raw at 8800 for 60 s	ALM_NoProgress TRUE, valve closes, horn on
6	Reset_PB TRUE → FALSE; then step LT101_Raw up by 200 (≈1.25 %) every 30 s	Filling continues; LevelAtStart follows the level at each window boundary
7	LT101_Raw = 16800 (80 %)	FillDemand.q FALSE, valve closes
8	LT101_Raw = 19300 (≈95.6 %) for 2 s	ALM_LT101_HH TRUE, horn on
9	LSHH101_OK = FALSE	ALM_LSHH101 latched. A reset has no effect until LSHH101_OK is TRUE again
10	Leave XV101_ZSO FALSE for 10 s after an open command	ALM_XV101_Fault TRUE, Inlet.fault_code = 3

To test the 30-minute timeout without waiting for it, write `SP_MaxFillTime = T#20s` during the test and write it back afterwards.

G.1.8 What to watch

Level_Pct, FillDemand.q, Permissive, Inlet.out_open, Inlet.travelling, Inlet.fault_code, tWindow.ET, LevelAtStart and all the ALM_... bits. In a trace, Level_Pct against XV101_So1 shows the 40–80 % band clearly.

G.1.9 Variations

- **Fill now.** To start filling at once when START is pressed with the level inside the band, add a FillNow : BOOL that is set on rStart.Q and cleared when Level_Pct >= SP_FillStop_Pct. Then use Filling := (FillDemand.q OR FillNow) AND Permissive;.
- **Proportional inlet.** Replace HYSTERESIS and VALVE_2POS with a PID (level to 60 %) driving an analogue control valve, and keep the whole supervision section as it is.
- **Several tanks.** Put the body in a FUNCTION_BLOCK FB_TankFill whose inputs and outputs are the I/O points, with one instance per tank. The alarm bits then become outputs of the block, which the program copies to %M.
- **No limit switches.** Set has_limits := FALSE. VALVE_2POS then reports is_open after travel_time, and the progress window still detects a valve that did not open.

G.2 Conveyor with start/stop, jam detection and fault reset

G.2.1 Task

A single-speed belt conveyor with a photo-eye at its discharge. The requirements are:

- START runs the motor and the motor stays on (seal-in). STOP, the motor overload or a jam alarm stops it.
- The belt is considered *up to speed* 3 s after the motor starts. A lamp shows it.
- If the discharge eye is blocked for more than 2 s while the belt is at speed, it is a jam. Latch a jam alarm and stop the motor.
- RESET clears the jam alarm, but only once the eye is clear.

This is the same program shown in the ladder editor screenshots in Chapter 16, so the example gives the logic in both forms.

G.2.2 I/O list

Address	Name	Type	Description
%IX0.0	Start_PB	BOOL	Start push button, NO
%IX0.1	Stop_PB	BOOL	Stop push button, NO (see the Important note in G.2.5)
%IX0.2	Overload	BOOL	Motor overload relay tripped
%IX0.3	Jam_Sensor	BOOL	Discharge photo-eye, TRUE = blocked
%IX0.4	Reset_PB	BOOL	Jam reset push button, NO
%QX0.0	Motor	BOOL	Conveyor motor contactor
%QX0.1	Up_To_Speed	BOOL	Panel lamp: at speed
%QX0.2	Jam_Alarm	BOOL	Jam beacon

G.2.3 Global variables

globals.st :

```
(* ---- Conveyor CV-01: field IO ----- *)
VAR_GLOBAL
    Start_PB    AT %IX0.0 : BOOL;    (* start push button, NO          *)
    Stop_PB     AT %IX0.1 : BOOL;    (* stop push button, NO (see Important note) *)
    Overload    AT %IX0.2 : BOOL;    (* motor overload relay tripped      *)
    Jam_Sensor  AT %IX0.3 : BOOL;    (* photo-eye at the discharge: TRUE = blocked *)
    Reset_PB    AT %IX0.4 : BOOL;    (* jam reset push button, NO        *)

    Motor       AT %QX0.0 : BOOL;    (* conveyor motor contactor         *)
    Up_To_Speed AT %QX0.1 : BOOL;    (* panel lamp: running at speed     *)
    Jam_Alarm   AT %QX0.2 : BOOL;    (* panel lamp / beacon: jam         *)
END_VAR
```

G.2.4 Program (Structured Text)

pou/Conveyor.st :

```
PROGRAM Conveyor
VAR
    SpeedTimer : TON;    (* belt is considered up to speed 3 s after start *)
    JamTimer   : TON;    (* the eye must be blocked 2 s to count as a jam *)
END_VAR

(* Rung 1: Start / stop with seal-in. Stop, overload or a jam drops the motor. *)
Motor := (Start_PB OR Motor) AND NOT Stop_PB AND NOT Overload AND NOT Jam_Alarm;

(* Rung 2: Belt is up to speed 3 s after the motor starts. *)
SpeedTimer(IN := Motor, PT := T#3s);
Up_To_Speed := SpeedTimer.Q;

(* Rung 3: Jam: sensor blocked for 2 s while running latches the alarm. *)
JamTimer(IN := Up_To_Speed AND Jam_Sensor, PT := T#2s);
IF JamTimer.Q THEN
    Jam_Alarm := TRUE;
END_IF;

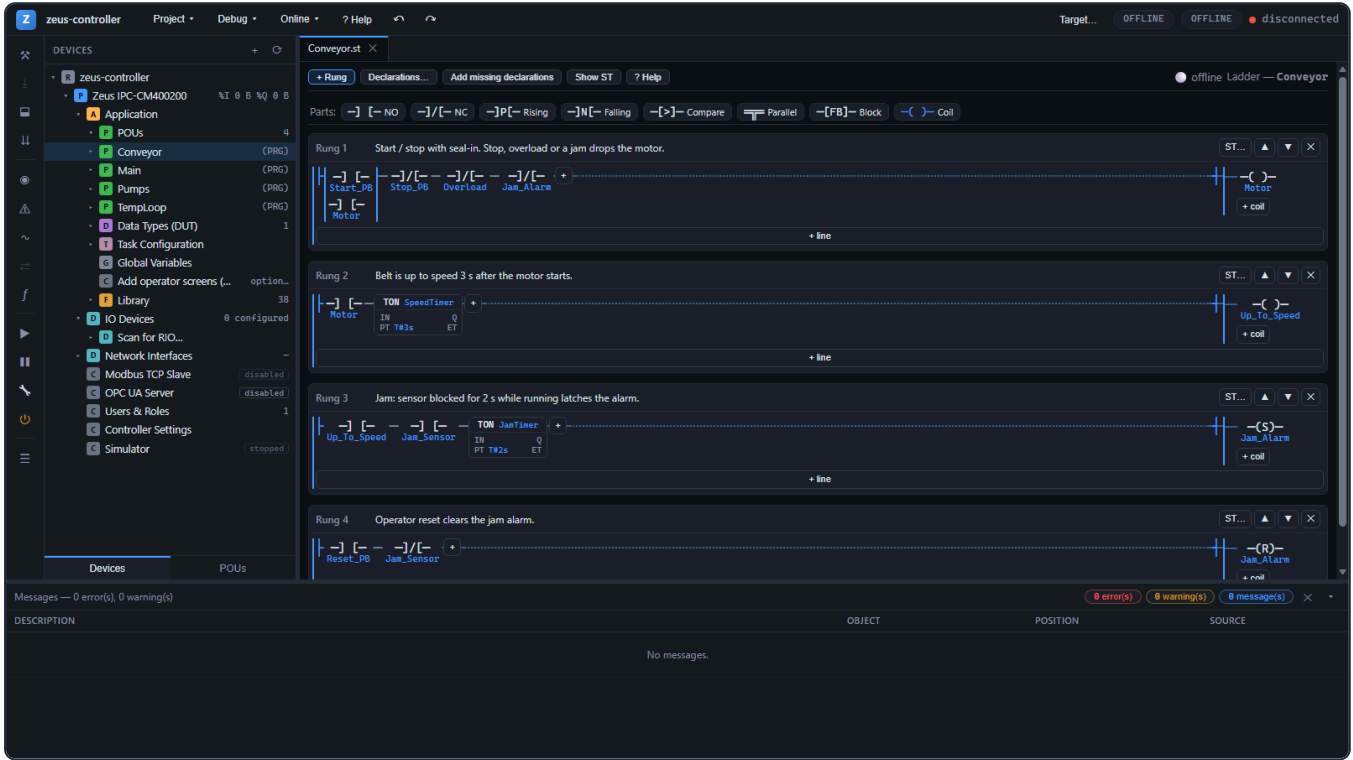
(* Rung 4: Operator reset clears the jam alarm - only once the eye is clear. *)
IF Reset_PB AND NOT Jam_Sensor THEN
    Jam_Alarm := FALSE;
END_IF;
END_PROGRAM
```

G.2.5 The same program in Ladder

Create the POU as **Program — Ladder** instead. The four rungs map one-to-one onto the four statements above:

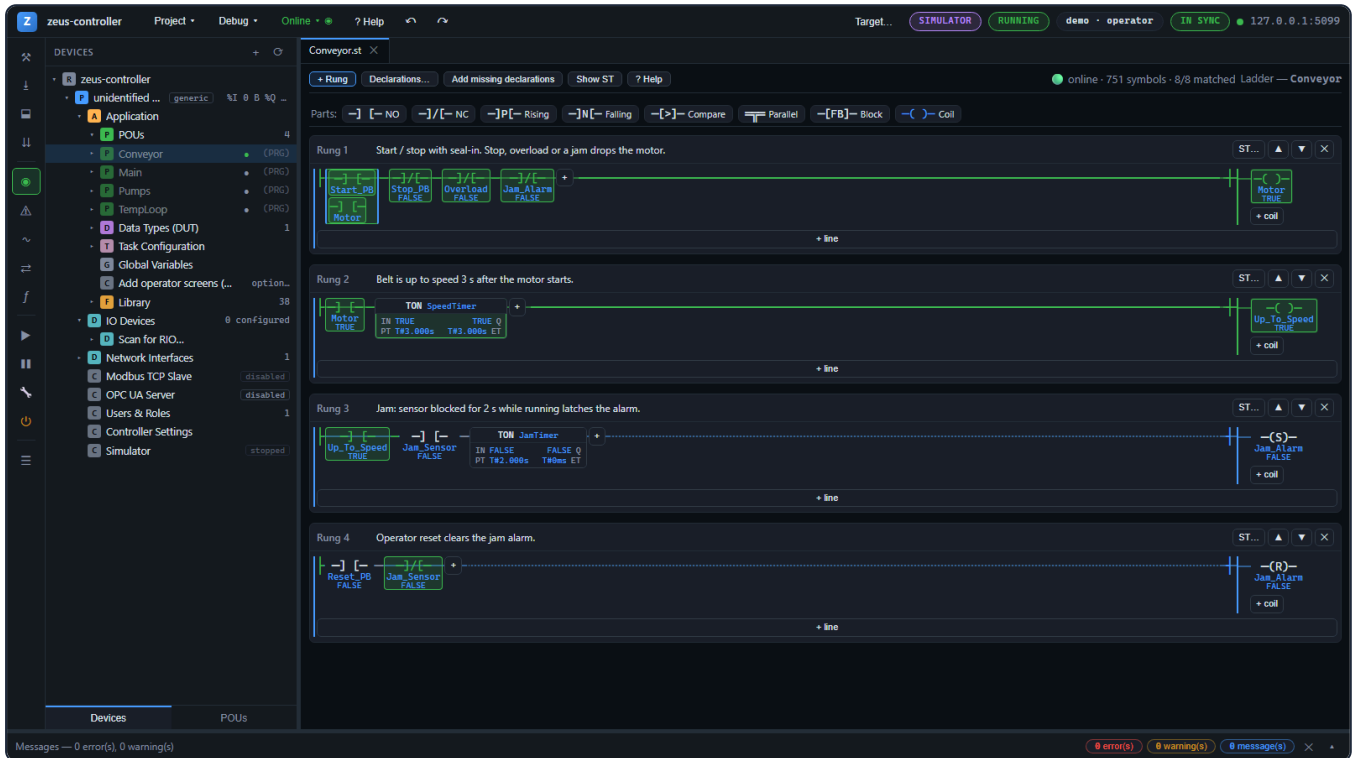
Rung	Contacts	Output
1	Start_PB (NO) in parallel with Motor (NO), in series with Stop_PB (NC), Overload (NC) and Jam_Alarm (NC)	Motor coil
2	Motor (NO) → TON SpeedTimer , PT = T#3s	Up_To_Speed coil

Rung	Contacts	Output
3	Up_To_Speed (NO), Jam_Sensor (NO) → TON JamTimer , PT = T#2s	Jam_Alarm set coil (S)
4	Reset_PB (NO), Jam_Sensor (NC)	Jam_Alarm reset coil (R)



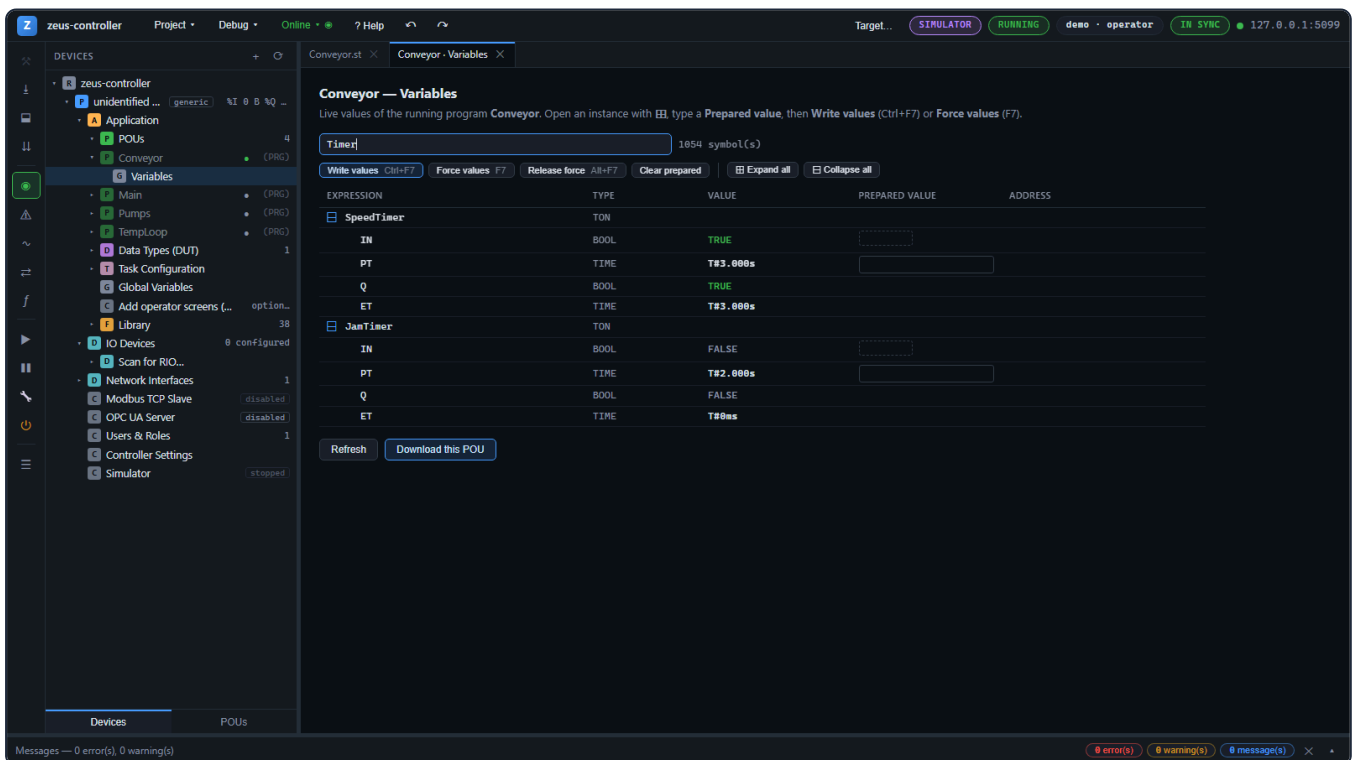
Conveyor program in the ladder editor

Online, the ladder view shows power flow and live timer values:



Conveyor program online, with live power flow

The program's **Variables** node shows both timer instances expanded:



Online variable table with SpeedTimer and JamTimer expanded

Note: A ladder POU is stored as an ST file. The diagram is held in a (*@LADDER@*) header comment, and the generated ST below it is what compiles. The generated text is equivalent to the listing in G.2.4. Click **Show ST** in the ladder editor to see it. Edit the declarations in the ladder editor's **Declarations...** pane, not in the text (Chapter 16).

Important: The example follows the screenshots and uses a *normally-open* stop button, read through an NC contact. A broken wire on that input can never stop the conveyor. On a real machine, wire the stop button NC, read it as `Stop_OK` through an NO contact, and wire the emergency stop through a safety relay independent of the controller. G.2.8 shows the fail-safe version.

G.2.6 Task Configuration

Use MainTask at 10 ms with `Conveyor` in its call list. The timers are `TON` instances, which measure real time, so the program does not depend on the interval.

```
"task": { "period_ms": 10, "calls": [ "Conveyor.st" ] }
```

G.2.7 Testing in the simulator

Step	Force / write	Expected result
1	<code>Start_PB</code> TRUE then FALSE	<code>Motor</code> TRUE and stays TRUE (seal-in). <code>SpeedTimer.ET</code> counts up
2	Wait 3 s	<code>Up_To_Speed</code> TRUE
3	<code>Jam_Sensor</code> TRUE for 1 s, then FALSE	<code>JamTimer.ET</code> reaches about 1 s and falls back. No alarm
4	<code>Jam_Sensor</code> TRUE for more than 2 s	<code>Jam_Alarm</code> TRUE, <code>Motor</code> FALSE, <code>Up_To_Speed</code> FALSE
5	<code>Reset_PB</code> TRUE while <code>Jam_Sensor</code> is still TRUE	<code>Jam_Alarm</code> stays TRUE
6	<code>Jam_Sensor</code> FALSE, then <code>Reset_PB</code> TRUE → FALSE	<code>Jam_Alarm</code> FALSE. The motor does <i>not</i> restart until START is pressed
7	While running: <code>Overload</code> TRUE	<code>Motor</code> FALSE at once

G.2.8 Variation: fail-safe inputs and a supervised starter

This version uses NC stop and overload inputs, a contactor feedback, and `MOTOR_DOL`. `MOTOR_DOL` adds the start/stop feedback timeout and the welded-contactor check (running without a command). Reset is edge-triggered, so a jammed reset button cannot hold faults clear.

`globals.st` :

```
VAR_GLOBAL
    Start_PB      AT %IX0.0 : BOOL;    (* NO *)
    Stop_OK       AT %IX0.1 : BOOL;    (* stop button wired NC: TRUE = not pressed *)
    Overload_OK   AT %IX0.2 : BOOL;    (* overload relay NC contact: TRUE = healthy *)
    Jam_Sensor    AT %IX0.3 : BOOL;
    Reset_PB      AT %IX0.4 : BOOL;
    Motor_Fb      AT %IX0.5 : BOOL;    (* contactor auxiliary contact *)

    Motor         AT %QX0.0 : BOOL;
    Up_To_Speed   AT %QX0.1 : BOOL;
    Jam_Alarm     AT %QX0.2 : BOOL;
    Motor_Fault   AT %QX0.3 : BOOL;
END_VAR
```

pou/ConveyorDOL.st :

```

PROGRAM ConveyorDOL
VAR
    Drive      : MOTOR_DOL;
    RunReq     : BOOL;      (* seal-in of the operator's start request *)
    SpeedTimer : TON;
    JamTimer   : TON;
    rReset     : R_TRIG;
END_VAR

rReset(CLK := Reset_PB);

RunReq := (Start_PB OR RunReq) AND Stop_OK AND NOT Jam_Alarm AND NOT Drive.fault;

Drive(start := RunReq, stop := NOT Stop_OK, interlock_ok := NOT Jam_Alarm,
    fb_running := Motor_Fb, ext_trip := NOT Overload_OK,
    fb_timeout := T#2s, reset := rReset.Q, dt_ms := 10);
Motor      := Drive.run_cmd;
Motor_Fault := Drive.fault;

SpeedTimer(IN := Drive.running, PT := T#3s);
Up_To_Speed := SpeedTimer.Q;

JamTimer(IN := Up_To_Speed AND Jam_Sensor, PT := T#2s);
IF JamTimer.Q THEN Jam_Alarm := TRUE; END_IF;
IF rReset.Q AND NOT Jam_Sensor THEN Jam_Alarm := FALSE; END_IF;
END_PROGRAM

```

In the simulator, also force `Motor_Fb` TRUE within 2 s of `Motor` going TRUE. If you do not, `Drive.fault` sets with `fault_code = 1` (no start confirmation). This is the first thing to test on site: the program and the contactor wiring must agree.

Other variations:

- **Restart after a jam.** Some lines restart by themselves after the reset. Replace the seal-in with a state that remembers "was running" and restart on `rReset.Q AND WasRunning`, behind a 5 s horn warning. Do this only if the risk assessment allows an automatic restart.
- **Line of conveyors.** Give each conveyor a `Permit` input that is `Up_To_Speed` of the conveyor downstream. Starting then cascades upstream, and a jam downstream stops everything that feeds it.

G.3 Temperature control with PID and auto-tune

G.3.1 Task

One heating zone of an oven. The requirements are:

- Measure the temperature with a 4–20 mA transmitter TT-201 (0–150 °C). Filter the noise with a 2 s first-order lag.
- Control it with a PID to an operator setpoint (default 65 °C) by driving a thyristor power controller through a 0–10 V output.
- Provide auto/manual with bumpless transfer in both directions.
- Provide a relay auto-tune the engineer can start from the HMI. The tuned gains are applied only when the engineer accepts them.

- Trip and latch on a transmitter out of range, the independent over-temperature thermostat TSHH-201, or a software high-high (default 120 °C). A trip sets the output to 0 % and drops the heater contactor. Reset is allowed only when the temperature is 5 °C below the trip point.

G.3.2 I/O list

Address	Name	Type	Description
%IW10	TT201_Raw	INT	Temperature, range 4–20 mA = 0–150 °C
%IX0.0	TSHH201_OK	BOOL	Over-temperature thermostat, NC
%IX0.1	Heater_Enable	BOOL	Heater contactor healthy and door closed
%QW4	Heater_Out	INT	Power demand to the thyristor unit, range 0–10 V (0–10000 mV)
%QX0.0	Heater_On	BOOL	Heater main contactor

G.3.3 Global variables

globals.st :

```
(* ---- Oven zone TIC-201: field IO ----- *)
VAR_GLOBAL
    TT201_Raw    AT %IW10 : INT;    (* 4-20 mA transmitter, 0..150 degC; channel range 4-20 mA *)
    TSHH201_OK   AT %IX0.0 : BOOL;  (* independent over-temperature thermostat, NC *)
    Heater_Enable AT %IX0.1 : BOOL; (* heater contactor healthy / door closed *)

    Heater_Out   AT %QW4 : INT;    (* 0-10 V to the thyristor power controller; range 0-10 V *)
    Heater_On     AT %QX0.0 : BOOL; (* heater main contactor *)
END_VAR

(* ---- Operator interface (HMI / SCADA) ----- *)
VAR_GLOBAL
    TIC201_Manual : BOOL;          (* TRUE = manual output *)
    TIC201_ManOut  : REAL;         (* manual output, % *)
    TIC201_Tune    : BOOL;         (* rising edge starts an auto-tune *)
    TIC201_TuneAbort : BOOL;
    TIC201_Accept  : BOOL;         (* rising edge copies the tuned gains into the PID *)
    TIC201_PV      : REAL;         (* filtered temperature, degC (for display) *)
    TIC201_Out     : REAL;         (* output actually sent, % *)
    TIC201_Trip    : BOOL;         (* over-temperature or sensor trip, latched *)
    TIC201_Reset   : BOOL;         (* rising edge resets the trip *)
END_VAR

VAR_GLOBAL RETAIN
    TIC201_SP      : REAL := 65.0; (* setpoint, degC *)
    TIC201_Kp      : REAL := 2.0;
    TIC201_Ti      : REAL := 120.0; (* s *)
    TIC201_Td      : REAL := 0.0;  (* s *)
    TIC201_TripAt   : REAL := 120.0; (* software high-high, degC *)
END_VAR
```

G.3.4 Program

pou/TempLoop.st :

```

PROGRAM TempLoop
VAR CONSTANT
    DT_MS      : DINT := 100;      (* MUST equal this program's task interval *)
END_VAR
VAR
    Scale      : SCALE;
    Filter     : FILTER_PT1;
    Loop       : PID;
    Tuner      : PID_AutoTune;
    rAccept    : R_TRIG;
    rReset     : R_TRIG;
    SensorBad  : BOOL;
    Tuning     : BOOL;
    OutPct     : REAL;
END_VAR

(* ---- Measurement: scale, then filter ----- *)
Scale(raw := TT201_Raw, raw_min := 4000, raw_max := 20000,
      eu_min := 0.0, eu_max := 150.0, clamp := TRUE);
SensorBad := Scale.under OR Scale.over;

Filter(x := Scale.eu, tau_ms := 2000, dt_ms := DT_MS, reset := FALSE);
TIC201_PV := Filter.y;

(* ---- Trip: over-temperature or a broken sensor ----- *)
IF SensorBad OR NOT TSHH201_OK OR TIC201_PV >= TIC201_TripAt THEN
    TIC201_Trip := TRUE;
END_IF;
rReset(CLK := TIC201_Reset);
IF rReset.Q AND NOT SensorBad AND TSHH201_OK AND TIC201_PV < TIC201_TripAt - 5.0 THEN
    TIC201_Trip := FALSE;
END_IF;

(* ---- Auto-tune ----- *)
Tuner(Start := TIC201_Tune AND NOT TIC201_Trip AND NOT TIC201_Manual,
      Abort := TIC201_TuneAbort OR TIC201_Trip,
      pv := TIC201_PV, sp := TIC201_SP, reverse := FALSE,
      out_bias := 30.0, relay_amp := 15.0, relay_hyst := 0.5,
      rule := 1, settle_cycles := 2, tune_cycles := 4,
      timeout_s := 3600.0, dt_ms := DT_MS);
Tuning := Tuner.tuning;

(* Tuned gains are applied only when the engineer accepts them. *)
rAccept(CLK := TIC201_Accept);
IF rAccept.Q AND Tuner.done THEN
    TIC201_Kp := Tuner.Kp;
    TIC201_Ti := Tuner.Ti_s;
    TIC201_Td := Tuner.Td_s;
END_IF;

(* ---- PID ----- *)
(* While tuning, the PID is held in manual at the tuner's output, so that
   the return to automatic afterwards is bumpless. *)
Loop(pv := TIC201_PV, sp := TIC201_SP,
     kp := TIC201_Kp, ti_s := TIC201_Ti, td_s := TIC201_Td,
     dt_ms := DT_MS, out_min := 0.0, out_max := 100.0, reverse := FALSE,
     man := TIC201_Manual OR Tuning,
     man_out := SEL(Tuning, TIC201_ManOut, Tuner.out),
     enable := NOT TIC201_Trip AND Heater_Enable);

```



```

IF Tuning THEN
    OutPct := Tuner.out;
ELSE
    OutPct := Loop.out;
END_IF;
IF TIC201_Trip OR NOT Heater_Enable THEN
    OutPct := 0.0;
END_IF;

(* In manual the operator's value is tracked back so a return to manual
   also starts from where automatic left off. *)
IF NOT TIC201_Manual THEN
    TIC201_ManOut := OutPct;
END_IF;

(* ---- Outputs ----- *)
TIC201_Out := OutPct;
Heater_On  := NOT TIC201_Trip AND Heater_Enable;
Heater_Out := REAL_TO_INT(OutPct * 100.0);    (* 0..100 % -> 0..10000 mV *)
END_PROGRAM

```

G.3.5 Design notes

- **dt_ms in one place.** `FILTER_PT1`, `PID` and `PID_AutoTune` all integrate over time and need the scan interval. The program runs in a 100 ms task (G.3.6), so `DT_MS` is 100 and every block receives the same constant.
- **Why 100 ms and not 10 ms.** An oven's time constants are minutes. Running the loop ten times faster adds no control quality. It adds load, and the derivative becomes noisier because the change of PV per scan is smaller relative to the noise.
- **Filter first, then control.** `FILTER_PT1` with `tau_ms := 2000` removes transmitter noise before it reaches the PID. The derivative acts on the (filtered) measurement, not on the error, so a setpoint change does not make the output jump. That is a property of the library `PID`.
- **Bumpless transfer.**
 - In manual, `PID` tracks: it back-calculates its integral from `man_out`. The switch back to automatic therefore starts from the manual output.
 - Going the other way, the program copies the automatic output into `TIC201_ManOut` while in automatic. Switching to manual therefore holds the last output instead of jumping to a stale manual value.
- **Auto-tune without a bump.** During a tune, the PID is held in manual with `man_out` equal to the tuner's output (`SEL(Tuning, TIC201_ManOut, Tuner.out)`). When the tune ends, `Tuning` falls and the PID returns to automatic starting from the tuner's last output.
- **Accept, do not apply.** `PID_AutoTune` proposes gains (`Kp`, `Ti_s`, `Td_s`, using the Tyreus–Luyben rule with `rule := 1`). An engineer reviews them before they are copied. Tuning results on a real plant are sometimes wrong, for example when a door was opened during the test.
- **Relay parameters.**
 - `out_bias` is the output that roughly holds the setpoint (read it in manual beforehand).
 - `relay_amp` is how far each side of that the relay switches. Choose it large enough to move the temperature clearly above the noise, and small enough to be safe.
 - `relay_hyst` must be larger than the remaining noise on the filtered PV.
 - `timeout_s` is generous because a slow oven needs several full oscillations.

- **Trip.** The software high-high and the sensor check are latched in the program. The hardware thermostat must *also* cut the heater power independently.

G.3.6 Task Configuration

Run `TempLoop` in its own 100 ms task. `MainTask` stays at 10 ms for everything else in the project.

To create the task:

1. Open **Task Configuration** → **Task Groups** tab.
2. Click + **Add task**. Enter the name `SlowLoop`, **Interval (ms)** `100` and **Offset (scans)** `0`. Tick `TempLoop` under **POUs to run in this task** and click **Add task**.
3. Click **Apply tasks**, then restart the runtime or use **Download all** so the new schedule takes effect.

The result in `controller.json`:

```
"task": {
  "period_ms": 10,
  "calls": [ "TempLoop.st" ],
  "groups": [ { "name": "SlowLoop", "period_ms": 100 } ],
  "programs": [ { "name": "TempLoop", "period_ms": 100 } ]
}
```

On the **Properties** tab, the **Running as** column must read `every 10 scans (100 ms)` for `TempLoop` once the controller runs it. If it reads `every scan`, the PID is running ten times faster than its `dt_ms` says, and it is badly detuned without any other symptom.

G.3.7 Testing in the simulator: a plant model

A PID cannot be tested against a forced input, because nothing responds to its output. For a desk test, add this plant model. It is a first-order oven with a 3 s transport delay. To use it:

- Remove the `AT %...` clauses from the five I/O declarations in `globals.st` (see G.0.3).
- Put `PlantSim` in the `SlowLoop` task *before* `TempLoop`.

`pou/PlantSim.st`:

```
(* DESK TEST ONLY - a crude oven model so the loop and the auto-tuner have
something to control in the simulator. Needs the IO globals WITHOUT their
AT clauses. Never download this to a controller with real IO. *)
PROGRAM PlantSim
VAR CONSTANT
    SIM_DT_MS : DINT := 100;
END_VAR
VAR
    OvenTemp : REAL := 20.0;      (* degC *)
    Ambient   : REAL := 20.0;
    Gain      : REAL := 1.3;      (* degC above ambient per % heater *)
    Tau_s     : REAL := 60.0;     (* thermal time constant *)
    Delay     : ARRAY[1..30] OF REAL; (* 3 s transport delay at 100 ms *)
    Idx       : INT := 1;
    Delayed   : REAL;
    HeatPct   : REAL;
END_VAR

TSHH201_OK := TRUE;
Heater_Enable := TRUE;

HeatPct := INT_TO_REAL(Heater_Out) / 100.0;      (* mV back to % *)
IF NOT Heater_On THEN HeatPct := 0.0; END_IF;

(* transport delay: read the oldest sample, then overwrite it *)
Delayed := Delay[Idx];
Delay[Idx] := HeatPct;
Idx := Idx + 1;
IF Idx > 30 THEN Idx := 1; END_IF;

(* first-order lag towards ambient + gain * heat *)
OvenTemp := OvenTemp + (Ambient + Gain * Delayed - OvenTemp)
               * DINT_TO_REAL(SIM_DT_MS) / 1000.0 / Tau_s;

TT201_Raw := REAL_TO_INT(4000.0 + OvenTemp / 150.0 * 16000.0);
END_PROGRAM
```

WARNING: PlantSim writes the temperature input. Delete it, and restore the AT clauses, before the project goes near a controller with real I/O.

Test procedure:

Step	Write	Expected result
1	Nothing	TIC201_PV rises from 20 °C towards 65 °C. Trace TIC201_PV , TIC201_SP and TIC201_Out
2	TIC201_Manual TRUE	TIC201_Out holds its last value (no bump). Loop.out tracks it
3	TIC201_ManOut = 30	The output steps to 30 %. The PV settles near 20 + 1.3 × 30 = 59 °C
4	TIC201_Manual FALSE	The output continues from 30 % and the PV moves back to the setpoint
5	TIC201_Tune TRUE → FALSE	Tuner.tuning TRUE. The output alternates between 15 % and 45 %. Tuner.progress counts cycles

Step	Write	Expected result
6	Wait for <code>Tuner.done</code>	<code>Tuner.Ku</code> , <code>Tuner.Tu</code> , <code>Tuner.Kp</code> , <code>Tuner.Ti_s</code> , <code>Tuner.Td_s</code> are shown. Control returns to the PID
7	<code>TIC201_Accept</code> TRUE → FALSE	<code>TIC201_Kp/Ti/Td</code> take the tuned values
8	<code>TIC201_SP</code> = 80	Compare the response with the one in step 1
9	<code>TIC201_TripAt</code> = 70	<code>TIC201_Trip</code> latches. Output 0, <code>Heater_On</code> FALSE

If `Tuner.failed` is TRUE, either the timeout expired before `tune_cycles` clean oscillations, or the measured amplitude was zero. Increase `relay_amp` , or reduce `relay_hyst` .

G.3.8 Variations

- **Relay or SSR heater.** Send `OutPct` to a `PWM` block (`period_ms` := 10000 for a contactor, 1000–2000 for an SSR, `dt_ms` := `DT_MS`) and drive a digital output with `PWM.q` . For a contactor, set `min_on_ms` so that very short pulses are dropped.
- **Heat and cool.** Run the PID over 0–100 % , feed it to `SPLIT_RANGE` (`split_pct` := 50), and drive the cooling valve from `a_out` and the heater from `b_out` . With `split_pct` := 50 , a PID output of 50 % means neither heating nor cooling.
- **Setpoint ramp.** Pass the operator setpoint through `RAMP` (`rate_per_s` := 0.05 is 3 °C/min) and give the PID `RAMP.out` , so that a product is not thermally shocked.
- **Several zones.** Wrap the body in a `FUNCTION_BLOCK_FB_Zone` . Each zone needs its own `PID` and `PID_AutoTune` instance. Tune the zones one at a time: neighbouring zones heat each other, and two relay tests running together disturb each other's oscillation.

G.4 Duty/standby pump set with alternation and run-hour balancing

G.4.1 Task

A sump is emptied by two identical pumps, P-301A and P-301B. The requirements are:

- Start the duty pump at 70 % level and stop it at 30 %.
- Only one pump runs normally. The duty role alternates so that both pumps accumulate similar running hours. The role changes only when the pumps are stopped, and only when the imbalance exceeds 50 h.
- If the duty pump faults, or is taken out of AUTO, the standby takes over at once.
- At 90 % level, the standby also runs (assist) until the level is back at 70 %.
- Supervise each pump for start and stop feedback, overload and a welded contactor.
- Keep retained run-hour meters with a 4000 h service interval, acknowledged from the HMI.
- Give a common alarm on any pump fault, no pump available, transmitter failure, or "assist level reached and nothing running for 10 s".

G.4.2 I/O list

Address	Name	Type	Description
%IW2	LT301_Raw	INT	Sump level, range 4–20 mA = 0–100 %

Address	Name	Type	Description
%IX0.0 / %IX0.3	P301A_Auto / P301B_Auto	BOOL	Local selector in AUTO
%IX0.1 / %IX0.4	P301A_RunFb / P301B_RunFb	BOOL	Contactor auxiliary contact
%IX0.2 / %IX0.5	P301A_OL_OK / P301B_OL_OK	BOOL	Overload relay, NC, TRUE = healthy
%IX0.6	Reset_PB	BOOL	Fault reset
%QX0.0 / %QX0.1	P301A_Run / P301B_Run	BOOL	Contactor
%QX0.2 / %QX0.3	P301A_Fault / P301B_Fault	BOOL	Fault lamp
%QX0.4	Station_Alarm	BOOL	Common alarm beacon

G.4.3 Global variables

globals.st :

```
(* ---- Sump pump station P-301A / P-301B: field IO ----- *)
VAR_GLOBAL
    LT301_Raw      AT %IW2      : INT;      (* sump Level, 4-20 mA = 0..100 % *)
    P301A_Auto     AT %IX0.0    : BOOL;     (* A: local selector in AUTO *)
    P301A_RunFb    AT %IX0.1    : BOOL;     (* A: contactor auxiliary *)
    P301A_OL_OK    AT %IX0.2    : BOOL;     (* A: overload relay NC, TRUE = healthy *)
    P301B_Auto     AT %IX0.3    : BOOL;
    P301B_RunFb    AT %IX0.4    : BOOL;
    P301B_OL_OK    AT %IX0.5    : BOOL;
    Reset_PB       AT %IX0.6    : BOOL;

    P301A_Run      AT %QX0.0    : BOOL;     (* A: contactor *)
    P301B_Run      AT %QX0.1    : BOOL;
    P301A_Fault    AT %QX0.2    : BOOL;     (* A: fault lamp *)
    P301B_Fault    AT %QX0.3    : BOOL;
    Station_Alarm  AT %QX0.4    : BOOL;     (* common alarm to the beacon *)
END_VAR

VAR_GLOBAL
    P301A_Service_Ack : BOOL;      (* HMI: service done on A *)
    P301B_Service_Ack : BOOL;
    Duty_Is_B         : BOOL;      (* HMI: which pump is duty now *)
    Level_Pct         : REAL;
END_VAR

VAR_GLOBAL RETAIN
    SP_Start_Pct      : REAL := 70.0;  (* duty pump starts *)
    SP_Stop_Pct       : REAL := 30.0;  (* duty pump stops *)
    SP_Assist_Pct     : REAL := 90.0;  (* standby joins in (inflow above one pump's rate) *)
    SP_Balance_h      : REAL := 50.0;  (* run-hour imbalance that moves the duty *)
END_VAR
```

G.4.4 Program

pou/PumpStation.st :

```

PROGRAM PumpStation
VAR CONSTANT
    DT_MS      : DINT := 10;          (* task interval of this program *)
END_VAR
VAR
    Level      : SCALE;
    Demand     : HYSTERESIS;          (* pump-down: on when high, off when Low *)
    Assist     : HYSTERESIS;
    Alt        : PUMP_ALT2;
    MotA       : MOTOR_DOL;
    MotB       : MOTOR_DOL;
    rReset     : R_TRIG;
    A_Avail    : BOOL;
    B_Avail    : BOOL;
    tNoPump    : TON;                 (* assist Level reached and nothing running *)
END_VAR
VAR RETAIN
    HoursA     : RUNTIME_HRS;         (* retained: the life of each pump survives a restart *)
    HoursB     : RUNTIME_HRS;
END_VAR

Level(raw := LT301_Raw, raw_min := 4000, raw_max := 20000,
      eu_min := 0.0, eu_max := 100.0, clamp := TRUE);
Level_Pct := Level.eu;

(* on_at > off_at selects the "cooling" sense: on when HIGH. *)
Demand(pv := Level_Pct, on_at := SP_Start_Pct, off_at := SP_Stop_Pct,
      enable := NOT (Level.under OR Level.over));
Assist(pv := Level_Pct, on_at := SP_Assist_Pct, off_at := SP_Start_Pct,
      enable := NOT (Level.under OR Level.over));

rReset(CLK := Reset_PB);

(* A pump is available when it is in AUTO and not faulted. *)
A_Avail := P301A_Auto AND NOT MotA.fault;
B_Avail := P301B_Auto AND NOT MotB.fault;

Alt(demand := Demand.q, a_avail := A_Avail, b_avail := B_Avail,
    a_hours := HoursA.total_h, b_hours := HoursB.total_h,
    lead_gap_h := SP_Balance_h, overlap := T#2s, dt_ms := DT_MS);
Duty_Is_B := Alt.lead_is_b;

MotA(start := Alt.a_run OR (Assist.q AND A_Avail),
    stop := NOT P301A_Auto,
    interlock_ok := TRUE,
    fb_running := P301A_RunFb, ext_trip := NOT P301A_OL_OK,
    fb_timeout := T#3s, reset := rReset.Q, dt_ms := DT_MS);
MotB(start := Alt.b_run OR (Assist.q AND B_Avail),
    stop := NOT P301B_Auto,
    interlock_ok := TRUE,
    fb_running := P301B_RunFb, ext_trip := NOT P301B_OL_OK,
    fb_timeout := T#3s, reset := rReset.Q, dt_ms := DT_MS);

HoursA(running := MotA.running, service_h := 4000.0,
    ack := P301A_Service_Ack, dt_ms := DT_MS);
HoursB(running := MotB.running, service_h := 4000.0,
    ack := P301B_Service_Ack, dt_ms := DT_MS);

P301A_Run := MotA.run_cmd;

```

```

P301B_Run    := MotB.run_cmd;
P301A_Fault  := MotA.fault;
P301B_Fault  := MotB.fault;
tNoPump(IN := Assist.q AND NOT (MotA.running OR MotB.running), PT := T#10s);
Station_Alarm := MotA.fault OR MotB.fault OR Alt.none_avail
               OR Level.under OR Level.over
               OR tNoPump.Q;

END_PROGRAM

```

G.4.5 Design notes

- **Division of work between the blocks.**

Block	Decides
HYSTERESIS (Demand)	Whether a pump is needed
PUMP_ALT2	Which pump runs
MOTOR_DOL	Whether it actually ran
RUNTIME_HRS	For how long

- **Pump-down sense.** HYSTERESIS with on_at > off_at works in the cooling sense: on at or above on_at , off at or below off_at . For a sump that is "start when high, stop when low", which is what is needed. The assist uses a second instance with 90 % / 70 %.
- **Balancing on hours, not on starts.** PUMP_ALT2 compares a_hours and b_hours and changes the duty pump only when the difference exceeds lead_gap_h , and only while demand is FALSE. Alternating on every start sounds equivalent but is not: a pump bumped for a minute in a test counts the same as one that ran a whole shift.
- **Failover.** A pump is *available* only when it is in AUTO and MOTOR_DOL reports no fault. When the duty pump becomes unavailable while running, PUMP_ALT2 sees a forced change under demand. It runs both available pumps for the overlap time (here only the healthy one), then makes the standby the duty.
- **Retained meters.** HoursA and HoursB are declared in VAR RETAIN , so the whole block instance, including its internal seconds counter, survives a restart. RUNTIME_HRS counts seconds internally because accumulating hours directly in a REAL stops advancing after a few thousand hours: the increment falls below the REAL's resolution.
- **dt_ms = 10** for MOTOR_DOL , RUNTIME_HRS and PUMP_ALT2 because the program runs in the 10 ms MainTask. RUNTIME_HRS defaults to 100. Leaving it at the default here would make the meters run ten times too fast.
- **The "nothing running" alarm has a delay.** Starting takes up to fb_timeout (3 s), so a bare Assist.q AND NOT running would flash on every assist start. tNoPump requires 10 s.

G.4.6 Task Configuration

Use MainTask at 10 ms with PumpStation in its call list.

```
"task": { "period_ms": 10, "calls": [ "PumpStation.st" ] }
```

G.4.7 Testing in the simulator

The simulated contactors do not report back, so you must play the contactor: force P301x_RunFb to follow P301x_Run within 3 s. Otherwise MOTOR_DOL faults with code 1. That fault path is itself a useful test.

Step	Force	Expected result
1	LT301_Raw = 12000 (50 %), P301A_Auto = P301B_Auto = TRUE, both ..._OL_OK = TRUE	Nothing runs. Duty_Is_B FALSE
2	LT301_Raw = 16000 (75 %)	P301A_Run TRUE. Force P301A_RunFb TRUE
3	LT301_Raw = 8000 (25 %)	P301A_Run FALSE. Force P301A_RunFb FALSE
4	Write HoursA.total_s = 200000 (≈ 55.6 h) and wait for the next scan	HoursA.total_h ≈ 55.6. With demand FALSE, Duty_Is_B becomes TRUE
5	LT301_Raw = 16000	P301B_Run TRUE. Force P301B_RunFb TRUE
6	P301B_OL_OK = FALSE	P301B_Fault TRUE. P301A_Run TRUE (failover). Force P301A_RunFb TRUE
7	LT301_Raw = 19000 (≈ 94 %), P301B_OL_OK = TRUE	Assist: MotB is still faulted and stays off. Station_Alarm TRUE
8	Reset_PB TRUE → FALSE	MotB fault cleared. Both pumps run while the level stays above 70 %
9	Leave P301A_RunFb FALSE after a start	After 3 s MotA.fault_code = 1 and the standby takes over

Note: Writing HoursA.total_s is a test shortcut. On a real controller, never write a retained meter from the monitor unless you mean to change the machine's recorded history. The write is recorded in the audit trail with your name.

G.4.8 Variations

- **Pressure-controlled booster set.** Replace the level hysteresis by a pressure switch or a pressure HYSTERESIS with on_at < off_at (the heating sense: start when pressure is low, stop when it is high).
- **Soft starters or VFDs.** Keep MOTOR_DOL for the run command and feedback. Add a SOFT_START per pump for the speed reference, with run := MotA.run_cmd .
- **Three pumps.** PUMP_ALT2 handles exactly two pumps. For three or more, keep one RUNTIME_HRS per pump and choose the duty in the program: the available pump with the fewest hours, evaluated only while no pump is running.

G.5 Valve and motor management with a device table

G.5.1 Task

A small skid with four on/off valves (XV-1 to XV-4, each with open and closed limit switches) and two DOL motors (M1, M2, each with a run-feedback contact). This example shows how to handle many similar devices as data rather than as copied code. The requirements are:

- Each device has a mode, **Off**, **Hand** or **Auto**, set from the HMI. In Hand it follows an HMI command. In Auto it follows the program.
- Each device reports a state (off, starting, on, stopping, fault), a fault code and a count of completed operations.

- Each device has parameters: feedback timeout, feedback fitted or not, and inverted output.
- Process interlocks: M1 only with XV-1 open. XV-2 (fill) and XV-3 (drain) never open together.
- The HMI gets one consistent snapshot of the table each scan.

The example uses user data types (Chapter 15): two enumerations, a parameter structure nested in a device structure, `ARRAY OF` that structure, a function block and a function that take the structure as input, and whole-structure and whole-array assignment.

G.5.2 I/O list

Address	Name	Type	Description
%IX0.0 ... %IX0.7	XV1_ZS0 , XV1_ZSC ... XV4_ZS0 , XV4_ZSC	BOOL	Open / closed limit switches of XV-1 ... XV-4
%IX1.0 , %IX1.1	M1_Run , M2_Run	BOOL	Motor contactor auxiliary contacts
%IX1.2	Reset_PB	BOOL	Fault reset
%QX0.0 ... %QX0.3	XV1_Sol ... XV4_Sol	BOOL	Valve solenoids
%QX0.4 , %QX0.5	M1_K , M2_K	BOOL	Motor contactors
%QX0.6	Common_Fault	BOOL	Common fault lamp

G.5.3 Data types

dut/E_Mode.st :

```

TYPE E_Mode :
(
    Off  := 0,      (* out of service: never commanded      *)
    Hand := 1,      (* operator commands it from the HMI        *)
    Auto := 2,      (* the program (sequence) commands it      *)
);
END_TYPE

```

dut/E_DevState.st :

```

TYPE E_DevState :
(
    DS_Off      := 0,  (* de-energised and at rest          *)
    DS_Starting := 1,  (* commanded on, feedback not yet confirmed *)
    DS_On       := 2,  (* commanded on and confirmed        *)
    DS_Stopping := 3,  (* commanded off, feedback not yet confirmed *)
    DS_Fault    := 4,  (* latched fault - needs a reset      *)
);
END_TYPE

```

dut/DevCfg_T.st :

```

TYPE DevCfg_T :
STRUCT
    FbTimeout  : TIME := T#10s;  (* travel time of a valve, start time of a motor *)
    HasFb      : BOOL := TRUE;   (* FALSE = no feedback fitted, time it instead *)
    Invert     : BOOL := FALSE;  (* TRUE = the output de-energises to act *)
END_STRUCT
END_TYPE

```

dut/Dev_T.st :

```

TYPE Dev_T :
STRUCT
    (* Commands *)
    Mode      : E_Mode := E_Mode.Auto;
    AutoCmd   : BOOL;      (* from the program / sequence *)
    HandCmd   : BOOL;      (* from the HMI *)
    Interlock : BOOL := TRUE; (* FALSE forces the device off *)
    (* Field *)
    FbOn      : BOOL;      (* valve: open limit; motor: run contact *)
    FbOff     : BOOL;      (* valve: closed limit; motor: not used *)
    Out       : BOOL;      (* to the solenoid / contactor *)
    (* Status *)
    State     : E_DevState;
    Fault     : BOOL;
    FaultCode : INT;      (* 1 no confirm on, 2 no confirm off,
                           3 both limits, 4 moved without command *)
    T_Entered : TIME;      (* when State was entered (TIME_NOW) *)
    Cycles    : DINT;      (* completed on-commands *)
    (* Parameters *)
    Cfg       : DevCfg_T;
END_STRUCT
END_TYPE

```

Note: All enumerators share one namespace across the whole project, so no two enumerations can use the same name. That is why the state enumerators carry a `DS_` prefix: a plain `off` would clash with `E_Mode.off`.

G.5.4 Global variables

globals.st :

```

(* ---- Field IO for 4 valves and 2 motors ----- *)
VAR_GLOBAL
    XV1_ZSO AT %IX0.0 : BOOL;   XV1_ZSC AT %IX0.1 : BOOL;
    XV2_ZSO AT %IX0.2 : BOOL;   XV2_ZSC AT %IX0.3 : BOOL;
    XV3_ZSO AT %IX0.4 : BOOL;   XV3_ZSC AT %IX0.5 : BOOL;
    XV4_ZSO AT %IX0.6 : BOOL;   XV4_ZSC AT %IX0.7 : BOOL;
    M1_Run  AT %IX1.0 : BOOL;   M2_Run  AT %IX1.1 : BOOL;
    Reset_PB AT %IX1.2 : BOOL;

    XV1_Sol AT %QX0.0 : BOOL;   XV2_Sol AT %QX0.1 : BOOL;
    XV3_Sol AT %QX0.2 : BOOL;   XV4_Sol AT %QX0.3 : BOOL;
    M1_K    AT %QX0.4 : BOOL;   M2_K    AT %QX0.5 : BOOL;
    Common_Fault AT %QX0.6 : BOOL;
END_VAR

(* ---- The device table: browsed by the HMI, SCADA and OPC UA ----- *)
VAR_GLOBAL
    Valves      : ARRAY[1..4] OF Dev_T;
    Motors      : ARRAY[1..2] OF Dev_T;
    Hmi_Valves   : ARRAY[1..4] OF Dev_T; (* consistent snapshot for the screens *)
    Hmi_Reset    : BOOL;
END_VAR

VAR_GLOBAL RETAIN
    ValveDefaults : DevCfg_T; (* copied to every valve at first scan *)
    MotorDefaults : DevCfg_T;
END_VAR

```

G.5.5 Function and function block

pou/F_DevHealthy.st :

```

(* A FUNCTION that takes a whole struct as its input. *)
FUNCTION F_DevHealthy : BOOL
VAR_INPUT
    D : Dev_T;
END_VAR
F_DevHealthy := NOT D.Fault AND D.Mode <> E_Mode.Off;
END_FUNCTION

```

pou/FB_Device.st :

```

(* One discrete device - a two-position valve or a DOL motor - held in a
   Dev_T. ALL state is in the struct, none in the block, so ONE instance of
   this block can service a whole ARRAY OF Dev_T in a FOR Loop. *)
FUNCTION_BLOCK FB_Device
VAR_INPUT
    Cfg      : DevCfg_T;      (* parameters, passed as a whole struct *)
    IsValve  : BOOL;         (* TRUE: FbOff is a closed limit switch *)
    ResetEdge : BOOL;         (* one-scan pulse from the caller *)
    Now      : TIME;         (* TIME_NOW() sampled once by the caller *)
END_VAR
VAR_IN_OUT
    Dev      : Dev_T;
END_VAR
VAR_OUTPUT
    Faulted  : BOOL;
END_VAR
VAR
    want     : BOOL;
    isOn     : BOOL;
    isOff    : BOOL;
    elapsed  : TIME;
END_VAR

IF ResetEdge AND Dev.Fault THEN
    Dev.Fault := FALSE;
    Dev.FaultCode := 0;
    Dev.State := DS_Off;
    Dev.T_Entered := Now;
END_IF;

(* What is wanted, by mode. *)
CASE Dev.Mode OF
    Off: want := FALSE;
    Hand: want := Dev.HandCmd;
    Auto: want := Dev.AutoCmd;
ELSE
    want := FALSE;
END_CASE;
want := want AND Dev.Interlock AND NOT Dev.Fault;

(* Where it is. Without feedback, position is assumed after the timeout. *)
elapsed := Now - Dev.T_Entered;
IF Cfg.HasFb THEN
    IF IsValve THEN
        isOn := Dev.FbOn AND NOT Dev.FbOff;
        isOff := Dev.FbOff AND NOT Dev.FbOn;
        IF Dev.FbOn AND Dev.FbOff AND NOT Dev.Fault THEN
            Dev.Fault := TRUE;
            Dev.FaultCode := 3;
        END_IF;
    ELSE
        isOn := Dev.FbOn;
        isOff := NOT Dev.FbOn;
    END_IF;
ELSE
    isOn := want AND elapsed >= Cfg.FbTimeout;
    isOff := NOT want AND elapsed >= Cfg.FbTimeout;
END_IF;

```

```

(* State machine. Every change of State restarts T_Entered. *)
CASE Dev.State OF
  DS_Off:
    IF want THEN
      Dev.State := DS_Starting;
      Dev.T_Entered := Now;
    ELSIF isOn AND Cfg.HasFb AND elapsed >= Cfg.FbTimeout THEN
      Dev.Fault := TRUE;          (* moved without a command *)
      Dev.FaultCode := 4;
    END_IF;
  DS_Starting:
    IF NOT want THEN
      Dev.State := DS_Stopping;
      Dev.T_Entered := Now;
    ELSIF isOn THEN
      Dev.State := DS_On;
      Dev.T_Entered := Now;
      Dev.Cycles := Dev.Cycles + 1;
    ELSIF elapsed >= Cfg.FbTimeout THEN
      Dev.Fault := TRUE;
      Dev.FaultCode := 1;
    END_IF;
  DS_On:
    IF NOT want THEN
      Dev.State := DS_Stopping;
      Dev.T_Entered := Now;
    ELSIF NOT isOn AND Cfg.HasFb THEN
      Dev.Fault := TRUE;          (* lost its feedback while on *)
      Dev.FaultCode := 1;
    END_IF;
  DS_Stopping:
    IF want THEN
      Dev.State := DS_Starting;
      Dev.T_Entered := Now;
    ELSIF isOff THEN
      Dev.State := DS_Off;
      Dev.T_Entered := Now;
    ELSIF elapsed >= Cfg.FbTimeout THEN
      Dev.Fault := TRUE;
      Dev.FaultCode := 2;
    END_IF;
END_CASE;

IF Dev.Fault THEN
  Dev.State := DS_Fault;
END_IF;

(* The output. Invert is for a fail-open valve whose solenoid closes it. *)
Dev.Out := (Dev.State = DS_Starting OR Dev.State = DS_On) XOR Cfg.Invert;
Faulted := Dev.Fault;
END_FUNCTION_BLOCK

```

G.5.6 Program

pou/Devices.st :

```

PROGRAM Devices
VAR
    Dev      : FB_Device;  (* ONE instance serves every element of both arrays *)
    rReset   : R_TRIG;
    FirstScan : BOOL := TRUE;
    Now      : TIME;
    i        : INT;
    AnyFault : BOOL;
    Healthy  : INT;        (* devices in service and not faulted *)
END_VAR

Now := TIME_NOW();
rReset(CLK := Reset_PB OR Hmi_Reset);

(* First scan: give every device its default parameters - a whole-struct
copy per element - and start its state clock. *)
IF FirstScan THEN
    FOR i := 1 TO 4 DO
        Valves[i].Cfg := ValveDefaults;
        Valves[i].T_Entered := Now;
    END_FOR;
    FOR i := 1 TO 2 DO
        Motors[i].Cfg := MotorDefaults;
        Motors[i].T_Entered := Now;
    END_FOR;
    FirstScan := FALSE;
END_IF;

(* ---- Inputs into the table ----- *)
Valves[1].FbOn := XV1_ZSO;  Valves[1].FbOff := XV1_ZSC;
Valves[2].FbOn := XV2_ZSO;  Valves[2].FbOff := XV2_ZSC;
Valves[3].FbOn := XV3_ZSO;  Valves[3].FbOff := XV3_ZSC;
Valves[4].FbOn := XV4_ZSO;  Valves[4].FbOff := XV4_ZSC;
Motors[1].FbOn := M1_Run;
Motors[2].FbOn := M2_Run;

(* ---- Process interlocks ----- *)
(* Transfer pump M1 may run only with its suction valve XV1 open. *)
Motors[1].Interlock := Valves[1].State = DS_On;
(* Fill valve XV2 and drain valve XV3 must never be open together. *)
Valves[3].Interlock := Valves[2].State = DS_Off;
Valves[2].Interlock := Valves[3].State = DS_Off;

(* ---- Device logic ----- *)
AnyFault := FALSE;
Healthy := 0;
FOR i := 1 TO 4 DO
    Dev(Cfg := Valves[i].Cfg, IsValve := TRUE, ResetEdge := rReset.Q,
        Now := Now, Dev := Valves[i]);
    AnyFault := AnyFault OR Dev.Faulted;
    IF F_DevHealthy(Valves[i]) THEN Healthy := Healthy + 1; END_IF;
END_FOR;
FOR i := 1 TO 2 DO
    Dev(Cfg := Motors[i].Cfg, IsValve := FALSE, ResetEdge := rReset.Q,
        Now := Now, Dev := Motors[i]);
    AnyFault := AnyFault OR Dev.Faulted;
    IF F_DevHealthy(Motors[i]) THEN Healthy := Healthy + 1; END_IF;
END_FOR;

```

```
(* ---- Table to outputs ----- *)
XV1_So1 := Valves[1].Out;
XV2_So1 := Valves[2].Out;
XV3_So1 := Valves[3].Out;
XV4_So1 := Valves[4].Out;
M1_K    := Motors[1].Out;
M2_K    := Motors[2].Out;
Common_Fault := AnyFault;

(* One consistent picture for the screens: the whole array in one copy. *)
Hmi_Valves := Valves;
END_PROGRAM
```

G.5.7 Design notes

- **One block instance for the whole table.** The current compiler accepts an `ARRAY OF` a function block type as a declaration, but it cannot *call* an element (`fbs[i](...)` is refused). The usual "array of instances in a FOR loop" pattern is therefore not available. This example uses a better-suited pattern: all state lives in the `Dev_T` structure, including the time the state was entered (`T_Entered`, from `TIME_NOW()`). The block itself holds nothing that must persist between calls, so one instance can serve every element in turn through `VAR_IN_OUT` `Dev : Dev_T`. Elapsed time is `Now - Dev.T_Entered` instead of a `TON` per device.
- **VAR_IN_OUT passes by reference.** `Dev := Valves[i]` gives the block the array element itself, not a copy, so its changes to `State`, `Out` and `Fault` land in the table. The parameter structure is passed the other way: `Cfg := Valves[i].Cfg` copies the whole `DevCfg_T` into the block's input.
- **TIME_NOW() once per scan.** The program samples the clock once and passes it to every call, so every device in one scan sees the same instant.
- **Enumerations in CASE.** The branches use bare enumerator names (`Off:`, `Hand:`, `DS_Starting:`). Comparisons elsewhere may use the qualified form (`D.Mode <> E_Mode.Off`), and so may initial values (`Mode : E_Mode := E_Mode.Auto;`).

Note: Write one enumerator per `CASE` label. In the current compiler, a comma list of *enumerators* (`Hand, Auto:`) is misread when it is not the first branch of the `CASE`. A comma list of integers (`1, 2:`) works everywhere.

- **Whole-structure copies.**
 - `Valves[i].Cfg := ValveDefaults;` copies one `DevCfg_T` into a member of an array element.
 - `Hmi_Valves := Valves;` copies the whole array of structures in one assignment. The HMI and OPC UA read `Hmi_Valves`, which changes only once per scan at a defined point. They never see a device half-way through its update.
 - A structure that contains a function block instance (a `TON`, say) cannot be copied as a whole. `Dev_T` contains none, which is another reason the timing uses `TIME_NOW()`.
- **Parameters retained, status not.** `ValveDefaults` and `MotorDefaults` are `RETAIN` and are copied into the table at the first scan. After a power cycle, devices therefore start in `DS_Off` with fresh timers, but with the parameters the engineer set. Set `MotorDefaults.FbTimeout` to the motors' start time (for example `T#3s`) during commissioning. The DUT default of 10 s suits valves.
- **Interlocks act through the table.** `Interlock := FALSE` makes `want` `FALSE`, so a device that loses its interlock is switched off through its normal `DS_Stopping` state. A valve that then fails to close is still detected.
- **Fault codes.** 1 = no confirmation of on (start or open), or lost while on. 2 = no confirmation of off. 3 = both limit switches made. 4 = went on without a command.

G.5.8 Task Configuration

Use MainTask at 10 ms with `Devices` in its call list. The timing uses `TIME_NOW()` differences, so it is independent of the interval.

```
"task": { "period_ms": 10, "calls": [ "Devices.st" ] }
```

G.5.9 Testing in the simulator

Open **Global Variables** online. `Valves` and `Motors` expand to `[1] ... [4]` and then to their members (Chapter 22).

Step	Write / force	Expected result
1	Nothing, after download	Every device <code>State = DS_Off</code> , <code>Cfg.FbTimeout = T#10s</code> (copied from the defaults), <code>Healthy = 6</code>
2	Force <code>XV1_ZSC ... XV4_ZSC TRUE</code>	All valves report closed; no fault
3	Write <code>Valves[1].AutoCmd TRUE</code>	<code>XV1_SoI TRUE</code> , <code>Valves[1].State = DS_Starting</code>
4	Force <code>XV1_ZSC FALSE</code> , <code>XV1_ZSO TRUE</code>	<code>DS_On</code> , <code>Cycles = 1</code> . <code>Motors[1].Interlock TRUE</code>
5	Write <code>Motors[1].AutoCmd TRUE</code> and do not force <code>M1_Run</code>	After 10 s <code>Motors[1].FaultCode = 1</code> , <code>Common_Fault TRUE</code>
6	<code>Reset_PB TRUE → FALSE</code> , force <code>M1_Run TRUE</code>	Reset, then <code>DS_Starting → DS_On</code>
7	Write <code>Valves[2].AutoCmd TRUE</code> and force <code>XV2_ZSC FALSE</code> , <code>XV2_ZSO TRUE</code> ; then write <code>Valves[3].AutoCmd TRUE</code>	XV-2 opens. XV-3 stays off because its interlock is FALSE
8	Write <code>Valves[4].Mode = Hand (1)</code> , then <code>Valves[4].HandCmd TRUE</code>	XV-4 follows the HMI command. <code>AutoCmd</code> is ignored
9	Force <code>XV4_ZSO</code> and <code>XV4_ZSC</code> both TRUE	<code>Valves[4].FaultCode = 3</code>
10	Compare <code>Hmi_Valves[1]</code> with <code>Valves[1]</code>	Identical after each scan

G.5.10 Variations

- **Analogue devices.** Add `Pos_Pct : REAL` and `Sp_Pct : REAL` to the structure and a second block `FB_ControlValve` for positioners, using the same table pattern.
- **Per-device parameters from the HMI.** Remove the first-scan copy and declare the whole `Valves` array `RETAIN`, so each valve keeps its own parameters. Remember that retained status (`State`, `T_Entered`) then also survives a restart. Reset those members at the first scan.
- **Sequence control.** A sequence (G.6) writes only `Valves[n].AutoCmd`. The mode, the interlocks and the supervision stay in the device layer, so the sequence does not need to know about them.

G.6 Batch mixing sequence in SFC

G.6.1 Task

A mixing vessel MX-401 on load cells. The sequence is:

1. **Idle.** Wait for START. The vessel must be empty (< 5 kg), the E-stop healthy and the agitator not faulted.
2. **FillA.** Record the tare. Open XV-A until the recipe weight of ingredient A has been charged.
3. **FillB.** Open XV-B until A + B have been charged.
4. **Agitate** and **HeatUp** → **HoldTemp**, in parallel:
 - The agitator runs throughout.
 - Steam heats the jacket to the recipe temperature, then holds it with a small on/off band.
 - Both branches must finish before the sequence continues: at least the mixing time, *and* at least the hold time at temperature.
5. **Drain.** Open the outlet with the agitator running until the vessel is empty, and for at least 10 s.
6. **Complete.** Count the batch and wait for the operator to acknowledge.

From any active step, **ABORT** goes to **Aborted**. So do an E-stop, an agitator fault, a load-cell or temperature signal failure, or a step overrunning its supervision time. Every output drops at once, and the sequence waits there for an acknowledgement.

G.6.2 I/O list

Address	Name	Type	Description
%IW2	WT401_Raw	INT	Load cells, range 4–20 mA = 0–1000 kg
%IW4	TT401_Raw	INT	Temperature, range 4–20 mA = 0–150 °C
%IX0.0	Agit_RunFb	BOOL	Agitator contactor auxiliary
%IX0.1	Agit_OL_OK	BOOL	Agitator overload, NC
%IX0.2	EStop_OK	BOOL	Safety relay healthy
%QX0.0	XV_A	BOOL	Ingredient A inlet valve
%QX0.1	XV_B	BOOL	Ingredient B inlet valve
%QX0.2	XV_Drain	BOOL	Outlet valve
%QX0.3	Agit_K	BOOL	Agitator contactor
%QX0.4	Steam_Valve	BOOL	Jacket steam valve
%QX0.5	Lamp_Done	BOOL	Batch complete lamp

G.6.3 Global variables

globals.st :

```

(* ---- Mixing vessel MX-401: field IO ----- *)
VAR_GLOBAL
    WT401_Raw    AT %IW2   : INT;    (* Load cells, 4-20 mA = 0..1000 kg      *)
    TT401_Raw    AT %IW4   : INT;    (* 4-20 mA = 0..150 degC                 *)
    Agit_RunFb   AT %IX0.0 : BOOL;    (* agitator contactor auxiliary          *)
    Agit_OL_OK   AT %IX0.1 : BOOL;    (* agitator overload, NC                 *)
    EStop_OK     AT %IX0.2 : BOOL;    (* from the safety relay, TRUE = healthy *)

    XV_A         AT %QX0.0 : BOOL;    (* ingredient A inlet valve              *)
    XV_B         AT %QX0.1 : BOOL;    (* ingredient B inlet valve              *)
    XV_Drain     AT %QX0.2 : BOOL;    (* outlet valve                          *)
    Agit_K       AT %QX0.3 : BOOL;    (* agitator contactor                   *)
    Steam_Valve  AT %QX0.4 : BOOL;    (* jacket steam on/off valve             *)
    Lamp_Done    AT %QX0.5 : BOOL;    (* batch complete Lamp                  *)
END_VAR

(* ---- Batch commands and status (HMI / SCADA) ----- *)
VAR_GLOBAL
    Cmd_Start    : BOOL;    (* start a batch                         *)
    Cmd_Abort    : BOOL;    (* abort the batch - safe stop, product lost *)
    Cmd_Ack      : BOOL;    (* operator acknowledges Complete / Aborted *)
    Batch_Active : BOOL;
    Batch_Count  : DINT;
    Weight_kg    : REAL;
    Temp_C       : REAL;
END_VAR

VAR_GLOBAL RETAIN
    Rcp_A_kg     : REAL := 300.0;    (* recipe: ingredient A                  *)
    Rcp_B_kg     : REAL := 150.0;    (* recipe: ingredient B                  *)
    Rcp_Temp_C   : REAL := 60.0;    (* recipe: heat to                      *)
    Rcp_Hold     : TIME := T#5m;    (* recipe: hold at temperature while mixing *)
    Rcp_Mix      : TIME := T#10m;    (* recipe: minimum mixing time          *)
    Max_Fill     : TIME := T#10m;    (* supervision: Longest allowed fill of one *)
    Max_Heat     : TIME := T#45m;    (* supervision: Longest allowed heat-up  *)
END_VAR

```

G.6.4 Program

Create the POU as **Program — SFC (sequence)**. The tree then shows a **Chart** node under it, which draws the sequence and animates the active steps (Chapter 17).

```
pou/Mixer.st :
```

```

PROGRAM Mixer
VAR
    Weight      : SCALE;
    Temp        : SCALE;
    Agitator     : MOTOR_DOL;
    HeatCtl      : HYSTERESIS;
    Tare_kg      : REAL;      (* vessel weight when the batch started *)
    Net_kg       : REAL;      (* weight charged in this batch *)
    StepFault    : BOOL;      (* a step overran its supervision time *)
    AbortReq     : BOOL;      (* any reason to abandon the batch *)
    ResetAgit    : BOOL;
END_VAR

(* ===== *)
(* Part 1 - runs EVERY scan, whatever step the chart is in. *)
(* ===== *)
Weight(raw := WT401_Raw, raw_min := 4000, raw_max := 20000,
        eu_min := 0.0, eu_max := 1000.0, clamp := TRUE);
Temp(raw := TT401_Raw, raw_min := 4000, raw_max := 20000,
     eu_min := 0.0, eu_max := 150.0, clamp := TRUE);
Weight_kg := Weight.eu;
Temp_C    := Temp.eu;
Net_kg     := Weight_kg - Tare_kg;

(* Step supervision, from the steps' own timers. *)
StepFault := (FillA.X AND FillA.T > Max_Fill)
             OR (FillB.X AND FillB.T > Max_Fill)
             OR (HeatUp.X AND HeatUp.T > Max_Heat);

AbortReq := Cmd_Abort OR NOT EStop_OK OR Agitator.fault OR StepFault
           OR Weight.under OR Weight.over OR Temp.under OR Temp.over;

(* Outputs are a function of the active steps: when a step is left,
   its outputs drop by themselves. *)
XV_A    := FillA.X;
XV_B    := FillB.X;
XV_Drain := Drain.X;
Lamp_Done := Complete.X;

HeatCtl(pv := Temp_C, on_at := Rcp_Temp_C - 1.0, off_at := Rcp_Temp_C + 0.5,
        enable := HeatUp.X OR HoldTemp.X);
Steam_Valve := HeatCtl.q;

ResetAgit := Idle.X AND Cmd_Ack;
Agitator(start := Agitate.X OR Drain.X, stop := FALSE,
          interlock_ok := EStop_OK, fb_running := Agit_RunFb,
          ext_trip := NOT Agit_OL_OK, fb_timeout := T#3s,
          reset := ResetAgit, dt_ms := 10);
Agit_K := Agitator.run_cmd;

(* ===== *)
(* Part 2 - the sequence. *)
(* ===== *)
INITIAL_STEP Idle:
END_STEP

STEP FillA:
    ACTION TakeTare(P);      (* once, on entry *)
    ACTION Batch_Active(S); (* Latch a BOOL *)

```

```

    ACTION Cmd_Start(R);          (* consume the start command *)
END_STEP

STEP FillB:
END_STEP

(* Simultaneous branch: mixing and heating run at the same time. *)
STEP Agitate:
END_STEP

STEP HeatUp:
END_STEP

STEP HoldTemp:
END_STEP

STEP Drain:
END_STEP

STEP Complete:
    ACTION CountBatch(P);
    ACTION Batch_Active(R);
END_STEP

STEP Aborted:
    ACTION Batch_Active(R);
    ACTION Cmd_Abort(R);
END_STEP

ACTION TakeTare:
    Tare_kg := Weight_kg;
END_ACTION

ACTION CountBatch:
    Batch_Count := Batch_Count + 1;
END_ACTION

(* Abort transitions first: where two transitions leave the same step,
   the first one written wins. *)
TRANSITION FROM FillA TO Aborted := AbortReq; END_TRANSITION
TRANSITION FROM FillB TO Aborted := AbortReq; END_TRANSITION
TRANSITION FROM (Agitate, HeatUp) TO Aborted := AbortReq; END_TRANSITION
TRANSITION FROM (Agitate, HoldTemp) TO Aborted := AbortReq; END_TRANSITION
TRANSITION FROM Drain TO Aborted := AbortReq; END_TRANSITION

(* The normal path. *)
TRANSITION FROM Idle TO FillA
    := Cmd_Start AND EStop_OK AND Weight_kg < 5.0 AND NOT Agitator.fault;
END_TRANSITION
TRANSITION FROM FillA TO FillB := Net_kg >= Rcp_A_kg; END_TRANSITION
TRANSITION FROM FillB TO (Agitate, HeatUp)
    := Net_kg >= Rcp_A_kg + Rcp_B_kg;
END_TRANSITION
TRANSITION FROM HeatUp TO HoldTemp := Temp_C >= Rcp_Temp_C - 1.0; END_TRANSITION
TRANSITION FROM (Agitate, HoldTemp) TO Drain
    := Agitate.T >= Rcp_Mix AND HoldTemp.T >= Rcp_Hold;
END_TRANSITION
TRANSITION FROM Drain TO Complete := Weight_kg < 2.0 AND Drain.T >= T#10s; END_TRANSITION
TRANSITION FROM Complete TO Idle := Cmd_Ack; END_TRANSITION

```

```

TRANSITION FROM Aborted TO Idle := Cmd_Ack AND NOT Cmd_Abort AND EStop_OK; END_TRANSITION
END_PROGRAM

```

G.6.5 Design notes

- **Statements first, chart last.** A POU body may start with ordinary statements and end with a chart. The statements run on every scan, whatever the chart is doing. They are the right place for three things:
 - Things that must never stop: scaling, the agitator's `MOTOR_DOL` supervision, the abort conditions.
 - Things that must see every step: the step supervision timeouts.
 - The output mapping.
If the agitator block were called inside a step, it would stop being called the moment the chart left that step, and its fault detection would stop with it.
- **Outputs as a function of the active steps.** `XV_A := FillA.X;` means the valve is open exactly while `FillA` is active. When the chart leaves the step, for any reason including an abort, the valve closes on the next scan. The alternative is to write `XV_A := TRUE;` inside the step, but that leaves the output `TRUE` after the step ends. A step's statements run only *while* the step is active; nothing resets what they wrote.
- **Step flags and timers.** Each step `S` provides `S.X` (active) and `S.T` (time active, `T#0ms` when inactive). They are ordinary variables. The supervision timeouts (`FillA.T > Max_Fill`) and the hold condition (`HoldTemp.T >= Rcp_Hold`) use them directly. There is no need for a `TON` per step.
- **Action qualifiers used.**

Association	Meaning here
<code>ACTION TakeTare(P);</code>	Runs the <code>TakeTare</code> action body once , on entry to <code>FillA</code>
<code>ACTION Batch_Active(S);</code>	Sets the <code>BOOL Batch_Active</code> and leaves it set
<code>ACTION Batch_Active(R);</code>	Resets it (in <code>Complete</code> and <code>Aborted</code>)
<code>ACTION Cmd_Start(R);</code>	Consumes the HMI start command, so a start held <code>TRUE</code> does not start the next batch too

`S` and `R` may name a `BOOL` variable directly. `N` and `P` name an action. Timed qualifiers (`L`, `D`, `SD`, `DS`, `SL`) are refused by the compiler. Use the step timer instead.

- **Simultaneous branch.** `TRANSITION FROM FillB TO (Agitate, HeatUp)` activates both steps. The right-hand branch moves on by itself (`HeatUp → HoldTemp`). `TRANSITION FROM (Agitate, HoldTemp) TO Drain` waits until *both* are active and the condition is `TRUE`, then deactivates both together.
- **Abort priority.** When two transitions leave the same step and both are `TRUE`, the one written first is taken. The abort transitions are therefore written first. A simultaneous branch has two possible step pairs, so it needs two abort transitions: (`Agitate, HeatUp`) and (`Agitate, HoldTemp`).
- **One transition per scan.** The chart evaluates every transition against the state at the start of the scan and then applies them. A step entered on this scan runs its actions on the next. A sequence therefore never races through several steps in one scan, and each step is visible online.
- **After an abort** the vessel may hold product. `Aborted → Idle` requires an acknowledgement. `Idle → FillA` requires the vessel to be empty, so the next batch cannot start on top of the remains. Emptying the vessel is a manual procedure, or a variation (G.6.8).

Note: The current compiler reports warning: action 'TakeTare' is never associated with a step (and the same for CountBatch), even though both are associated and run correctly. The warning is issued before the associations are resolved. You can ignore it for actions that are associated with a step. Check each warning of this kind against the chart once, rather than ignoring all of them.

G.6.6 Task Configuration

Use MainTask at 10 ms with Mixer in its call list. MOTOR_DOL is given `dt_ms := 10` to match.

```
"task": { "period_ms": 10, "calls": [ "Mixer.st" ] }
```

G.6.7 Testing in the simulator

Open the **Chart** node beside the **Variables** node. The active step is filled green and shows its elapsed time. For quicker tests, write short recipe times first: `Rcp_Mix = T#20s`, `Rcp_Hold = T#10s`.

Step	Force / write	Expected result
1	WT401_Raw = 4000 (0 kg), TT401_Raw = 6133 (20 °C), EStop_OK = TRUE, Agit_OL_OK = TRUE	Idle active
2	Cmd_Start TRUE	FillA: XV_A TRUE, Batch_Active TRUE, Cmd_Start back to FALSE, Tare_kg = 0
3	WT401_Raw = 8800 (300 kg)	FillB: XV_A FALSE, XV_B TRUE
4	WT401_Raw = 11200 (450 kg)	Agitate and HeatUp both active. Agit_K TRUE. Force Agit_RunFb TRUE within 3 s. Steam_Valve TRUE
5	TT401_Raw = 10400 (60 °C)	HoldTemp replaces HeatUp. The steam valve cycles as you move TT401_Raw around 10300–10450
6	Wait for the mixing and hold times	Drain: XV_Drain TRUE, agitator still running
7	WT401_Raw = 4000	After 10 s in Drain: Complete, Lamp_Done TRUE, Batch_Count + 1
8	Cmd_Ack TRUE, then FALSE	Idle
9	Start again and, in FillB, write Cmd_Abort TRUE	Aborted. Every valve FALSE. Cmd_Abort resets itself
10	Start again and leave WT401_Raw unchanged in FillA	After Max_Fill (write T#15s to test): StepFault → Aborted
11	During Agitate, Agit_OL_OK = FALSE	Agitator.fault → Aborted. Cmd_Ack in Idle also resets the agitator fault

Cmd_Start, Cmd_Ack and Cmd_Abort are levels. The chart consumes Cmd_Start and Cmd_Abort itself. Cmd_Ack should be a momentary button on the HMI, or written back to FALSE by hand in the test.

G.6.8 Variations

- **Emptying after an abort.** Add a step AbortDrain between Aborted and Idle, entered on an operator command, that opens XV_Drain until the weight is below 2 kg.

- **Hold and restart.** For a sequence that must be resumable (a HOLD that keeps the product good, as opposed to an ABORT that loses it), use the ISA-88 layer in `50_s88.st`. **New POU** offers **Program — S88 phase (batch)**, which generates a phase with the standard states and commands (Chapter 28).
- **Recipes from SCADA.** The `Rcp_...` variables are retained globals. A batch system or OptiZeus SCADA writes them by OPC UA before `Cmd_Start`. To stop a recipe changing mid-batch, copy them to program-local variables in a `P` action of `FillA`, and use the copies in the transitions.

G.7 A multi-task project

G.7.1 Task

A filling line where some functions must be fast and others need not be. The example shows how to divide one application between three tasks, and how the tasks exchange data safely.

Program	Task	Interval	Does
FastIO	MainTask	10 ms	Inputs, interlock chain with first-out, pump starter, outputs, part counting
Calc	Calc	100 ms, offset 3	Scaling, filtering, a 1 s average, low-pressure detection, flow totalising
Report	Report	1000 ms, offset 7	Life bit and uptime for SCADA, rejects per minute, shift maximum and minimum

G.7.2 I/O list

Address	Name	Type	Description
%IX0.0	EStop_OK	BOOL	Safety relay healthy
%IX0.1	Guard_OK	BOOL	Guard door closed
%IX0.2	Air_OK	BOOL	Air pressure switch
%IX0.3	Pump_OL_OK	BOOL	Pump overload, NC
%IX0.4	Pump_RunFb	BOOL	Pump contactor auxiliary
%IX0.5	Start_PB	BOOL	Start, NO
%IX0.6	Stop_OK	BOOL	Stop, NC
%IX0.7	Reject_Eye	BOOL	Reject chute photo-eye, one pulse per part
%IX1.0	Reset_PB	BOOL	Reset
%IW2	FT501_Raw	INT	Flow, range 4–20 mA = 0–20 m ³ /h
%IW4	PT501_Raw	INT	Discharge pressure, range 4–20 mA = 0–10 bar
%QX0.0	Pump_K	BOOL	Pump contactor
%QX0.1	Lamp_Run	BOOL	Running lamp
%QX0.2	Lamp_Trip	BOOL	Trip lamp

G.7.3 Global variables

`globals.st` :

```

(* ---- Filling line: field IO ----- *)
VAR_GLOBAL
    EStop_OK      AT %IX0.0 : BOOL; (* safety relay healthy *)
    Guard_OK      AT %IX0.1 : BOOL; (* guard door closed *)
    Air_OK        AT %IX0.2 : BOOL; (* air pressure switch *)
    Pump_OL_OK    AT %IX0.3 : BOOL; (* pump overload, NC *)
    Pump_RunFb    AT %IX0.4 : BOOL; (* pump contactor auxiliary *)
    Start_PB      AT %IX0.5 : BOOL;
    Stop_OK       AT %IX0.6 : BOOL; (* stop button, NC *)
    Reject_Eye    AT %IX0.7 : BOOL; (* reject chute photo-eye, one pulse a part *)
    Reset_PB      AT %IX1.0 : BOOL;
    FT501_Raw     AT %IW2  : INT;  (* flow, 4-20 mA = 0..20 m3/h *)
    PT501_Raw     AT %IW4  : INT;  (* pressure, 4-20 mA = 0..10 bar *)

    Pump_K        AT %QX0.0 : BOOL;
    Lamp_Run      AT %QX0.1 : BOOL;
    Lamp_Trip     AT %QX0.2 : BOOL;
END_VAR

(* ---- Exchanged between the tasks ----- *)
VAR_GLOBAL
    (* written by FastIO (10 ms) *)
    Line_Running  : BOOL;
    Trip_Active   : BOOL;
    Trip_FirstOut : INT;          (* 1 EStop 2 Guard 3 Air 4 Overload *)
    Reject_Count  : DINT;         (* free-running counter - never reset *)
    (* written by Calc (100 ms) *)
    Flow_m3h      : REAL;
    Pressure_bar  : REAL;
    Flow_Avg_m3h  : REAL;
    Low_Pressure  : BOOL;         (* consumed by FastIO as a trip input *)
    Total_m3      : LREAL;
    (* written by Report (1 s) *)
    Rejects_per_min : DINT;
    Life_Bit       : BOOL;        (* toggles every second - SCADA watchdog *)
    Uptime_s       : DINT;
    Flow_Max_Shift : REAL;
    Flow_Min_Shift : REAL;
    (* commands from the HMI *)
    Hmi_ResetTotal : BOOL;        (* Level: Calc clears it when done *)
    Hmi_NewShift   : BOOL;        (* Level: Report clears it when done *)
END_VAR

```

G.7.4 Programs

pou/FastIO.st :


```

(* MainTask, 10 ms: inputs, interlocks, outputs. Nothing slow in here. *)
PROGRAM FastIO
VAR CONSTANT
    FAST_DT_MS : DINT := 10;
END_VAR
VAR
    GuardDb      : DEBOUNCE;
    Trips        : TRIP_FIRST_OUT;
    Pump         : MOTOR_DOL;
    rReset       : R_TRIG;
    rReject      : R_TRIG;
    RunReq       : BOOL;
END_VAR

(* Inputs *)
GuardDb(x := Guard_OK, stable_time := T#50ms);
rReset(CLK := Reset_PB);
rReject(CLK := Reject_Eye);
IF rReject.Q THEN
    Reject_Count := Reject_Count + 1;  (* the slow task reads the difference *)
END_IF;

(* Interlock chain - healthy when TRUE, first-out latched. The guard is
   debounced on OPENING only (Guard_OK OR debounced): a closed guard is
   healthy from the first scan, an opening must last 50 ms to trip. *)
Trips(ok_1 := EStop_OK, ok_2 := Guard_OK OR GuardDb.q, ok_3 := Air_OK,
      ok_4 := Pump_OL_OK, ok_5 := NOT Low_Pressure,
      reset := rReset.Q);
Trip_Active := Trips.tripped;
Trip_FirstOut := Trips.first_out;

(* Pump with start/stop seal-in *)
RunReq := (Start_PB OR RunReq) AND Stop_OK AND NOT Trips.tripped;
Pump(start := RunReq, stop := NOT Stop_OK, interlock_ok := NOT Trips.tripped,
     fb_running := Pump_RunFb, ext_trip := NOT Pump_OL_OK,
     fb_timeout := T#3s, reset := rReset.Q, dt_ms := FAST_DT_MS);

(* Outputs *)
Pump_K := Pump.run_cmd;
Line_Running := Pump.running;
Lamp_Run := Pump.running;
Lamp_Trip := Trips.tripped OR Pump.fault;
END_PROGRAM

```

pou/Calc.st :

```

(* Task "Calc", 100 ms: scaling, filtering, totalising. *)
PROGRAM Calc
VAR CONSTANT
    CALC_DT_MS : DINT := 100;      (* MUST equal this task's interval *)
END_VAR
VAR
    FlowScale : SCALE;
    PresScale : SCALE;
    FlowFilt  : FILTER_PT1;
    FlowAvg   : AVERAGE_N;        (* 10 samples x 100 ms = 1 s window *)
    LowPres   : HYSTERESIS;
    tLowPres  : TON;
END_VAR
VAR RETAIN
    Totaliser : TOTALIZER;        (* the running total survives a restart *)
END_VAR

FlowScale(raw := FT501_Raw, raw_min := 4000, raw_max := 20000,
          eu_min := 0.0, eu_max := 20.0, clamp := TRUE);
PresScale(raw := PT501_Raw, raw_min := 4000, raw_max := 20000,
          eu_min := 0.0, eu_max := 10.0, clamp := TRUE);

FlowFilt(x := FlowScale.eu, tau_ms := 1000, dt_ms := CALC_DT_MS, reset := FALSE);
Flow_m3h := FlowFilt.y;
Pressure_bar := PresScale.eu;

FlowAvg(pv := Flow_m3h, n := 10, reset := FALSE);
Flow_Avg_m3h := FlowAvg.out;

(* Low discharge pressure while running, for 3 s: a trip for FastIO.
   HYSTERESIS in the cooling sense (on_at > off_at) is "on when HIGH", so
   invert the comparison by feeding it the pressure DEFICIT. *)
LowPres(pv := 10.0 - Pressure_bar, on_at := 9.0, off_at := 8.5,
        enable := Line_Running);
tLowPres(IN := LowPres.q, PT := T#3s);
Low_Pressure := tLowPres.Q;

(* Total, in m3, from a rate in m3/h. *)
Totaliser(rate := Flow_m3h, rate_base := 2, dt_ms := CALC_DT_MS,
          run := Line_Running, reset := Hmi_ResetTotal, cutoff := 0.2);
Total_m3 := Totaliser.total;
IF Hmi_ResetTotal THEN
    Hmi_ResetTotal := FALSE;      (* acknowledge the command by clearing it *)
END_IF;
END_PROGRAM

```

pou/Report.st :

```

(* Task "Report", 1000 ms: statistics and housekeeping. *)
PROGRAM Report
VAR
    ShiftFlow    : MINMAX_HOLD;
    MinuteSecs   : INT;
    MinuteBase   : DINT;
    FirstRun     : BOOL := TRUE;
END_VAR

Life_Bit := NOT Life_Bit;
Uptime_s := Uptime_s + 1;

IF FirstRun THEN
    MinuteBase := Reject_Count;
    FirstRun := FALSE;
END_IF;

(* Rejects per minute, from the difference of a counter FastIO never
   resets. A one-scan pulse from the 10 ms task would be missed here 99
   times out of 100; a counter cannot be. *)
MinuteSecs := MinuteSecs + 1;
IF MinuteSecs >= 60 THEN
    Rejects_per_min := Reject_Count - MinuteBase;
    MinuteBase := Reject_Count;
    MinuteSecs := 0;
END_IF;

(* Shift peaks. *)
ShiftFlow(pv := Flow_m3h, reset := Hmi_NewShift, enable := Line_Running);
Flow_Max_Shift := ShiftFlow.max_val;
Flow_Min_Shift := ShiftFlow.min_val;
IF Hmi_NewShift THEN
    Hmi_NewShift := FALSE;
END_IF;
END_PROGRAM

```

G.7.5 How the tasks run

All tasks are rates of the one real-time MainTask clock, not separate threads. In a scan where `Calc` is due, it runs inside that scan, after the programs before it in the call list. This has three consequences:

- **Data is always consistent.** A program never sees another program's variables half-updated, and no locking is needed. The globals in the "exchanged between the tasks" block are ordinary variables.
- **A slow task costs scan time when it runs.** `Report` runs in one scan in a hundred, and that scan is longer by `Report`'s execution time. Staggering the tasks with offsets (`Calc` on scan 3 of every 10, `Report` on scan 7 of every 100) keeps them from landing on the same scan.
- **A task that sits out a scan loses nothing.** Its variables, timers and block instances stay as they were.

G.7.6 Design notes

- **dt_ms per task.** `Calc` declares `CALC_DT_MS := 100` and gives it to `FILTER_PT1` and `TOTALIZER`. If `Calc` is ever moved to another task, change that constant too.
 - `AVERAGE_N` is also affected, less visibly: its window is 10 *samples*, which is 1 s at 100 ms but 0.1 s at 10 ms.
 - `FastIO` gives `MOTOR_DOL` its own `FAST_DT_MS := 10`.

- **Pulses do not cross tasks; counters do.** A signal TRUE for one 10 ms scan is invisible to a 1 s task 99 times in 100. `FastIO` therefore counts reject pulses into `Reject_Count`, which it never resets. `Report` takes the difference each minute. The same rule applies in reverse: a slow task's output lasts at least one of its own periods, so a fast task always sees it.
- **Commands consumed by the task that acts on them.** The HMI sets `Hmi_ResetTotal`. `Calc` acts on it and clears it, and `Report` does the same with `Hmi_NewShift`. The command is a level that stays set until the slow task has seen it, so it cannot be missed.
- **A slow input to a fast interlock.** `Low_Pressure` is calculated at 100 ms with a 3 s delay and consumed by the 10 ms interlock chain as `ok_5`. The extra delay of up to 100 ms does not matter for a condition that must already persist for 3 s. The E-stop, guard and overload inputs, whose timing does matter, are read directly in `FastIO`.
- **First-out interlocks.** `TRIP_FIRST_OUT` latches the *first* input to go unhealthy (`Trip_FirstOut` 1–5). It clears on reset only when every input is healthy again.
- **Guard debounce.** The guard is debounced on opening only: `Guard_OK OR GuardDb.q`. A closed guard is healthy from the first scan, so the line does not start up with a spurious guard trip. An opening must last 50 ms to trip.
- **Unique local names across programs.** Give each program's locals names that are unique across all programs. Two programs that each declare a local `n` compile, but the monitor, a force and the trace resolve a bare name to only one of them. The constants here are therefore `FAST_DT_MS` and `CALC_DT_MS`, not two `DT_MS`.
- **Global names that collide with library internals.** A global with the same name as a local variable inside a library block produces a shadowing warning at that block. An early draft of this example used a global `Heartbeat`, which collides with a local of `S88_RESOURCE` in `50_s88.st`. It is called `Life_Bit` for that reason.

G.7.7 Task Configuration

To create the tasks:

1. Open **Task Configuration** → **Task Groups** tab.
2. Click + **Add task**. Enter the name `Calc`, **Interval (ms)** `100` and **Offset (scans)** `3`. Tick `Calc` and click **Add task**.
3. Click + **Add task** again. Enter the name `Report`, **Interval (ms)** `1000` and **Offset (scans)** `7`. Tick `Report` and click **Add task**.
4. Check the **Programs** table: `FastIO` → `MainTask`, `Calc` → `Calc`, `Report` → `Report`.
5. Click **Apply tasks**. The IDE also puts all three programs in `MainTask`'s call list.
6. Open **MainTask** and check the order in **POU calls**: `FastIO.st` first. Use **Move up** / **Move down** if needed.
7. Restart the runtime or use **Download all**.

The result in `controller.json`:

```

"task": {
  "period_ms": 10,
  "calls": [ "FastIO.st", "Calc.st", "Report.st" ],
  "groups": [
    { "name": "Calc", "period_ms": 100, "offset": 3 },
    { "name": "Report", "period_ms": 1000, "offset": 7 }
  ],
  "programs": [
    { "name": "Calc", "period_ms": 100, "offset": 3 },
    { "name": "Report", "period_ms": 1000, "offset": 7 }
  ]
}

```

A task interval must be a whole multiple of MainTask's interval. The offset must be smaller than the number of scans per run (here 10 and 100). The IDE refuses anything else, and so does the controller. For example, a request for 25 ms on a 10 ms MainTask is refused with the suggestion of 20 or 30 ms.

Note: The **MainTask** page (tree node **MainTask**) contains the sentence "several tasks with different intervals are not supported yet". It predates the **Task Groups** tab. Tasks with different intervals are configured on **Task Groups** as described above.

G.7.8 Testing in the simulator

Step	Force / write	Expected result
1	All <code>..._OK</code> inputs TRUE, <code>Stop_OK</code> TRUE	<code>Trip_Active</code> FALSE. Task Configuration → Properties shows every 10 scans (100 ms) for <code>Calc</code> and every 100 scans (1000 ms) for <code>Report</code>
2	Watch <code>Life_Bit</code>	Toggles once per second
3	<code>Start_PB</code> TRUE → FALSE, force <code>Pump_RunFb</code> TRUE	<code>Line_Running</code> TRUE
4	<code>FT501_Raw</code> = 12000 (10 m³/h), <code>PT501_Raw</code> = 12000 (5 bar)	<code>Flow_m3h</code> rises to 10 with a 1 s lag. <code>Total_m3</code> rises by about 0.0028 m³ per second
5	<code>PT501_Raw</code> = 5200 (0.75 bar)	After 3 s <code>Low_Pressure</code> TRUE → <code>Trip_Active</code> TRUE, <code>Trip_FirstOut</code> = 5, pump stops
6	<code>PT501_Raw</code> = 12000, then <code>Reset_PB</code> TRUE → FALSE	Trip cleared
7	<code>Guard_OK</code> FALSE for 20 ms, then for 100 ms	Only the second trips, with <code>Trip_FirstOut</code> = 2
8	Toggle <code>Reject_Eye</code> 30 times within a minute	After the minute boundary, <code>Rejects_per_min</code> = 30
9	Write <code>Hmi_ResetTotal</code> TRUE	<code>Total_m3</code> = 0 and <code>Hmi_ResetTotal</code> returns to FALSE within 100 ms

To see the task timing, trace `Life_Bit`, `Flow_m3h` and `Lamp_Trip` together. `Flow_m3h` moves in 100 ms steps. `Lamp_Trip` reacts within one 10 ms scan.

G.7.9 Variations

- **Moving a program between tasks.** Change the task on the **Task Groups** tab, then change the program's `_DT_MS` constant. Check every `AVERAGE_N` and `DERIV` in it: `AVERAGE_N`'s window is counted in samples, and `DERIV` has its own `dt_ms`.
- **Core assignment.** On a controller with the scan core isolated (`isolcpus`), leave the services unpinned. Otherwise, pin OPC UA and the HMI server away from the scan core on the **Task Groups** tab (Chapter 18). The three programs here all run on the scan thread, whatever their interval.
- **Watchdog for SCADA.** OptiZeus SCADA can alarm when `Life_Bit` has not changed for 5 s. That detects a stopped program, not only a lost connection.

G.8 IO-Link sensor: parameters at start-up and event handling

Important: Not yet validated on hardware. The IO-Link library (`stdlib/70_ioLink.st`) and this example have been tested against a simulated master only, not against a real ODOT IP-EC-8IA. Byte order, how a refused ISDU is reported, and event behaviour are still bench items (Chapter 12). Treat this example as a pattern to verify on your hardware before you depend on it.

G.8.1 Task

An IO-Link pressure sensor PT-601 is connected to port 1 of an ODOT IP-EC-8IA master on EtherCAT. The requirements are:

- Read the process value every scan. The sensor sends a 16-bit signed value, most significant byte first. In this sensor's IODD, one count is 0.01 bar.
- At start-up, and again after the sensor has been lost and has come back (for example after a replacement):
 1. Read the Vendor Name (index `16#0010`) and the Product Name (index `16#0012`) for the record.
 2. Read switch point SP1 (index `16#003C` , subindex 1).
 3. If SP1 differs from the value the plant wants, write it, then read it back to verify.
- Decode the master's events. Report "communication lost" (`16#1800`) on our port, and any error-type event. The operator acknowledges events from the HMI.

The indices above follow the IO-Link Smart Sensor Profile convention. Take the real indices, subindices, lengths and scaling from the IODD of your device.

G.8.2 Rack and I/O

Configure the rack from the ESI (Chapter 9 and Chapter 12). Master module in slot 1. An *Iolink Input 2 bytes* module for port 1. The **ISDU** module (34 bytes in, 39 bytes out) and an **IO-Link Events/2** module. The offsets below are an example. Read yours from the device's IO map in the IDE.

Module	Area	Byte offset	Constant
Port 1 process data	%I	2–3	PD_IN_OFF
ISDU, input	%I	40–73	ISDU_IN_OFF
ISDU, output	%Q	12–50	ISDU_OUT_OFF
IO-Link Events/2, input	%I	74–81	EV_IN_OFF

Module	Area	Byte offset	Constant
IO-Link Events/2, output	%Q	51	EV_OUT_OFF

The program does not use `AT` declarations for these modules. The library blocks take the byte offsets as parameters and read the process image through `PI_IN_BYTE`, `PI_OUT_BYTE` and `PI_SET_OUT_BYTE`. A located array (`AT %IB40 : ARRAY...`) is not supported.

WARNING: If a module is moved in the rack, its offsets change and these constants must change with it. Nothing detects a wrong offset that is still inside the process image: the program then reads and writes another module's bytes. An offset outside the image is detected (`ErrorID 4, Ev.Fault`).

G.8.3 Global variables

globals.st :

```
(* ---- IO-Link pressure sensor PT-601 on port 1 of an IP-EC-8IA ----- *)
(* The module offsets are BYTE offsets in %I / %Q, read from the
   device's IO map in the IDE. They change if a module moves in the rack. *)
VAR_GLOBAL CONSTANT
    IOL_PORT      : INT := 1;      (* master port the sensor is on      *)
    PD_IN_OFF     : DINT := 2;     (* port 1 process data, %IB2..%IB3   *)
    ISDU_IN_OFF   : DINT := 40;    (* ISDU module, 34 bytes in, %IB40.. *)
    ISDU_OUT_OFF  : DINT := 12;    (* ISDU module, 39 bytes out, %QB12.. *)
    EV_IN_OFF     : DINT := 74;    (* IO-Link Events/2 module, 8 bytes in *)
    EV_OUT_OFF    : DINT := 51;    (* ... and its clear/toggle output byte *)
END_VAR

VAR_GLOBAL
    (* Process value *)
    PT601_bar      : REAL;
    PT601_Raw      : INT;
    (* Identification read at start-up *)
    PT601_Vendor   : ARRAY[1..32] OF BYTE; (* index 16#0010, text bytes *)
    PT601_VendorLen : INT;
    PT601_Product  : ARRAY[1..32] OF BYTE; (* index 16#0012, text bytes *)
    PT601_ProductLen : INT;
    PT601_SP1_Read : DINT;                (* switch point as found *)
    (* Status *)
    PT601_Ready    : BOOL;                (* parameterised and verified *)
    PT601_Step     : INT;                 (* start-up sequence step *)
    PT601_ErrStep  : INT;                 (* step that failed *)
    PT601_ErrID    : INT;                 (* IOL_ISDU ErrorID *)
    PT601_CommLost : BOOL;                (* event 16#1800 on our port *)
    PT601_DevError : BOOL;                (* any error-type event *)
    PT601_EvCode   : DINT;                (* first event code shown *)
    (* HMI commands *)
    PT601_Retry    : BOOL;                (* rerun the start-up *)
    PT601_AckEvents : BOOL;
END_VAR

VAR_GLOBAL RETAIN
    PT601_SP1_Want : DINT := 600;        (* switch point wanted, device units *)
END_VAR
```

G.8.4 Program

pou/IoISensor.st :


```

(* NOT YET VALIDATED ON HARDWARE - see Appendix G.8 and Chapter 12. *)
PROGRAM IolSensor
VAR
    Rd      : IOL_ReadParam;
    Wr      : IOL_WriteParam;
    Ev      : IOL_Events;
    RdExec   : BOOL;          (* set by the sequence, used on the NEXT call *)
    WrExec   : BOOL;
    RdIndex  : DINT;
    RdSub    : INT;
    tSettle  : TON;           (* Let the port come up before talking to it *)
    rRetry   : R_TRIG;
    i        : INT;
END_VAR

(* ---- Process data: every scan, big-endian ----- *)
PT601_Raw := IOL_IN_INT(PD_IN_OFF);
PT601_bar := INT_TO_REAL(PT601_Raw) * 0.01;      (* 0.01 bar per count - from the IODD *)

(* ---- The ISDU blocks are called FIRST, with what the sequence decided
      Last scan. That guarantees each block sees Execute low for at least
      one call between two requests. ----- *)
Rd(Execute := RdExec, InBase := ISDU_IN_OFF, OutBase := ISDU_OUT_OFF,
    Port := IOL_PORT, Index := RdIndex, SubIndex := RdSub, Timeout := T#2s);
Wr(Execute := WrExec, InBase := ISDU_IN_OFF, OutBase := ISDU_OUT_OFF,
    Port := IOL_PORT, Index := 16#003C, SubIndex := 1,
    Length := 2, Value := PT601_SP1_Want, UseValue := TRUE, Timeout := T#2s);

(* ---- Start-up sequence: one ISDU transfer at a time ----- *)
rRetry(CLK := PT601_Retry);
IF rRetry.Q THEN
    RdExec := FALSE;
    WrExec := FALSE;
    PT601_Ready := FALSE;
    PT601_Step := 0;
END_IF;

tSettle(IN := PT601_Step = 0, PT := T#5s);

CASE PT601_Step OF
    0: (* wait for the port to come up *)
        IF tSettle.Q AND NOT PT601_CommLost THEN
            RdIndex := 16#0010; RdSub := 0;      (* Vendor Name *)
            RdExec := TRUE;
            PT601_Step := 10;
        END_IF;

    10: (* Vendor Name *)
        IF Rd.Done THEN
            PT601_VendorLen := Rd.RxLength;
            FOR i := 1 TO 32 DO PT601_Vendor[i] := Rd.RxData[i]; END_FOR;
            RdExec := FALSE;
            PT601_Step := 15;
        ELSIF Rd.Error THEN
            PT601_ErrStep := 10; PT601_ErrID := Rd.ErrorID;
            RdExec := FALSE;
            PT601_Step := 900;
        END_IF;
END_CASE

```

```

15: (* next request: Product Name *)
    RdIndex := 16#0012; RdSub := 0;
    RdExec := TRUE;
    PT601_Step := 20;

20: IF Rd.Done THEN
    PT601_ProductLen := Rd.RxLength;
    FOR i := 1 TO 32 DO PT601_Product[i] := Rd.RxData[i]; END_FOR;
    RdExec := FALSE;
    PT601_Step := 25;
    ELSIF Rd.Error THEN
    PT601_ErrStep := 20; PT601_ErrID := Rd.ErrorID;
    RdExec := FALSE;
    PT601_Step := 900;
    END_IF;

25: (* next request: switch point SP1 *)
    RdIndex := 16#003C; RdSub := 1;
    RdExec := TRUE;
    PT601_Step := 30;

30: IF Rd.Done THEN
    PT601_SP1_Read := Rd.Value;
    RdExec := FALSE;
    IF Rd.Value = PT601_SP1_Want THEN
    PT601_Step := 100; (* already right: nothing to write *)
    ELSE
    PT601_Step := 35;
    END_IF;
    ELSIF Rd.Error THEN
    PT601_ErrStep := 30; PT601_ErrID := Rd.ErrorID;
    RdExec := FALSE;
    PT601_Step := 900;
    END_IF;

35: WrExec := TRUE;
    PT601_Step := 40;

40: (* write SP1 *)
    IF Wr.Done THEN
    WrExec := FALSE;
    PT601_Step := 45;
    ELSIF Wr.Error THEN
    PT601_ErrStep := 40; PT601_ErrID := Wr.ErrorID;
    WrExec := FALSE;
    PT601_Step := 900;
    END_IF;

45: (* read it back - a write that was accepted is not proof it took *)
    RdIndex := 16#003C; RdSub := 1;
    RdExec := TRUE;
    PT601_Step := 50;

50: IF Rd.Done THEN
    PT601_SP1_Read := Rd.Value;
    RdExec := FALSE;
    IF Rd.Value = PT601_SP1_Want THEN
    PT601_Step := 100;
    ELSE
    PT601_ErrStep := 50; PT601_ErrID := 0; (* read-back mismatch *)

```

```

        PT601_Step := 900;
    END_IF;
ELSIF Rd.Error THEN
    PT601_ErrStep := 50; PT601_ErrID := Rd.ErrorID;
    RdExec := FALSE;
    PT601_Step := 900;
END_IF;

100: (* ready *)
    PT601_Ready := TRUE;

900: (* failed - wait for PT601_Retry *)
    PT601_Ready := FALSE;
END_CASE;

(* ---- Events: port and device diagnostics ----- *)
Ev(InBase := EV_IN_OFF, OutBase := EV_OUT_OFF, Count := 2, Ack := PT601_AckEvents);
PT601_DevError := Ev.AnyError;
PT601_EvCode   := Ev.FirstCode;
PT601_CommLost := FALSE;
FOR i := 1 TO 2 DO
    IF Ev.EvCode[i] = 16#1800 AND Ev.EvPort[i] = IOL_PORT THEN
        PT601_CommLost := TRUE;
    END_IF;
END_FOR;

(* A sensor that dropped off the port may have been replaced: when the
   port comes back, parameterise it again. *)
IF PT601_CommLost AND PT601_Step >= 100 THEN
    PT601_Ready := FALSE;
    PT601_Step := 0;
END_IF;
END_PROGRAM

```

G.8.5 Design notes

- **Big-endian process data.** IO-Link transmits the most significant byte first, and the process image is little-endian. Reading %IW2 would give a byte-swapped value. IOL_IN_INT(PD_IN_OFF) assembles the two bytes in the right order. Use IOL_IN_DINT or IOL_IN_REAL for 32-bit data. Many sensors put status bits in the low bits of the process data word. Mask or shift them off (SHR) as the IODD describes.
- **One ISDU transfer at a time.** The ISDU module has one request slot. Rd and Wr share it, and the sequence never sets both Execute inputs at once.
- **Execute timing.** The blocks follow the PLCopen convention: a rising edge on Execute starts a transfer, and Done / Error stay TRUE while Execute is held. The program calls the blocks first, with the Execute value the sequence chose on the previous scan, and then runs the sequence. When the sequence sees Done, it drops Execute and moves to an intermediate step (15 , 25 , 35 , 45) that sets up the next request. Each block therefore sees Execute low for at least one call between two requests. Setting Execute TRUE again on the same scan as Done would restart nothing, and the next state would misread the old Done as its own.
- **Parameters latched on the edge.** RdIndex and RdSub are set in the same step that raises RdExec. The block reads them on its next call, which is when it sees the rising edge.
- **Error handling.** Any error stops the sequence in step 900, recording the step (PT601_ErrStep) and the ErrorID. Nothing is retried automatically: a sensor that refuses a parameter needs a person, not a loop that writes it every two seconds. PT601_Retry restarts the sequence.

ErrorID	Meaning
1	Timeout: no reply with our operation ID within <code>Timeout</code>
2	A read returned zero bytes (taken to mean the index was refused)
3	Bad argument
4	A module offset is outside the process image

- **Read-back after write.** A write the master accepted does not prove the device stored the value, for example if it was outside the device's range. Step `50` compares the read-back value and fails with `PT601_ErrID = 0` on a mismatch.
- **Replacement.** When event `16#1800` (port communication lost) appears for our port, `PT601_Ready` drops and the sequence returns to step 0. It waits there until the event has been acknowledged and the port has been up for 5 s, then parameterises the sensor again. A replacement sensor therefore receives the plant's switch point without anyone entering it. The master's own **Start-up ISDU** entries on the **IO-Link ports** page (Chapter 12) are an alternative for values that never change.
- **Text parameters.** Vendor and product names arrive as bytes in `RxData`. The dialect has no byte-to-character conversion, so they are stored as byte arrays. The HMI or SCADA displays them as text.

G.8.6 Task Configuration

Use `MainTask` at 10 ms with `IoISensor` in its call list. The ISDU and event modules travel in the cyclic process data, so the blocks make progress once per EtherCAT exchange. A slower task only makes the start-up take longer.

```
"task": { "period_ms": 10, "calls": [ "IoISensor.st" ] }
```

G.8.7 Testing in the simulator

The simulator replaces the EtherCAT master with the `sim` driver, keeping the same `%I` / `%Q` sizes. The program runs, but no IO-Link master answers. Useful tests are:

Step	Action	Expected result
1	Download, wait 5 s	<code>PT601_Step = 10</code> . The ISDU output bytes are written: check <code>%QB12 = 1</code> (port), <code>%QB13 / %QB14 = 0/16</code> (index <code>16#0010</code>), and <code>%QB18</code> = the operation ID
2	Wait 2 s more	<code>PT601_Step = 900</code> , <code>PT601_ErrStep = 10</code> , <code>PT601_ErrID = 1</code> (timeout). This is the no-master path
3	Force <code>%IB2 = 16#03</code> , <code>%IB3 = 16#E8</code>	<code>PT601_Raw = 1000</code> , <code>PT601_bar = 10.0</code> (checks the byte order of <code>IOL_IN_INT</code>)
4	Force <code>%IB74 = 16#31</code> , <code>%IB76 = 16#18</code> , <code>%IB77 = 16#00</code>	<code>Ev.Active = 1</code> , <code>PT601_DevError TRUE</code> , <code>PT601_CommLost TRUE</code> (type 3 = error, port 1, code <code>16#1800</code>)
5	Write <code>PT601_AckEvents TRUE</code>	Bits 0 and 2 of <code>%QB51</code> set (clear bit for event 1, toggle bit). The toggle flips on each acknowledgement

Step	Action	Expected result
6	To simulate a reply, first change <code>Timeout := T#2s</code> in the <code>Rd</code> call to <code>T#60s</code> and download. Write <code>PT601_Retry TRUE</code> → <code>FALSE</code> ; when <code>PT601_Step = 10</code> , force <code>%IB40</code> to the value in <code>%QB18</code> and <code>%IB41 = 2</code>	<code>Rd.Done TRUE</code> , <code>PT601_VendorLen = 2</code> , and the sequence moves on to step 20. Restore the timeout afterwards

On the bench with real hardware, work through the checklist in Chapter 12 before relying on the example. In particular, confirm how a refused index is reported (ErrorID 1 or 2), and compare `IOL_IN_INT` with the sensor's own display.

G.8.8 Variations

- **32-bit float process value.** `PT601_bar := IOL_IN_REAL(PD_IN_OFF);` for a sensor that sends Float32 (4 bytes, so a 4-byte port module).
- **Several sensors.** Put the sequence in a `FUNCTION_BLOCK FB_IoIDevice` with `Port`, `InBase` and `OutBase` inputs. All instances on one master share the ISDU module, so run them one after the other: start the next instance's sequence only when the previous one reports ready or failed.
- **Parameter sets from a recipe.** Replace the single `PT601_SP1_want` with a small table of `{Index, SubIndex, Length, Value}` rows, and loop over it with the same read / compare / write / read-back steps.

Appendix H. Controller configuration file

The IDE writes the controller's configuration for you when you use **Download hardware**; you rarely edit it by hand. This appendix lists what it contains, so that an engineer reading one — in a project folder (`controller.json`), or on a controller — knows what each section does. The section and key names below are the ones the runtime's configuration parser reads. The complete schema, with every default and validation rule, is document Zeus-PLC-04 (*Configuration schema*).

H.1 Sections

Section	Contains
(<i>top level</i>)	<code>name</code> — the controller name shown in the IDE and in logs
<code>process_image</code>	<code>input_bytes</code> , <code>output_bytes</code> (normally derived from the IO devices), <code>marker_bytes</code> (%M, default 4096), <code>retain_file</code> , <code>retain_interval_ms</code>
<code>task</code>	<code>period_ms</code> / <code>period_us</code> , <code>rt_priority</code> , <code>cpu</code> , <code>overrun_policy</code> (<code>skip</code> , <code>catchup</code> or <code>fault</code>), <code>overrun_fault_limit</code> , <code>io_startup_grace_ms</code> (default start-up wait for critical devices, 30000), <code>lock_memory</code> , <code>prefault_stack_kb</code> , <code>programs[]</code> (<code>name</code> , <code>rate</code> , <code>offset</code>), task groups
<code>log</code>	<code>level</code> , <code>file</code> , <code>audit_file</code> , <code>rotation</code> , <code>stderr</code> / <code>syslog</code> output
<code>monitor</code>	The online connection used by the IDE: <code>enabled</code> , <code>bind</code> , <code>port</code> (5020), <code>allow_write</code> , <code>require_token</code> , <code>token</code> , <code>max_clients</code> , <code>cpu</code> , <code>identities[]</code> , <code>users[]</code> (name, role, salted hash, iterations, <code>must_change</code>)
<code>opcua</code>	<code>enabled</code> , <code>port</code> (4840), anonymous access, <code>security_required</code> , <code>expose[]</code> (published variables, optionally <code>writable</code>), <code>expose_all</code> , <code>users[]</code> , <code>cpu</code>
<code>enip_tags</code>	EtherNet/IP tag server: <code>enabled</code> , <code>bind</code> , <code>port</code> (44818), <code>allow_write</code> / <code>read_only</code> , <code>max_clients</code>
<code>modbus_slave</code>	<code>enabled</code> , <code>bind</code> , <code>port</code> (502), <code>unit_id</code> , <code>read_only</code> , register map, <code>cpu</code>
<code>web</code>	Operator screens server: <code>enabled</code> , <code>port</code> , <code>root</code> , <code>allow_write</code> , <code>cpu</code>
<code>leds</code>	RUN and IO lamp GPIOs, active level, blink periods
<code>clock</code>	Clock memory in %M: <code>byte</code> (the clock-memory byte) and <code>pulses[]</code> (<code>bit</code> , <code>period_ms</code>)
<code>alarms[]</code>	Controller alarms: <code>id</code> , <code>name</code> , <code>message</code> , <code>priority</code> (<code>diagnostic</code> , <code>low</code> , <code>high</code> , <code>critical</code>), <code>area</code> / <code>offset</code> / <code>bit</code> , <code>invert</code> , <code>latched</code> , <code>delay_ms</code> , <code>suppress</code>
<code>batch_context</code>	Where the batch context (unit, phase, batch id) is published: <code>enabled</code> , <code>area</code> , <code>offset</code>
<code>security</code>	<code>program_verify</code> (<code>off</code> , <code>warn</code> , <code>enforce</code>) and <code>program_key</code> for signed programs
<code>redundancy</code>	Role, peer, checkpoint and takeover timings (redundancy is in development)
<code>io[]</code>	One entry per IO device: <code>name</code> , <code>driver</code> , <code>mode</code> (<code>sync</code> / <code>async</code>), <code>enabled</code> , <code>inhibited</code> , <code>critical</code> , <code>startup_timeout_ms</code> (0–600000 ms a critical device may take to connect at start before FAULT; absent = <code>task.io_startup_grace_ms</code> , 0 = must be up on the first scan), <code>module_policy</code> , <code>output_on_stop</code> / <code>output_on_fault</code> (<code>clear</code> or <code>hold</code>), <code>hold_last_on_fault</code> , <code>degrade</code> , <code>cpu</code> , <code>rt_priority</code> , <code>scale[]</code> , <code>params</code> (driver-specific)

H.2 `io[].scale[]` — signal ranges

Key	Meaning
<code>area</code>	"I" or "Q"
<code>offset</code>	Byte offset of the first channel, relative to the device's own slice (like the block offsets)
<code>count</code>	How many consecutive channels share this range
<code>width</code>	16 or 32
<code>signed</code>	Whether the card's counts are signed
<code>raw</code>	<code>[low, high]</code> — counts the card produces at the ends of its range
<code>eng</code>	<code>[low, high]</code> — what those ends mean, e.g. <code>[4000, 20000]</code> for 4–20 mA in μ A
<code>clamp</code>	Clamp out-of-range counts

Up to 32 scalings per device.

H.3 Driver parameters (`io[].params`) — main keys

Driver	Main parameters
<code>modbus_tcp</code>	Host, port, unit id, poll period, timeout, <code>blocks[]</code> (function code, start, count, area, offset)
<code>ethercat</code>	<code>ifname</code> , <code>cycle_us</code> , <code>dc</code> , <code>expected_slaves</code> , <code>require_bus</code> , <code>modules[]</code> , <code>pdo_assign</code> , <code>startup_sdo[]</code> , <code>override_input_bytes</code> / <code>override_output_bytes</code> , <code>blocks[]</code>
<code>profinet</code>	<code>ifname</code> , <code>station_name</code> , <code>mac</code> , <code>ip</code> , <code>vendor_id</code> , <code>device_id</code> , <code>gsdml</code> , <code>send_clock</code> , <code>reduction_ratio</code> , <code>watchdog_factor</code> , <code>modules[]</code> , <code>expected[]</code> , override sizes, <code>blocks[]</code>
<code>sim</code>	Simulated IO used by the simulator

`blocks[]` are the addressed IO blocks (name, area, offset, count, width, aliases) that the device tree and the alias binding use; see Chapter 11. A block's `offset` is relative to its device: the runtime and the IDE add the device's base (Chapter 7, Section 7.8).