

Whisker IDE User Manual

Cloud and Offline editions

D6 Labs

October 2026

1 Getting Started

1.1 About this manual

1.2 System requirements

1.3 Downloading the IDE

1.4 Installing the IDE

1.4.1 Step 1 — Extract the zip

1.4.2 Step 2 — Run the installer

1.4.3 Step 3 — Confirm the install

1.5 First launch

1.5.1 Cloud edition — sign in

1.5.2 Offline edition

1.5.3 The first workspace

1.6 Core concepts

1.6.1 I/O acronyms used in this manual

1.6.2 Arena memory

1.7 A tour of the workspace

1.8 Three ways to run a program

1.9 What's next

2 The Workspace

2.1 Anatomy of the main window

2.2 Toolbar

2.2.1 The monitoring gate

2.3 Project tree

2.4 Cloud Explorer (Cloud edition)

2.5 Editor area

2.6 Properties panel

2.6.1 Typing a tag that does not exist

2.7 Toolbox

2.8 Bottom panel

2.9 Status bar

2.10 Dock layout

2.11 Keyboard shortcuts

2.12 Theme

3 Projects and Applications

3.1 Application versus Project

- 3.2 The `.widez` file
- 3.3 Creating an Application
- 3.4 Adding a Project later
- 3.5 Opening an Application
 - 3.5.1 Sample projects
- 3.6 The active project
- 3.7 Saving
- 3.8 Removing a Project
- 3.9 Changing a Project's target
- 3.10 The Design Spec
- 3.11 Your project on the controller
- 3.12 Project settings
- 3.13 Where the IDE keeps its own settings
- 4 Setting Up Your Nexus.io Automation Controller
 - 4.1 Out of the box
 - 4.2 Choose a commissioning path
 - 4.3 Powering on
 - 4.4 The Settings screen
 - 4.5 Network settings
 - 4.5.1 DHCP (the default)
 - 4.5.2 Static address
 - 4.6 Finding the controller from the IDE
 - 4.7 RS-485 and the Modbus server
 - 4.8 Operator login at the panel
 - 4.8.1 When the cloud is unreachable
 - 4.9 Screen sleep
 - 4.10 Before you open the IDE
- 5 Target Hardware in the IDE
 - 5.1 How the IDE knows a target
 - 5.2 Nexus.io Automation Controller (`Nexus.io AC`)
 - 5.3 SmartController (Classic) (`SmartController Classic`)
 - 5.4 WhiskerHMI (`WhiskerHMI`)
 - 5.5 Choosing a target
 - 5.6 What changes with the target

6 Tag Database

6.1 What a tag is

6.1.1 Data types

6.1.2 Classes

6.1.3 Addresses

6.2 Opening the editor

6.2.1 Columns

6.2.2 Toolbar

6.3 Creating a tag

6.4 Importing and exporting tags

6.5 Editing a tag

6.6 Tags created from the Properties panel

6.7 The undefined-tag sweep

6.8 System tags

6.9 Process variables

6.10 Modbus addresses and renumbering

6.11 History

6.12 Retained variables

6.13 Tag Cross Reference

6.14 Naming conventions

7 I/O Configuration and Devices

7.1 Where I/O comes from

7.2 Opening the editor

7.2.1 Serial port settings

7.3 Adding a device

7.3.1 Modbus devices from an IODF

7.3.2 Arena TCP devices

7.3.3 EtherNet/IP PLCs

7.4 The Configured Devices table

7.5 I/O points and their tags

7.6 System tags for every device

7.7 Importing tags from another controller

7.8 Adding EtherNet/IP tags

7.9 Banner wireless I/O

7.10 IODF files

7.11 Building after I/O changes

8 Ladder Logic Editor

- 8.1 Programs, tasks and functions
- 8.2 Anatomy of a network
- 8.3 The Toolbox
- 8.4 Placing elements
- 8.5 Selecting and the Properties panel
- 8.6 Branches
- 8.7 Deleting
- 8.8 Networks
- 8.9 Undo, redo and the clipboard
- 8.10 Comparisons
- 8.11 Tags from the editor
- 8.12 Building, emulating and monitoring
- 8.13 Keyboard reference

9 Function Blocks, Faceplates and Tasks

9.1 Tasks

- 9.1.1 Task kinds
- 9.1.2 Creating a task
- 9.1.3 Task Configuration
- 9.1.4 Structuring a program with freewheel tasks
- 9.1.5 Choosing a period

9.2 Functions

9.3 Function blocks

9.4 Faceplates

- 9.4.1 Creating a faceplate from the ladder
- 9.4.2 Placing faceplates from the HMI palette
- 9.4.3 What the operator sees

9.5 Python as a callable block

10 HMI Designer

- 10.1 Concepts
- 10.2 Screens
- 10.3 The ELEMENTS palette
- 10.4 Placing and arranging elements
- 10.5 Element properties
- 10.6 Navigation between screens
- 10.7 Previewing screens

10.8 Operator login at the panel

11 Alerts Editor

11.1 What an alert is

11.2 Opening the editor

11.3 Creating an alert

11.4 Alert properties

11.4.1 Alert

11.4.2 Output

11.4.3 Behavior

11.4.4 Rule

11.5 How alerts run

11.6 Alert history at runtime

11.7 Removing an alert

12 Modbus Server and Map Viewer

12.1 How the server works

12.2 Exposing a tag

12.3 Address layout and data types

12.4 Auto-numbering

12.5 The Modbus Map view

12.6 Exporting the map for the SCADA engineer

12.7 When the SCADA cannot connect

13 Python on the Controller

13.1 The Python editor

13.2 Part 1 — Python on the Nexus.io controller

13.2.1 app.py and run(ctx)

13.2.2 The ctx API

13.2.3 Deploying Python

13.2.4 Testing without a controller

13.2.5 The Python Log

13.2.6 Python function blocks (PYFB)

13.3 Part 2 — SmartController (Classic)

13.3.1 Project files

13.3.2 Connecting over serial

13.3.3 The File System tree

13.3.4 Sync

13.3.5 The console

13.3.6 What is different from the Nexus.io controller

14 Connecting, Deploying and Emulating

14.1 Before you connect

14.2 Discovering controllers

14.2.1 Connecting by address

14.3 Which identity the IDE presents

14.4 What happens after connect

14.5 Deploying with Build

14.5.1 The signed bundle

14.5.2 Retained values

14.6 Run, Stop and Debug

14.7 Why monitoring is off until you deploy

14.8 Build & Emulate

14.9 Disconnecting

14.10 A typical session

15 Monitoring and Debugging

15.1 The monitoring gate

15.2 Where live values appear

15.3 The Tag Monitor panel

15.4 Debug Mode shading in the ladder editor

15.5 Writing and forcing values

15.6 The Output panel

15.7 The Statistics panel

15.8 The Python Log panel

15.9 Historian and trends

15.10 Diagnostic recipes

15.10.1 Start with the system tags

15.10.2 “The program is deployed but nothing happens”

15.10.3 “A value is wrong by a scale factor”

15.10.4 “An alert will not clear”

15.10.5 “Monitoring stops by itself”

15.10.6 “The connection drops”

15.10.7 “The panel HMI shows old values”

16 Security, Provisioning and Enrolment

16.1 The three postures

16.2 Path A — cloud provisioning and enrolment

16.2.1 Provision New Device

16.2.2 The reconnect moment

16.2.3 The one-time nonce dialog

16.2.4 Device actions in the Cloud panel

16.2.5 Revocation and renewal

16.3 The Security dialog — Account section

16.3.1 Panels — identities for WhiskerHMI PCs

16.4 Path B — standalone, and the optional site CA

16.4.1 Who is the administrator

16.4.2 Make this keystore a site

16.4.3 Issue engineer identity

16.4.4 Import site identity on the engineer's PC

16.4.5 Provision unit (site)

16.4.6 Panels — identities for WhiskerHMI PCs

16.4.7 Revoke, the CRL and the seven-day warning

16.5 Local keystore — Offline edition and bench units

16.6 PIN login at the panel

16.7 Signed deploys and what a controller refuses

17 Generating HMI Installers

17.1 What the installer contains

17.2 Before you generate

17.3 The Generate Installer dialog

17.3.1 If generation fails

17.3.2 Test the installer before hand-off

17.4 What the customer sees

17.5 Changing the configuration on site

17.6 Updating an installed application

18 Project Properties Reference

18.1 Identity

18.2 HMI Settings

18.3 Deployment

18.4 Cloud

18.5 Device Settings (Classic only)

18.6 Modbus Server (Nexus.io only)

- [18.7 CIP tag server \(Nexus.io only\)](#)
- [18.8 Display \(WhiskerHMI only\)](#)
- [18.9 Authentication \(WhiskerHMI only\)](#)
- [18.10 Security \(WhiskerHMI only\)](#)
- [18.11 Window Layout \(WhiskerHMI only\)](#)
- [18.12 The Application node](#)

[19 Ladder Instruction Reference](#)

- [19.1 How to read the tables](#)
- [19.2 Basic: contacts and coils](#)
 - [19.2.1 NO Contact, NC Contact](#)
 - [19.2.2 P Contact, N Contact \(edge\)](#)
 - [19.2.3 Coil, Neg Coil](#)
 - [19.2.4 Set Coil, Reset Coil](#)
 - [19.2.5 Flash](#)
- [19.3 Timers](#)
 - [19.3.1 TON — On-Delay Timer](#)
 - [19.3.2 TOF — Off-Delay Timer](#)
 - [19.3.3 TP — Pulse Timer](#)
 - [19.3.4 TONR — Retentive Timer](#)
- [19.4 Counters](#)
 - [19.4.1 CTU — Count Up](#)
 - [19.4.2 CTD — Count Down](#)
 - [19.4.3 CTUD — Count Up/Down](#)
- [19.5 Edge detection and latches](#)
 - [19.5.1 R_TRIG, F_TRIG](#)
 - [19.5.2 SR — Set-Dominant Latch, RS — Reset-Dominant Latch](#)
- [19.6 Compare](#)
 - [19.6.1 EQ, NE, GT, GE, LT, LE — inline comparisons](#)
 - [19.6.2 CMP — Compare Block](#)
 - [19.6.3 MEQ — Masked Equal](#)
- [19.7 Math](#)
 - [19.7.1 MATH operations](#)
 - [19.7.2 LIMIT — Clamp to Range](#)
 - [19.7.3 SEL — Select by Boolean](#)

19.7.4 MUX — Select 1 of 8

19.7.5 PACK, UNPACK

19.8 Process

19.8.1 SCALE — Linear Scaling

19.8.2 HYST — Deadband Comparator

19.8.3 TOTAL — Totalizer

19.8.4 FILTER — First-Order Lag

19.8.5 ROC — Rate of Change

19.8.6 PID — Closed-Loop Controller

19.8.7 ALT — Lead/Lag Pump Alternator

19.9 Industrial

19.9.1 DEBOUNCE — Input Filter

19.9.2 MOTOR — Motor Device Block

19.9.3 VALVE — Valve Device Block

19.9.4 FIRSTOUT — First-Out Interlock Annunciator

19.10 Sequencing and time

19.10.1 DRUM — Step Sequencer

19.10.2 RTC — Wall Clock

19.10.3 SCHED — Time-of-Day Window

19.11 Program flow

19.11.1 PYFB — Python Function

19.11.2 CALL, RET

19.12 Common traps

20 Historian and Trends

20.1 Two historians

20.2 Recording history on the controller

20.2.1 Choosing the tags

20.2.2 Retention

20.2.3 What is stored

20.3 The Trend Chart widget

20.4 History on the WhiskerHMI PC

20.4.1 The Trends dialog

20.4.2 Exporting history to CSV

20.5 Alert history

21 The WhiskerHMI PC Application

21.1 Starting the application

21.2 Connecting to the controller

21.3 Panel identity — connecting to a managed controller

21.3.1 What the build carries

21.3.2 Issuing a panel identity

21.3.3 Importing the identity on the panel PC

21.3.4 Revoking a panel

21.3.5 Messages and what they mean

21.4 Windows and the layout editor

21.5 The status bar

21.6 Trends and CSV export

21.7 Operator login and roles

21.7.1 What the operator sees

21.8 The audit log

21.9 Files on the PC

22 The AI Designer and AI Fix

23 CIP Tag Server

23.1 How the server works

23.2 Turning the server on

23.3 Publishing a tag

23.4 What a client may and may not write

23.5 Setting up an AVEVA ABCIP driver

23.6 What a refused write looks like

23.7 When the SCADA cannot connect

24 The Walkthrough scenario

24.1 What you're building

24.2 Choose your path

24.3 The hardware

24.3.1 Wiring map

24.3.2 RS-485 settings

24.4 Tags — where they come from

24.5 The control logic — what it does

24.6 The alarm

24.7 The HMI screen

24.8 Watching the live data — Tag Monitor

24.9 What's next

25 Your First PLC Project — Ladder

25.1 Before you start

25.2 Step 1 — Create the Application and Project

25.3 Step 2 — Add the mSmart Universal IO module

25.4 Step 3 — Rename the I/O tags you'll use

25.5 Step 4 — Add the memory tags

25.6 Step 5 — Write the ladder logic

25.6.1 Network 1 — Decode the operator mode (CMP)

25.6.2 Network 2 — Latch the alarm

25.6.3 Network 3 — Clear the alarm latch

25.6.4 Network 4 — Auto-release the Clear Alarms button (one-shot)

25.6.5 Network 5 — call_motor SET (start float closes the tank's request)

25.6.6 Network 6 — call_motor RESET (stop float clears the request)

25.6.7 Network 7 — Drive `motor_run`

25.7 Step 6 — Define the HighLevelAlarm alert

25.8 Step 7 — Build the HMI screen

25.9 Step 8 — Connect and Build

25.10 Step 9 — Test it

25.11 What's next

26 Your First PLC Project — Python

26.1 Before you start

26.2 Step 1 — Create the Application and Project

26.3 Step 2 — Add the mSmart Universal IO module

26.4 Step 3 — Rename the I/O tags

26.5 Step 4 — Add the memory tags

26.6 Step 5 — Write the Python application

26.7 Step 6 — Define the HighLevelAlarm alert

26.8 Step 7 — Build the HMI screen

26.9 Step 8 — Connect and Build

26.10 Step 9 — Test it

26.11 What Python gives you that ladder doesn't

26.12 What's next

27 Your First WhiskerHMI Project

27.1 Before you start

27.2 Step 1 — Reopen the Walkthrough-Ladder Application

27.3 Step 2 — Add a WhiskerHMI project to the Application

27.4 Step 3 — Add an ArenaTCP device pointing at the controller

27.5 Step 4 — Import tags from the Motor project

27.6 Step 5 — Copy the HMI screen from Motor

27.7 Step 6 — Build the Panel runtime

27.8 Step 7 — Generate the Windows installer

27.9 Step 8 — Install and run on a PC

27.10 Step 9 — Test it

27.11 What's next

28 Where to go next

28.1 The full user manual

28.2 Exposing tags to Modbus

28.3 The Design Spec

28.4 Connecting to a controller

28.5 Sample projects

28.6 When you get stuck

28.7 Support

28.8 What we'd love to hear about

Appendix A — Keyboard Shortcuts

28.9 Global

28.10 Ladder editor

28.11 HMI Designer

28.12 Python editor

28.13 Design Spec editor

28.14 SmartController Classic Console

28.15 Mouse

Appendix B — File Formats Reference

28.16 Application and project archives

28.16.1 `project.yaml` keys worth knowing

28.17 `symbols.yaml`

28.18 Build output

28.18.1 The signed bundle

28.19 Provisioning and security files sent to a controller

28.20 The keystore on your PC

28.20.1 Site packages

28.21 Export formats

Appendix C — Troubleshooting

28.22 Installing and starting

28.22.1 Windows warns that the installer might be dangerous

28.22.2 The IDE starts, then closes at once

28.23 Connecting

28.23.1 The Connect to Target dialog finds nothing

28.23.2 The row says “Awaiting enrolment”

28.23.3 I connected plainly and want the controller managed

28.23.4 “Connection failed” with a TLS or handshake error

28.23.5 Connecting by address falls back to plain and then is refused

28.23.6 “Connected” but the toolbar buttons stay grey

28.23.7 The connection drops as soon as the controller enrolls or is provisioned

28.23.8 The connection drops repeatedly for no visible reason

28.24 Enrolling and provisioning

28.24.1 Provision New Device showed a nonce dialog

28.24.2 Enrolment never completes

28.24.3 Provision unit (site) says “Connect to the unit first”

28.24.4 Provision unit (site) is refused with “bound to another CA”

28.24.5 The Site card warns the revocation list is N days old

28.25 Building and deploying

28.25.1 Build is disabled

28.25.2 “Deploy stopped: This project requires a PIN at the panel...”

28.25.3 “N tags are used but not defined”

28.25.4 Build failed with compiler errors

28.25.5 The deploy is refused: signature, chain or “signed deploys only”

28.25.6 Any IDE on the network can deploy to my controller

28.25.7 “Replace project on target?”

28.25.8 The deploy succeeds but the program does not change

28.25.9 “The VM would not start”

28.26 Monitoring

28.26.1 Monitoring shows nothing, the monitoring button is disabled

28.26.2 The Tag Monitor is empty or every value is a dash

28.26.3 The controller holds a different project

28.26.4 A value will not change when the ladder should write it

28.27 Emulation

28.27.1 The emulator will not start

28.27.2 “Python application not running: ...” after Build & Emulate

28.28 Runtime behaviour

28.28.1 A Modbus client sees no registers

28.28.2 Modbus RTU modules on an RS-485 port never answer

28.28.3 InTouch changed dotted tag names to underscores

28.28.4 An EtherNet/IP driver cannot connect, or sees no tags

28.28.5 The EtherNet/IP driver connects but every read fails

28.28.6 A value poked from the SCADA snaps back

28.28.7 The historian or a trend chart is empty

28.28.8 The panel’s status bar shows “No cell”

28.28.9 An alert will not clear

28.28.10 PIN login at the panel is locked out

28.28.11 The panel shows a blank screen after a deploy

28.29 WhiskerHMI panel identity

28.29.1 “No panel identity installed on this PC — use Import panel identity... in the panel’s menu”

28.29.2 Import failed: “passphrase incorrect”

28.29.3 Import failed: “this panel was built for another authority”

28.29.4 Import failed: the certificate is expired or not yet valid

28.29.5 The panel was connected and is now refused

28.29.6 The Panel menu is missing from the status bar

28.30 SmartController (Classic)

28.30.1 The controller restarts every 30 seconds with a blank screen

28.30.2 The console shows nothing, or “Connected” but nothing answers

28.30.3 Sync failed: “could not read the device’s / manifest.json”

28.30.4 After a Sync the controller still runs the old program

28.31 When all else fails

Appendix D — Glossary

Appendix E — Getting Help

28.32 Documentation

28.33 Support

28.34 Reporting bugs

28.35 Feature requests

28.36 Release notes

1 Getting Started

Welcome to the Whisker IDE — the design environment for the D6 Labs Nexus.io family of automation controllers. This chapter takes you from download to a first look at the workspace, introduces the handful of concepts the rest of the manual builds on, and points you to the hands-on walkthroughs.

1.1 About this manual

The Whisker IDE ships in two editions built from the same code:

- **Cloud edition.** Signs in to a Whisker.io account. Adds the Cloud Explorer, cloud provisioning of controllers, Publish to Cloud and the Account section of the Security dialog. It also has a **Continue Offline** button on the sign-in screen for days without a connection.
- **Offline edition.** No account and no cloud features. It installs as *Whisker PLC IDE Standalone*. Its Security dialog holds the site certificate authority and the local keystore, so a plant can run its own managed controllers without a cloud account.

This one manual covers both. Where a feature needs an account, the chapter says so; everything else — tags, ladder, HMI, alerts, Python, emulation, deploying — works the same in both editions.

1.2 System requirements

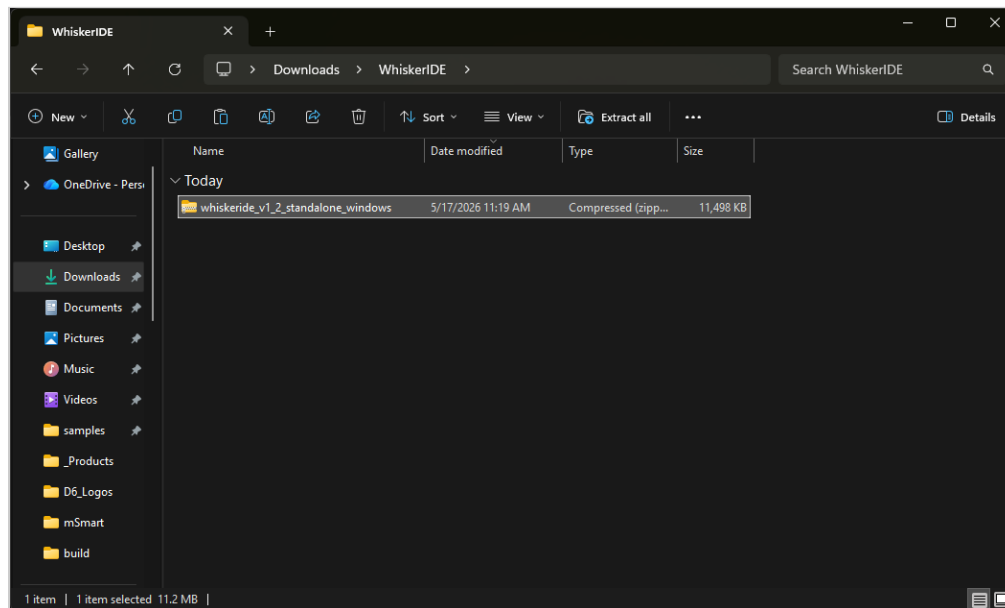
	Minimum	Recommended
Operating system	Windows 10 (64-bit), version 1809 or later	Windows 11
Processor	64-bit dual-core	Quad-core or better
Memory	4 GB RAM	8 GB RAM
Disk space	1 GB free	2 GB free (projects, HMI runtime bundles)
Display	1280 × 720	1920 × 1080 or larger
Network	Ethernet or Wi-Fi on the same LAN as your controller	Same, plus internet access for the Cloud edition

The IDE itself runs without an internet connection. You need the LAN to reach your controllers, and the Cloud edition needs internet access to sign in and to talk to Whisker.io.

1.3 Downloading the IDE

D6 Labs supplies the IDE as a single zip file per edition, named `whiskeride_v<version>_cloud_windows.zip` or `whiskeride_v<version>_standalone_windows.zip`. Ask your D6 Labs contact for the edition you have licensed. The zip contains the installer plus four short text files (README, LICENSE, CHANGELOG, THIRD-PARTY-NOTICES) and the Quick Start PDF.

When the download completes, locate the file in your **Downloads** folder (or wherever your browser saves downloads).



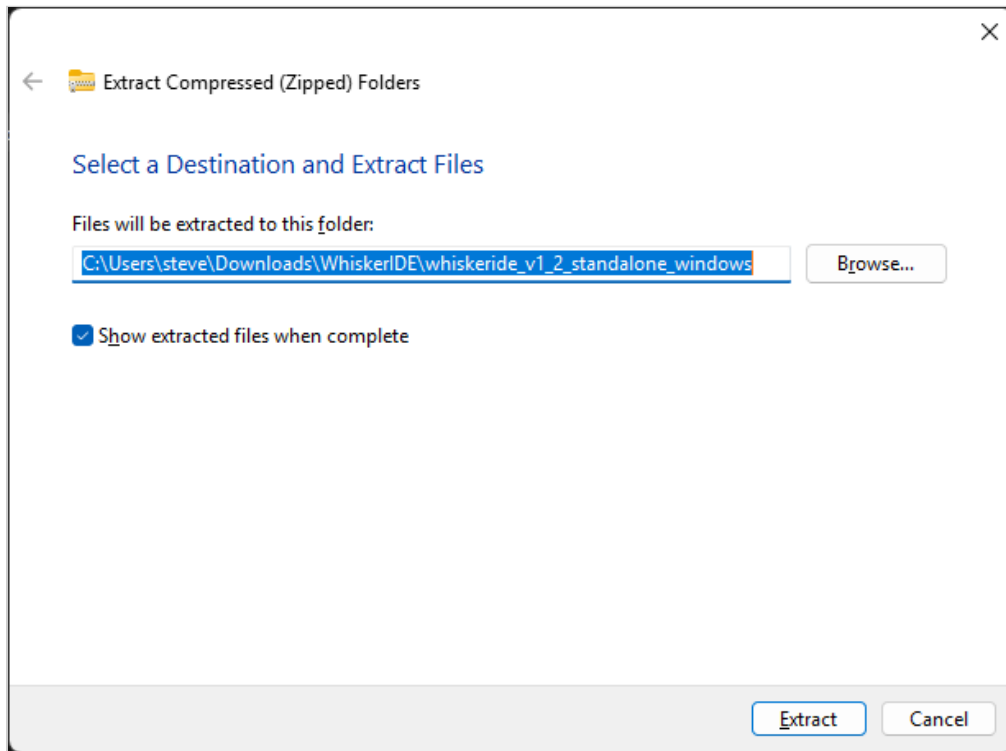
Downloaded zip in File Explorer

Note. If your browser warns that the file might be dangerous, choose **Keep**. Windows SmartScreen can flag any installer it has not seen many times before. Check that the file came from D6 Labs, then continue.

1.4 Installing the IDE

1.4.1 Step 1 — Extract the zip

1. Right-click the downloaded zip file.
2. Choose **Extract All...**
3. Accept the default destination and click **Extract**.



Windows Extract All dialog

The extracted folder contains:

File	What it is
WhiskerIDE-Cloud-<version>-Setup.exe or WhiskerIDE-Standalone-<version>-Setup.exe	The installer
README.txt	Quick reference matching this chapter
LICENSE.txt	End-user licence agreement
CHANGELOG.txt	Notable changes in this release
THIRD-PARTY-NOTICES.txt	Open-source component attributions

File	What it is
WhiskerIDE-QuickStart-v<version>.pdf	The Quick Start guide

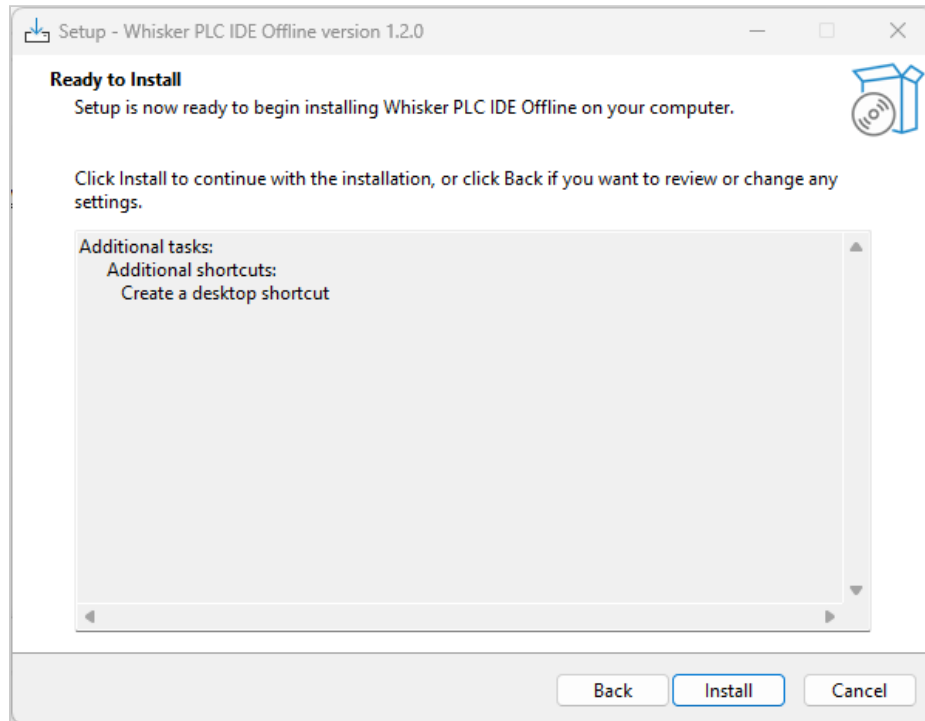
1.4.2 Step 2 — Run the installer

1. Double-click the `Setup.exe` file.

Note. Windows may show a “Windows protected your PC” warning because the installer was downloaded from the internet. Click **More info** → **Run anyway**. This appears *before* the User Account Control prompt in the next step.

2. Windows asks for permission to make changes. Click **Yes**.
3. The setup wizard appears. The defaults suit almost everyone:

Screen	What to do
License Agreement	Read it, choose <i>I accept</i> , click Next
Select Additional Tasks	Tick Create a desktop icon if you want one. Click Next
Ready to Install	Click Install
Completing Setup	Leave Launch Whisker IDE ticked. Click Finish



Setup wizard ready-to-install screen

4. When installation finishes the IDE launches.

1.4.3 Step 3 — Confirm the install

You should now have a Start menu entry named **Whisker PLC IDE** (Cloud edition) or **Whisker PLC IDE Standalone** (Offline edition), a desktop shortcut if you chose one, and an entry under **Settings** → **Apps**. The IDE is installed under *Program Files* in a folder of the same name. You can delete the zip and the extracted folder.

The installer also places the HMI runtime, the I/O scanner service and the PLC compiler alongside the IDE; you do not install those separately.

1.5 First launch

1.5.1 Cloud edition — sign in

The Cloud edition opens on a sign-in screen with **Email**, **Password**, a **Remember me** box, **Sign In**, and below a divider, **Continue Offline**.

Sign In does two things. It authenticates you with Whisker.io, and it fetches your **IDE certificate** — the credential a managed controller checks before it accepts your connection or your deploy. You do not set up certificates by hand; signing in

once per PC is enough. If your account uses two-factor authentication, a **Two-Factor Authentication** screen asks for the **Verification Code** from your authenticator app.

Continue Offline skips the account. You get the same workspace with the cloud features hidden; you can still emulate, connect to a standalone controller (or a managed one you hold a site identity for) and deploy. Sign in later from the **Cloud** indicator in the status bar.

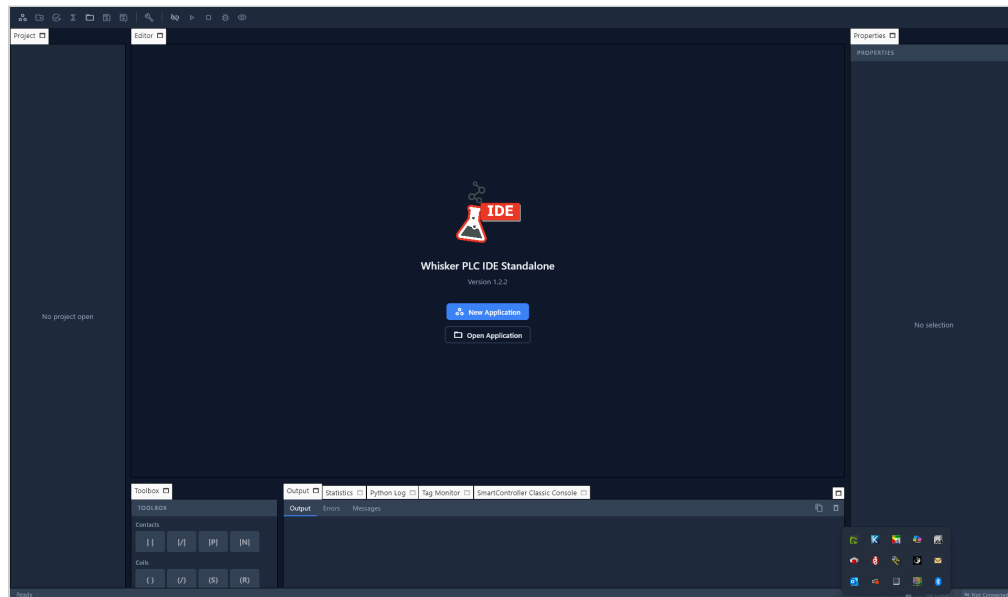
Note. Remember me keeps your e-mail address and password in Windows-protected storage, encrypted for your Windows user account, so the IDE can sign you in next time. Another Windows account on the same PC cannot read them, but anyone using *your* account can open the IDE signed in — do not tick it on a shared login. (Earlier releases kept them in plain files in Documents; the first start of this release moves them into protected storage and deletes the files.)

1.5.2 Offline edition

The Offline edition has no sign-in screen; it opens straight to the workspace.

1.5.3 The first workspace

Either way, you arrive at an empty workspace with two buttons in the editor area: **New Application** and **Open Application**.



Whisker IDE on first launch, empty workspace

On first launch the IDE also creates your default project folder,

DocumentsProjects, and copies two finished sample projects into it:

`walkthrough_ladder.widez` (the Quick Start's motor scenario in ladder logic) and `walkthrough_py.widez` (the same scenario in Python). **Open Application** starts in that folder, so opening a sample is a two-click way to see a complete project.

The window title tells you which edition you are running: **Whisker PLC IDE** or **Whisker PLC IDE Standalone**. When an Application is open, the title shows its file name; a leading `*` means there are unsaved changes.

1.6 Core concepts

Application — the file you open and save. An Application is a `.widez` archive that holds one or more Projects plus an Application-level Design Spec and a little metadata (customer, site, notes). A single-controller installation is an Application with one Project; a controller with an operator PC is an Application with two.

Project — one program for one controller (or one PC application). A Project has a Target, a Tag Database, and — depending on the Target — ladder programs and tasks, I/O configuration, HMI screens, alerts and Python files. Projects live inside the Application archive; they are not separate files on disk. See *Projects and Applications*.

Target — the kind of hardware a Project runs on. The **Target Hardware** dropdown offers three:

- `Nexus.io AC` — the **Nexus.io Automation Controller**, a panel PLC with a 10-inch touchscreen. The HMI screens you draw in the IDE are shown on that touchscreen; there is no separate HMI box.
- `SmartController Classic` — the **SmartController (Classic)**, D6 Labs' earlier MicroPython controller, programmed over USB serial with the Python editor. No display.
- `WhiskerHMI` — a **Windows application**, not hardware. The IDE builds your screens into a program you install on any Windows PC; it reads live tags from a Nexus.io controller over the network and shows the same screens the controller's own touchscreen can show. Install it on as many PCs as you need.

Target Hardware in the IDE describes each in detail.

Tag — a named variable. Tags have a type — BOOL, INT (16-bit), DINT (32-bit) or REAL — and an address in controller memory: %MX for BOOL, %MW for INT, %MD for DINT, %MF for REAL; physical inputs are %I... and outputs %Q... . Ladder logic, HMI widgets, alerts, the Modbus map and Python all refer to tags by name. See *Tag Database*.

Task — the schedule a ladder program runs on. There are three kinds: **Cyclic** (every N milliseconds — the `Main` task every new project starts with is cyclic at 10 ms), **Event** (once, when a chosen memory bit changes on a rising, falling or either edge) and **Freewheel** (as fast as the controller allows, for logic that paces itself). Tasks appear under **Tasks** in the Project Tree. See *Function Blocks*, *Faceplates and Tasks*.

Device — a piece of external I/O the controller talks to: a Modbus RTU module on RS-485, a Modbus TCP device, another Nexus.io controller (ArenaTCP) or an EtherNet/IP PLC. Adding a device creates its tags for you. See *I/O Configuration and Devices*.

Design Spec — a Markdown document attached to every Application and Project, with sections for overview, I/O, tags, control logic, screens and alarms. The IDE keeps its tag and Modbus tables up to date; the AI Designer reads it. See *Projects and Applications*.

1.6.1 I/O acronyms used in this manual

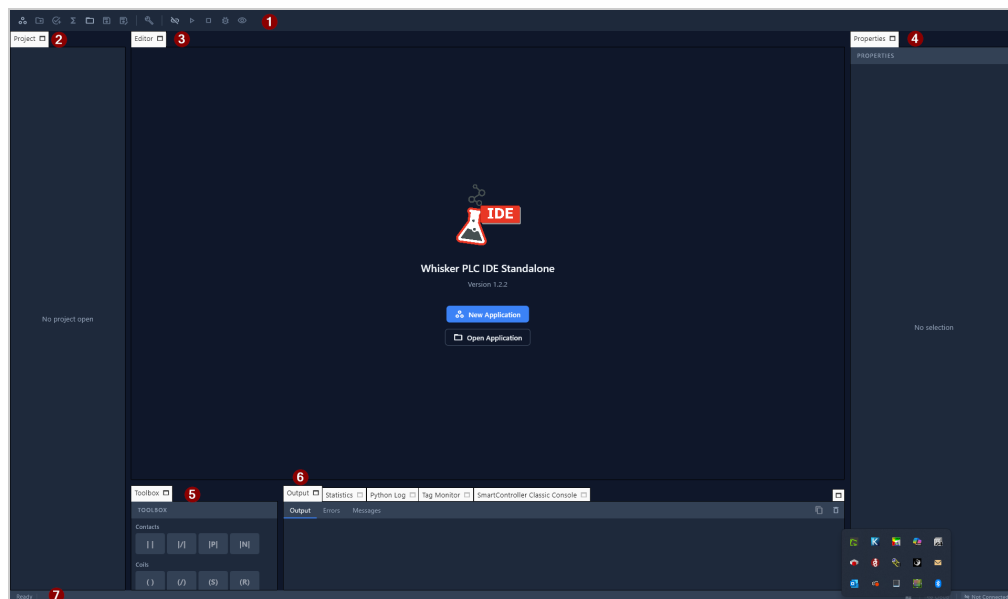
Acronym	Meaning	Example
DI	Digital input — on/off signal into the controller	float switch, push button, limit switch
DO (or DQ)	Digital output — on/off signal out of the controller	contactor coil, relay, indicator lamp
AI	Analog input — continuously variable signal in	4–20 mA pressure transmitter, 0–10 V level sensor
AO (or AQ)	Analog output — continuously variable signal out	4–20 mA valve command, 0–10 V speed reference

The IDE assigns the %I / %Q / %IW / %QW addresses when you add a device; you never type them by hand.

1.6.2 Arena memory

The **arena** is the controller's shared in-memory tag store: one block of RAM holding the current value of every tag. The ladder runtime writes it every scan; the HMI, the Modbus server, the Python task, the historian and the cloud agent all read and write the same block, which is why every part of the controller sees the same values. Day to day you can think of it as “where tag values live”. The name shows up in **Tag Monitor**, in **ArenaTCP** (one controller or PC reading another controller's arena) and in the Python `ctx.read_tag()` call.

1.7 A tour of the workspace



Annotated workspace with all panels labelled

1. **Toolbar** — file operations, Build, Build & Emulate, Connect, the controller's run/stop/debug modes and monitoring. The row changes with the active project's Target.
2. **Project tree** — the open Application and everything in its Projects. In the Cloud edition a **Cloud** tab beside it is the Cloud Explorer.
3. **Editor area** — the Tag Database, ladder programs, HMI screens and other editors, each in a tab.
4. **Properties panel** — click anything (a tag, a ladder element, a widget, the project itself) and edit it here. This is how all editing of existing items works; dialogs are only for creating and deleting.
5. **Toolbox** — what you can add to the active editor.

6. **Bottom panel** — **Output** (build and deploy messages), **Statistics**, **Tag Monitor** (live values), **Python Log**, and the others described in *The Workspace*.
7. **Status bar** — project name, cloud sign-in state and controller connection state.

The Workspace covers every region in detail.

1.8 Three ways to run a program

- **Build & Emulate** — compiles the open project and runs it on your PC. No controller needed; monitoring works against the emulator. This is the path to take if you are working without hardware.
- **Build (F5)** — for a Nexus.io project the button is enabled only while you are connected to a controller: press it and the IDE compiles and downloads the program to the connected controller in one action. Building for a SmartController (Classic) or a WhiskerHMI project needs no connection.
- **Run** — for a WhiskerHMI project, starts the PC application on this machine with the screens you just built.

Before any build, the IDE checks that every tag your logic refers to exists in the Tag Database and offers to create the missing ones.

1.9 What's next

The Quick Start walks you through one physical scenario — a motor, three float switches, an Off/Hand/Auto selector and a latched high-level alarm — three times over:

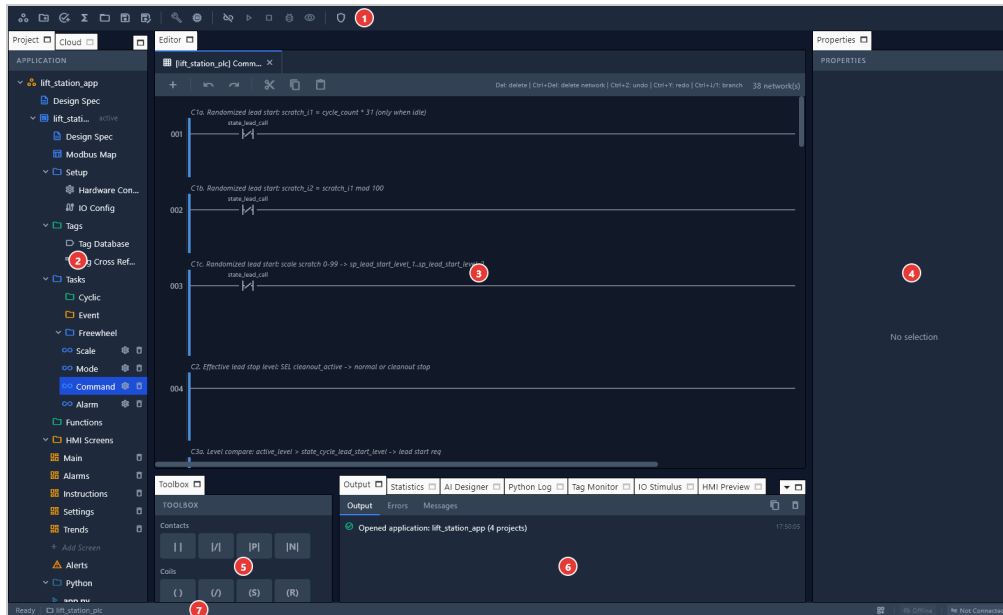
- *The Walkthrough scenario* sets up the hardware, wiring and tag names the walkthroughs share, and explains the two ways to start with a new controller: enrol it into your Whisker.io account, or simply connect to it as the standalone unit it ships as (with a site certificate authority you run yourself as the optional way to manage it without the cloud).
- *Your First PLC Project — Ladder* builds the control logic as ladder networks on a Nexus.io controller.
- *Your First PLC Project — Python* builds the same logic in Python on the same controller.
- *Your First WhiskerHMI Project* builds a Windows operator panel that connects to the controller over the network.

If you have a controller on your bench, start with *The Walkthrough scenario*. If you do not, the ladder and Python walkthroughs can be completed against the emulator with **Build & Emulate**; only the deploy and the physical test need hardware.

2 The Workspace

This chapter is a reference for every region of the Whisker IDE main window. Read it once end to end to learn the layout, then come back to individual sections as you need them.

2.1 Anatomy of the main window



1. **Toolbar** — file operations, build, connect and controller controls
2. **Project tree** — the open Application (plus the **Cloud** tab in the Cloud edition)
3. **Editor area** — open editors as tabs
4. **Properties panel** — properties of the selected item
5. **Toolbox** — palette for the active editor
6. **Bottom panel** — Output, Statistics, Tag Monitor and the other diagnostic tabs
7. **Status bar** — project, cloud and connection state

2.2 Toolbar

The buttons shown depend on the active project's Target and on whether you are connected. Hover a button for its tooltip; the tooltips are the names used throughout this manual.

Button (tooltip)	What it does	Shown / enabled
New Application	Creates an Application (see <i>Projects and Applications</i>)	Always
New Project (Ctrl+N)	Adds a project to the open Application	Enabled when an Application is open
New Task (Ctrl+Shift+N)	Adds a ladder task	Nexus.io projects
New Function (Ctrl+Alt+N)	Adds a function block	Nexus.io projects
Open Application (Ctrl+O)	Opens a .widez archive	Always
Save (Ctrl+S)	Saves the Application and every project in it	Enabled when there are unsaved changes
Save As (Ctrl+Shift+S)	Saves under a new name	A project is open
Build (F5)	Nexus.io: compiles and downloads the program to the connected controller. Classic: writes the application and configuration packages to a folder. WhiskerHMI: compiles only (tooltip Build)	Nexus.io: enabled only while connected. Others: whenever a project is open
Build & Emulate (run on this PC, no target needed)	Compiles and runs the program on this PC; becomes Stop Emulation while the emulator runs	Nexus.io projects; no connection needed
Run	Starts the WhiskerHMI PC application with the last build	WhiskerHMI projects
Generate Installer	Packages the WhiskerHMI application as a Windows installer	WhiskerHMI projects
Connect / Disconnect	Opens the Connect to Target dialog (Nexus.io) or Connect to SmartController Classic (serial)	All but WhiskerHMI

Button (tooltip)	What it does	Shown / enabled
Run Mode (F7)	Puts the controller in Run	Nexus.io, connected
Stop Mode (Shift+F7)	Puts the controller in Stop	Nexus.io, connected
Debug Mode (F9)	Run with values streamed to the IDE	Nexus.io, connected
Enable Monitoring / Disable Monitoring	Turns live values on in the editors and Tag Monitor	Nexus.io, connected — see the monitoring gate below
Publish to Cloud	Publishes the project to Whisker.io (Location, Device, Version, Release Notes)	Cloud edition, signed in
Security (shield)	Opens the Security dialog: your account certificate, the site CA, bench keystore	Cloud edition

Right of the buttons, in the Cloud edition, an access badge appears when the open Application is registered in the cloud and your account may not edit it: **Read-Only** (editing and deployment disabled) or **No Access**. A green **Connected** pill (with the serial port name for a Classic controller) shows at the far right while a connection is up. While the emulator is running the whole toolbar is tinted amber so you cannot mistake emulator values for a controller's.

2.2.1 The monitoring gate

Monitoring shows the controller's memory under your project's tag names. That is only truthful if the controller is running *this* project, so the IDE keeps **Enable Monitoring** disabled until this session has built and downloaded the open project to the connected controller and it has not been edited since. Hover the greyed button to see why — for example *This project has not been downloaded to the target yet. Use Build + Download to enable monitoring.* The same rule applies to Tag Monitor. Details: *Monitoring and Debugging*.

2.3 Project tree

The left panel is headed **APPLICATION** and shows the open Application as a tree. Under the Application node:

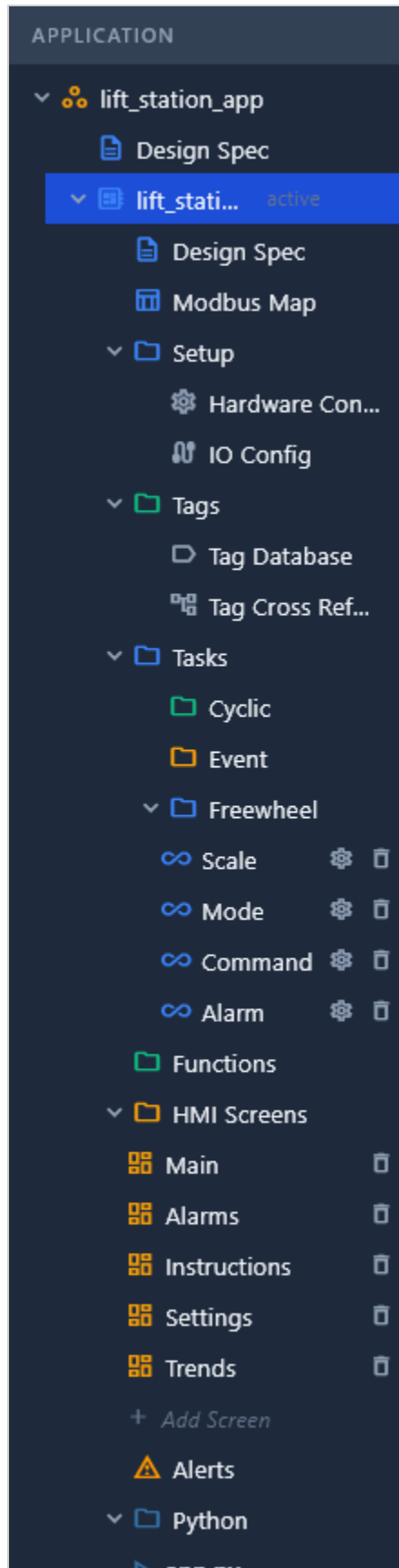
- **Design Spec** — the Application-level design document.

- One node per project, named *project (target)* where the target is shown as `Nexus.io AC`, `Classic` or `HMI`. The active project is marked *active*; click another project's node to make it active. Right-click a project for **Delete Project**.

Each project expands to the nodes its target supports:

Node	Nexus.io	Classic	WhiskerHMI
Files (the controller's file tree, with New File , New Folder and Sync to Target)	—	yes	—
Design Spec	yes	yes	yes
Modbus Map (appears once any tag is exposed)	yes	—	—
Setup → Hardware Config, IO Config	yes	—	yes
Tags → Tag Database, Tag Cross Reference	yes	yes	yes
Tasks → Cyclic, Event, Freewheel	yes	—	—
Functions	yes	—	—
HMI Screens (with Add Screen)	yes	—	yes
Alerts	yes	—	yes
Python (with Add Python File)	yes	—	—

A single click on a node selects it *and* opens its editor. Task, function and screen rows carry small gear (configure) and delete icons; screens can be copied and pasted between projects with the right-click menu.



2.4 Cloud Explorer (Cloud edition)

The **Cloud** tab beside the project tree is the **CLOUD EXPLORER**: your account's locations and, under each, its devices. The header has **New Location** (name, latitude, longitude) and **Refresh**. Right-click a location for **Provision New Device**; right-click a device for **Import Tag Configuration**, **Save Tag Configuration**, **Edit Device** (name, serial number) and, for Nexus.io controllers, **Rotate Device Cert**, **Decommission Device** and **Approve Re-Enrollment**. When you are not signed in it shows **Not logged in** and a **Log In** button. Provisioning and the certificate actions are explained in *Security, Provisioning and Enrolment*.

2.5 Editor area

Editors open as tabs in the centre. Which editor opens depends on the node you clicked:

- **Tag Database** and **Tag Cross Reference**
- **Ladder editor** — a task's or function's program
- **Hardware Config** — the project's target
- **IO Config** — devices and their points
- **Task Configuration / Function Configuration** — from the gear icon
- **HMI Designer** — one tab per screen
- **Design Spec** — Markdown, with View and Edit modes
- **Python editor** — one tab per file
- **Alerts editor**
- **Modbus Map** — read-only view with CSV export

Click a tab to switch and its ✕ to close it. A tab with unsaved changes shows a small dot beside its title. In an Application with several projects, tab titles are prefixed with the project name in square brackets. Tabs cannot be dragged to reorder.

2.6 Properties panel

The right-hand panel shows the editable properties of whatever is selected: a tag, a device or point, a ladder element, an HMI widget, a task, an alert, or the project or Application itself. **Select something, edit it here** is the IDE's one editing rule; dialogs are used only to create things (Add Device, New Task, Create New Tag) and to confirm deletions.

Double-clicking a ladder element brings the Properties panel to the front if you have docked it behind another panel; a single click only selects.

PROPERTIES

**Tank**
tank_wet

Position & Size

X

120

Y

80

Width

120

Height

260

Rotation (degrees)

0

Appearance

Shape

cylinder

Body Color

FF546E7A

Liquid Color

FF00B4D8

Show Level %

☒

Capacity

100.0

Tag Bindings

Level (real)

active_level

Raw Range

Min

to

Max

Scaled Range

0.0

to

100.0

Units

2.6.1 Typing a tag that does not exist

Any field in the Properties panel that takes a tag name accepts a name that is not in the Tag Database yet. When you leave the field, the IDE asks: *Tag “name” does not exist. Create it now?* The **Create New Tag** dialog is pre-filled with the name; choose **Data Type**, **Variable Class** and **Memory Area** (the address is assigned for you), add a description if you like, and click **Create**. **Cancel** leaves the name in the field and the tag undefined.

Undefined names do not get lost. When you **Save**, the IDE lists any tags used but not defined and lets you **Create All** or **Save Anyway**. **Build**, **Build & Emulate** and deploying show the same list and will not proceed until every tag exists.

2.7 Toolbox

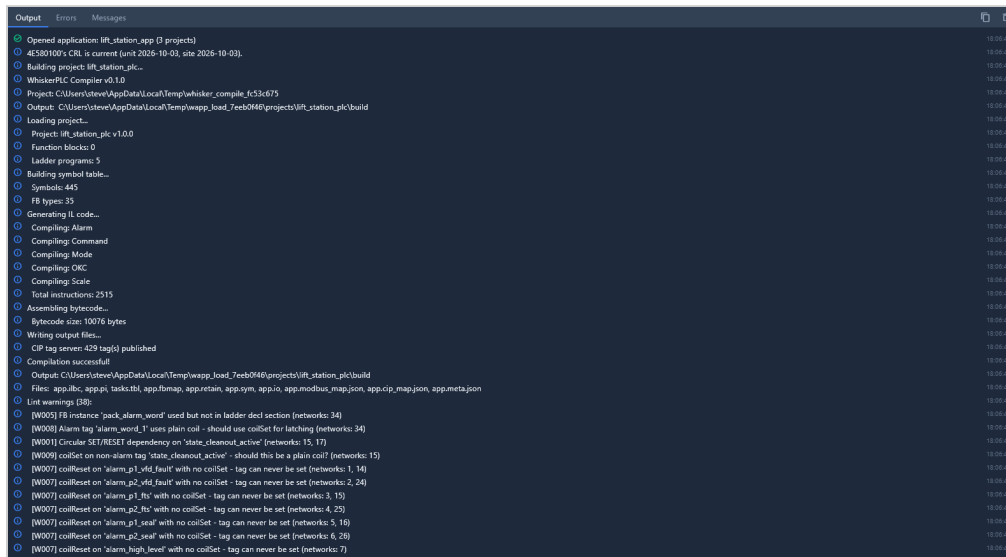
The Toolbox at bottom-left lists what you can add to the active editor: ladder elements (contacts, coils, timers, counters, comparison, math, function-block calls) when a ladder program is open; widgets (gauge, tank, LED, push button, selector switch, numeric input and display, text, trend chart, alert table, lock button and more) when a screen is open. How elements are placed is described in *Ladder Logic Editor* and *HMI Designer*.

2.8 Bottom panel

The tabs to the right of the Toolbox:

- **Output** — everything Build, Build & Emulate, Connect and deploy report, colour-coded: information, success (green), warning (amber), error (red). **Copy All** and **Clear** buttons in the header; right-click a line to **Copy** it.
- **Statistics** — a live dashboard of cards: the controller (status, mode, name, address, uptime, monitoring), a scan-time gauge with rate, minimum and maximum, a scan-time trend, the program on the controller (code size, tasks, function-block instances, scans), a Python loop-time gauge when a Python program is running, and the project's tags by type, programs, networks and HMI screens.
- **Python Log** — the live output of the running Python program, with copy and **Clear Log** buttons.
- **Tag Monitor** — every addressed tag with **Name**, **Type**, **Address**, **Value** and **Description**; a **Filter...** box; a toggle that hides or shows system-generated tags (hidden by default); and a button that copies the visible rows to the clipboard as tab-separated text. It reads *Not connected — no live values* until monitoring is on.

- **IO Stimulus** — inputs (DI and AI) as switches and number fields you can set by hand while watching the logic respond; outputs are shown read-only because the program computes them. It says *monitoring off* — *values are not live* until monitoring is enabled. See *Connecting, Deploying and Emulating*.
- **HMI Preview** — a zoomable preview of the screen being designed.
- **SmartController Classic Console** — the MicroPython REPL of a connected SmartController (Classic). Up and Down arrows recall earlier commands; Ctrl+C interrupts the running program.



```

Output  Errors  Messages
[+] Opened application 'Vlt_station_app' (3 projects)
[+] 2538100's CRL is current (unit 2026-10-03, size 2026-10-03).
[+] Building project 'Vlt_station_plc...'
[+] WhiskerPLC Compiler v0.1.0
[+] Project: C:\Users\steve\AppData\Local\Temp\whisker_compile_fc53c675
[+] Output: C:\Users\steve\AppData\Local\Temp\whisker_compile_fc53c675\projects\Vlt_station_plc\build
[+] Loading project...
[+] Project: Vlt_station_plc v1.0.0
[+] Function blocks: 0
[+] Ladder programs: 5
[+] Building symbol table...
[+] Symbols: 445
[+] FB types: 35
[+] Generating IL code...
[+] Compiling Alarm
[+] Compiling Command
[+] Compiling Mode
[+] Compiling OKC
[+] Compiling Scale
[+] Total instructions: 2515
[+] Assembling bytecode...
[+] Bytecode size: 10076 bytes
[+] Writing output files...
[+] CIP tag server: 429 tag(s) published
[+] Compilation successful
[+] Output: C:\Users\steve\AppData\Local\Temp\whisker_compile_fc53c675\projects\Vlt_station_plc\build
[+] Files: app\lib, app\cpu, tasks\lib, app\map, app\main, app\sym, app\io, app\modbus_map.json, app\cip_map.json, app\meta.json
[+] Link warnings (58):
[+] [W005] FB instance 'pack_alarm_word' used but not in ladder decl section (networks: 34)
[+] [W008] Alarm tag 'alarm_word_1' uses plain coil - should use coilSet for latching (networks: 34)
[+] [W001] Circular SET/RESET dependency on 'state_cleanout_active' (networks: 15, 17)
[+] [W009] coilSet on non-alarm tag 'state_cleanout_active' - should this be a plain coil? (networks: 15)
[+] [W007] coilReset on 'alarm_p1_and_purif' with no coilSet - tag can never be set (networks: 1, 14)
[+] [W007] coilReset on 'alarm_p2_and_purif' with no coilSet - tag can never be set (networks: 2, 24)
[+] [W007] coilReset on 'alarm_p1_fts' with no coilSet - tag can never be set (networks: 3, 15)
[+] [W007] coilReset on 'alarm_p2_fts' with no coilSet - tag can never be set (networks: 4, 25)
[+] [W007] coilReset on 'alarm_p1_seal' with no coilSet - tag can never be set (networks: 5, 16)
[+] [W007] coilReset on 'alarm_p2_seal' with no coilSet - tag can never be set (networks: 6, 26)
[+] [W007] coilReset on 'alarm_high_level' with no coilSet - tag can never be set (networks: 7)
  
```

2.9 Status bar

Left to right: **Ready**; the open project's name, with (*modified*) when it has unsaved changes; a **Reset layout (Ctrl+Shift+L)** button; the cloud sign-in state — your user name when signed in, **Offline** when you chose Continue Offline, or **Cloud** when not signed in (click it for **Log In / Log Out**); and the controller connection — **Not Connected**, **Connecting...**, **Connected**, **Connected (Running)**, **Connected (Debug)** or **Error**.

2.10 Dock layout

Every region is a docking panel. Drag a panel's tab or header to move it into another group, beside another panel, or into its own floating window; drag the splitters to resize. No panel can be closed. The IDE does not yet remember your arrangement between sessions — each launch starts from the default layout — and **Reset layout (Ctrl+Shift+L)** returns to it at any time.

2.11 Keyboard shortcuts

The shortcuts you will use most:

Key	Action
Ctrl+S	Save
Ctrl+O	Open Application
Ctrl+N	New Project
F5	Build
Ctrl+Shift+L	Reset layout

The full list, including the ladder and HMI editors' keys, is in the appendix *Keyboard Shortcuts*.

2.12 Theme

The IDE has a single dark theme; there is no light theme or theme setting.

3 Projects and Applications

The Whisker IDE organises your work in two levels: an **Application** that you open and save as a file, containing one or more **Projects**, each of which is a program for one controller or one PC application. This chapter explains the two, the file they live in, and every action you can take on them.

3.1 Application versus Project

A **Project** is one program for one target: a Tag Database, an I/O configuration and — depending on the target — ladder programs and tasks, HMI screens, alerts and Python files. A Project targets exactly one kind of hardware and is deployed to exactly one controller.

An **Application** groups the Projects that make up one system. A single controller is an Application with one Project. A lift station with a Nexus.io Automation Controller and an operator PC is one Application with two Projects: a `Nexus.io AC` project for the control logic and a `WhiskerHMI` project for the operator screens. Projects in one Application can see each other's tags — the WhiskerHMI project imports the controller's tags rather than retyping them — and HMI screens can be copied between them.

3.2 The `.widez` file

An Application is saved as a **single `.widez` archive**. Inside it are an `application.yaml` (name, version, metadata, the Application design spec and the list of projects) and one folder per Project under `projects/`, each holding that Project's `project.yaml`, `symbols.yaml` (tags), `io_map.yaml`, `tasks.yaml`, `functions.yaml`, ladder programs, HMI screens, `design_spec.md`, Python files and `alerts.yaml`. Projects are not separate files on disk; the archive is the unit you copy, back up and send to a colleague. The layout is documented in the appendix *File Formats*.

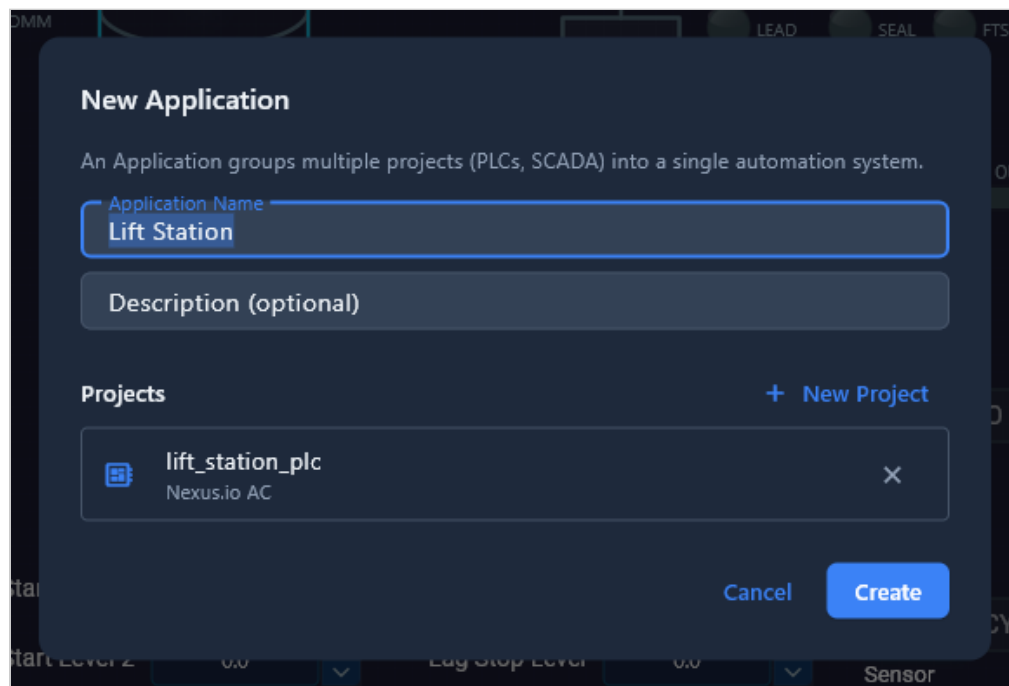
Tip. Treat the archive as opaque. Do not unzip, edit and re-zip it; the IDE rewrites the whole file on every Save. To inspect the contents, unzip a copy.
{.tip}

3.3 Creating an Application

Click **New Application** on the toolbar (or on the empty workspace). The **New Application** dialog asks for:

- **Application Name** — required.
- **Description (optional)**.
- **Projects** — click **New Project** to add each one; the nested dialog asks for a **Project Name** (letters, numbers and underscores, starting with a letter) and the **Target Hardware**: `Nexus.io AC`, `SmartController Classic` or `WhiskerHMI`.

Click **Create**. The Application opens in the Project tree. Nothing is on disk yet — press **Ctrl+S** to save it.



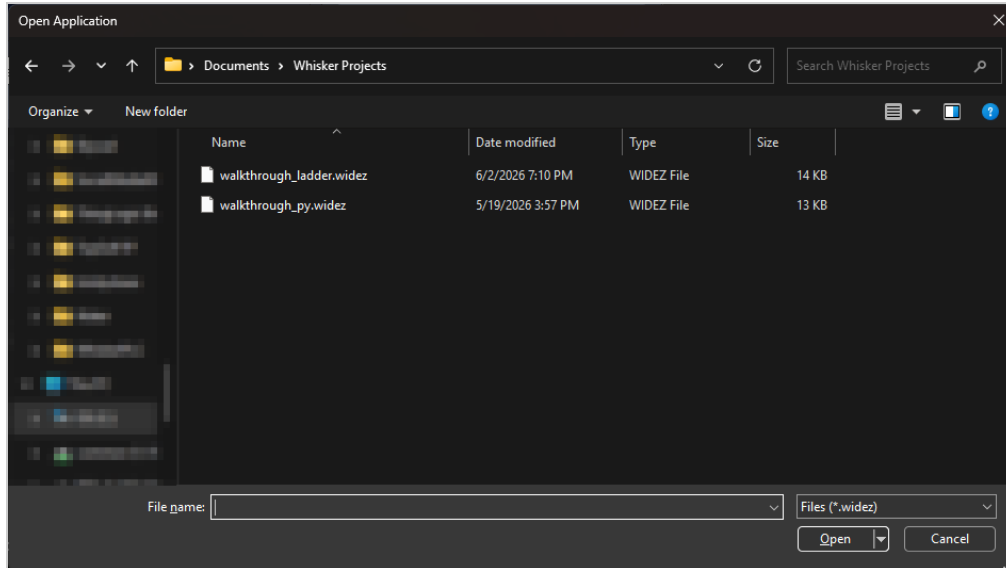
3.4 Adding a Project later

With an Application open, click **New Project (Ctrl+N)**. The dialog asks for **Project Name** (unique within the Application), **Target Hardware** and an optional **Description**; click **Create**. The project appears under the Application node and becomes the active project.

Note. **New Project** is disabled until an Application is open. Every Project lives inside an Application; there is no stand-alone project file.

3.5 Opening an Application

Click **Open Application (Ctrl+O)**. The file picker is filtered to `.widez`. The first time, it starts in your default project folder, **DocumentsProjects**; after that, Windows reopens the folder you last opened an Application from. If another Application is open and has unsaved changes you are asked to save first.



The IDE does not register the `.widez` extension with Windows, so double-clicking an archive in Explorer does not open it; use the IDE.

3.5.1 Sample projects

On first launch the IDE created **DocumentsProjects** and copied two finished samples into it: `walkthrough_ladder.widez` and `walkthrough_py.widez`, the Quick Start's motor scenario in ladder and in Python. Open either to explore a complete project. They are copied only if not already present, so your edits to them survive an IDE update.

3.6 The active project

Only one Project is active at a time. The active project's node is marked *active* in the tree, its target decides which toolbar buttons appear, and Build, Connect and Emulate act on it. Click another project's node to switch. Editor tabs stay open across the switch; in a multi-project Application each tab title is prefixed with its project's name in square brackets. Switching projects drops any controller connection, since a connection belongs to the project it was made for.

3.7 Saving

- **Save (Ctrl+S)** writes the whole Application — every Project in it — to its `.widez` file. The button is enabled only when something has changed.
- **Save As (Ctrl+Shift+S)** writes a new archive. The copy is a *different* Application: every Project inside it is given a new identity, so a controller that holds the original will treat a deploy of the copy as a different project (see *Connecting, Deploying and Emulating* for what that means).
- There is no autosave. Closing the IDE, or opening another Application, with unsaved changes prompts you to save, discard or cancel. **Discard** cannot be undone.

Unsaved changes are shown by a leading `*` in the window title, a dot on the affected editor tab and *(modified)* in the status bar.

3.8 Removing a Project

Right-click the Project's node and choose **Delete Project**. The IDE asks you to confirm; the removal becomes permanent when you next save the Application.

3.9 Changing a Project's target

Open **Setup** → **Hardware Config** and pick a different **Select Target Hardware**. The tree and toolbar reshape at once, but nothing is converted: ladder programs are meaningless to a WhiskerHMI project and Python files to a ladder one. Change the target only on a project you have just created; otherwise add a new Project with the right target.

3.10 The Design Spec

Every Application and every Project has a **Design Spec** node. It opens a Markdown document — **Design Specification** — with **View** and **Edit** modes and sections for Overview, IO Configuration, Tag Database, Control Logic, HMI Screens, Alarms and Notes. It is the place for the narrative a `.widez` cannot otherwise carry: what the system does, the control strategy, commissioning notes.

Two parts of a Project's spec are maintained by the IDE inside marked blocks: the **Tag Database** tables (I/O tags, setpoints, internal and process variables) and the **Modbus Registers** tables of every exposed tag. They are refreshed each time the document is shown; anything you write outside the markers is left alone.

3.11 Your project on the controller

By default a deployed Nexus.io project carries its editable source with it (Project Properties → Deployment → **Store editable source on target**). When the IDE connects to a controller it compares the open project with the one the controller holds: if they differ, or if nothing is open, it offers **Download from target** so you can pull the project straight off the device — useful on a site visit with no copy to hand. Deploying over a controller that holds a *different* project asks for confirmation first. Turn the switch off for projects whose source must stay off the device. The whole sequence is in *Connecting, Deploying and Emulating*.

3.12 Project settings

Everything else that is set once per project — version, HMI PIN and operator login, history retention, cloud intervals, Modbus numbering, and the WhiskerHMI display, authentication and window layout — is on the Properties panel when the project node is selected. Every field is listed in *Project Properties Reference*.

3.13 Where the IDE keeps its own settings

The IDE stores small per-project preferences (column visibility, the last values used in export dialogs) in ****Documents_ide_prefs.json (Cloud edition) or Documents_ide_offline_prefs.json**** (Offline edition). You can delete the file to reset those preferences; it holds nothing you cannot recreate. There is no recent-files list.

4 Setting Up Your Nexus.io Automation Controller

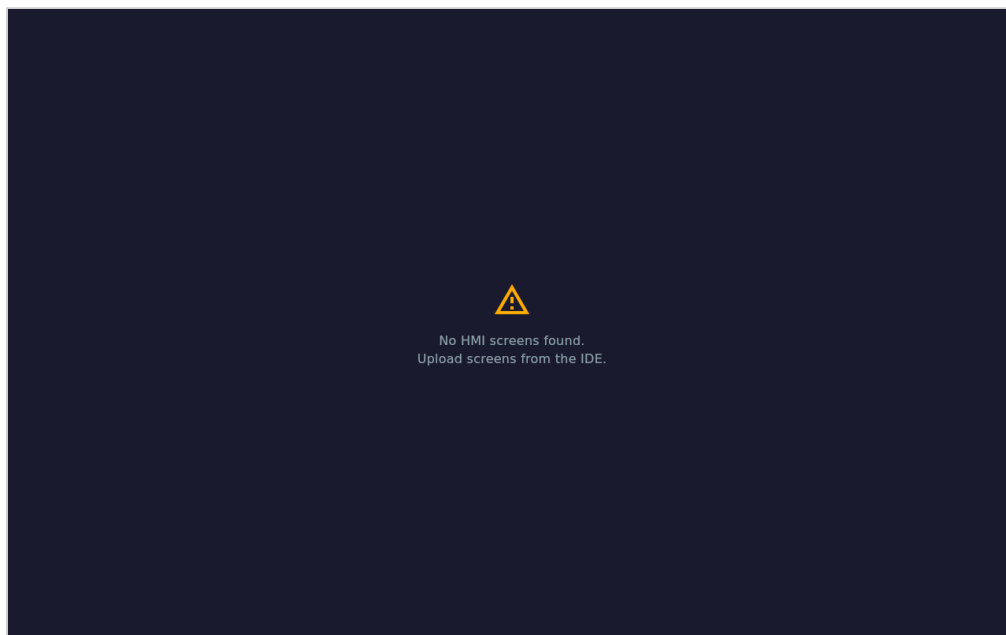
This chapter covers the controller side of getting started: what a new Nexus.io Automation Controller does when you first power it on, how to choose between the two ways of commissioning it, and what you can do on the touchscreen itself (network settings, touch calibration, the operator login). The IDE side — discovering the controller, connecting and deploying — is in *Connecting*, *Deploying and Emulating*, and the security side of commissioning is in *Security, Provisioning and Enrolment*.

It assumes you have mounted the controller and wired power, Ethernet and any RS-485 devices as described on the hardware quick-start sheet that ships with the unit.

4.1 Out of the box

A new Nexus.io controller ships **blank**. It has no program, no HMI screens and no security identity. When it boots, the touchscreen shows the HMI runtime with a single message:

No HMI screens found. Upload screens from the IDE.



On the network the controller announces itself so the IDE can find it. In the IDE's **Connect to Target** dialog a new controller is listed with the label **Standalone**. In this posture it accepts a plain connection from any IDE and an unsigned deploy, and a WhiskerHMI panel connects to it plainly. Nothing forces you to enrol it anywhere. Whether it stays that way is your choice; the next section explains the two ways to go.

4.2 Choose a commissioning path

A controller is **managed** when it is bound to a certificate authority that vouches for the IDEs allowed to program it: mutual TLS on the IDE and data ports, signed deploys only. A controller connected to the cloud is always managed — enrolment makes it so. A controller that is not connected to the cloud is managed only if you decide to make it so, and the IDE, the controller and a WhiskerHMI panel all respect that decision. Decide which applies to your site before you go further.

	Path A — Cloud	Path B — Standalone (no cloud account)
Needs	Cloud edition of the IDE, signed in to a Whisker.io account	Any edition
What you do	Cloud Explorer → Provision New Device ; the serial is filled in from the connected controller	Connect (by discovery or by address) and deploy. There is nothing to provision
Who vouches for your IDE	Your Whisker.io account's certificate authority. The IDE fetched your user certificate when you signed in	Nobody, unless you choose to run a site CA : one PC makes its keystore a site (Security → Site → make this keystore a site), issues identity packages to the engineers and provisions the unit with Security → Site → Provision unit (site)

	Path A — Cloud	Path B — Standalone (no cloud account)
Afterwards	The controller restarts managed : mutual TLS on the IDE and data ports, signed deploys only	The controller stays Standalone : any IDE that can reach it can deploy, unsigned. With the site CA it is managed, with certificates from the site instead of the cloud
Best for	Fleets, remote sites, anything you want visible in Whisker.io	Isolated control networks, single-engineer sites and sites with no internet. Add the site CA when several engineers share the plant or you want signed deploys and revocation. The customer owns the site CA

Both paths are walked through step by step in the Quick Start (*The Walkthrough scenario*), and the mechanics of enrolment, managed mode, the site CA and revocation are in *Security, Provisioning and Enrolment*. Nothing in the rest of this chapter depends on which path you choose.

Note. Once enrolled or provisioned, the controller no longer shows as *Standalone*; the Connect dialog labels it **Managed**. You may also see **Awaiting enrolment**: a locked-down configuration some sites order, in which the controller accepts nothing but the enrolment or site provisioning until it has been made managed. A controller you receive does not start in it unless it was ordered that way.

4.3 Powering on

Connect power. The backlight comes on, the operating system boots and the HMI runtime starts. Until a project has been deployed you see the “No HMI screens found” message above; after a deploy you see the first screen of your project.

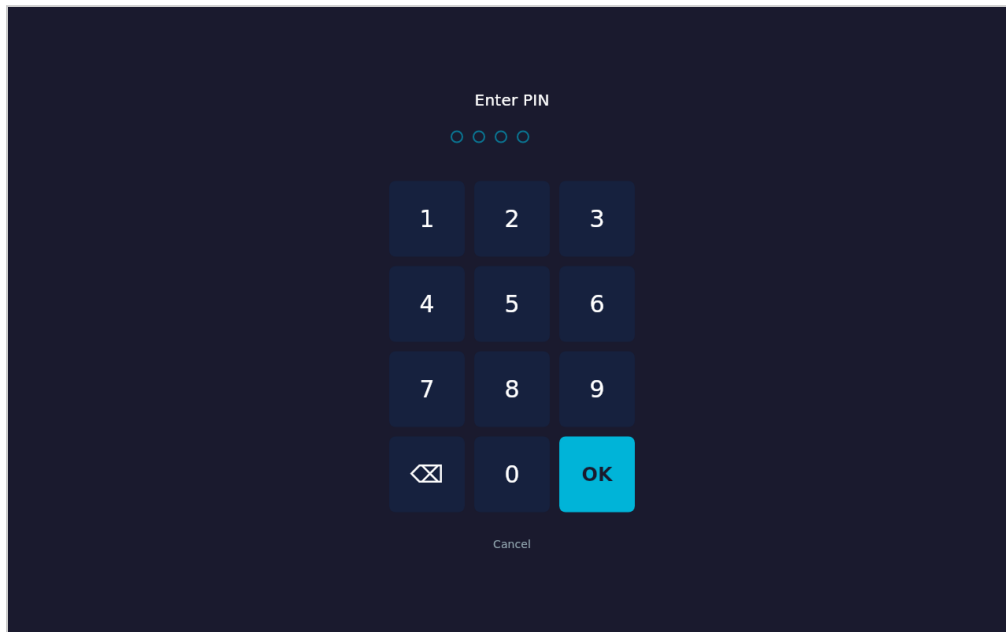
The runtime has a thin **status bar** across the top of the screen. It hides itself after a few seconds; **tap the top-right corner of the screen** to bring it back. The status bar shows:

- an **ONLINE / OFFLINE** indicator — whether the HMI is talking to the controller's own data service (not whether the cloud is reachable),
- on a controller with a cellular modem, the **cell signal**: four bars (green for three or four, amber for two, orange for one), the technology and the signal strength in dBm and as a percentage, for example `LTE -74 dBm 67%`. It reads **No cell** when the modem is not connected and `--` for a value the modem has not reported yet. The values come from the controller's cloud service and update every few seconds, so a controller running without the cloud (standalone) shows **No cell** even when it has a modem,
- the name of the logged-in operator and a **Lock** button, when operator login is enabled for the project,
- a **Settings** (gear) button.



4.4 The Settings screen

Tap the **Settings** gear in the status bar. A PIN prompt titled **Enter PIN** appears with a numeric keypad and a **Cancel** button.



This is the project's **Settings PIN** — the **PIN** field under *HMI Settings* in Project Properties (see *Project Properties Reference*). It is four to six digits and defaults to `0000`. Change it before the controller goes to a customer site so operators cannot alter the network settings. After three wrong entries the keypad locks for 30 seconds (**Too many attempts. Wait 30s**).

Note. The Settings PIN is separate from the operator login described later in this chapter. The Settings PIN protects the controller's own settings; the operator login controls who may press which buttons on your HMI screens.

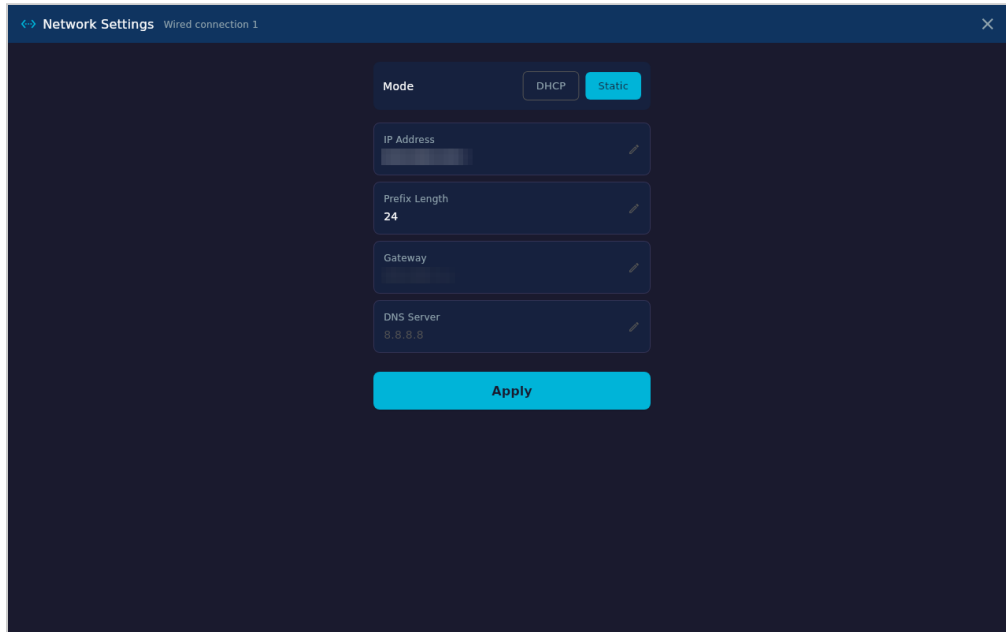
After the PIN, the **Settings** menu offers:

Entry	What it does
Network Settings	IP address, gateway, DNS — see below
Calibrate Touch	Aligns the touch panel to the display. Follow the on-screen targets
Audit Log	Who changed what on this panel, and the upload status of those records. Shown only when the project has operator login enabled

Tap the ✕ in the header to close Settings.

4.5 Network settings

Tap **Network Settings**. The form shows the name of the Ethernet connection, a **DHCP / Static** mode switch and four fields: **IP Address**, **Prefix Length**, **Gateway** and **DNS Server**.



Network Settings Wired connection 1

Mode: DHCP Static

IP Address

Prefix Length: 24

Gateway

DNS Server: 8.8.8.8

Apply

4.5.1 DHCP (the default)

Out of the box the controller asks your network's DHCP server for an address. In DHCP mode the **IP Address** field shows the address the controller currently holds — this is the quickest way to read the address off the panel when you need to type it into the IDE.

DHCP suits most plant LANs. Its drawbacks: the address can change when the lease is renewed, and an isolated control network may have no DHCP server at all.

4.5.2 Static address

For a fixed address:

1. Tap **Static**.
2. Tap each field in turn. An on-screen keypad opens; enter the value and confirm.
 - **IP Address** — the address you want the controller to use.
 - **Prefix Length** — the subnet size in prefix form: 24 is the same as a mask of 255.255.255.0.
 - **Gateway** — needed only if the controller must reach another subnet (the cloud, or a SCADA host elsewhere).
 - **DNS Server** — needed for the cloud path; otherwise optional.
3. Tap **Apply**. The panel confirms with **Static IP applied**, or **Switched to DHCP** if you went the other way. The change takes effect at once.

Warning. A static address outside your PC's subnet makes the controller unreachable from that PC. Pick an address in the same subnet, or move the PC. {.warning}

If the panel reports **No active ethernet connection**, the cable is not plugged in or the link is down; fix that before changing settings.

4.6 Finding the controller from the IDE

Once the controller has an address you rarely need the panel again. Click **Connect** in the IDE toolbar: the **Connect to Target** dialog scans the LAN and lists every controller it finds, with its address and its posture (*Standalone*, *Managed* or *Awaiting enrolment*). Click one and press **Connect**. A standalone controller accepts the connection and a deploy at once; a managed one needs the identity that enrolled or provisioned it; one awaiting enrolment accepts the connection so that it can be provisioned, but nothing can be deployed to it until it has been made managed.

Discovery uses mDNS, which does not cross a VPN or most routers. If your controller is not listed, click **Enter address manually**, type the address you read from the panel's Network Settings, keep the default port, and connect. Details are in *Connecting, Deploying and Emulating*.

4.7 RS-485 and the Modbus server

The controller has two RS-485 connectors, labelled **RS485A** and **RS485B**. In a project they are the ports `rs485` (RS485A) and `rs485_2` (RS485B). There is nothing to set up on the panel for either port. Their use — baud rate, the Modbus RTU devices on them and their register maps — is part of each project and configured in *I/O Configuration and Devices*. Each port supports 9600, 19200, 38400, 57600 and 115200 baud.

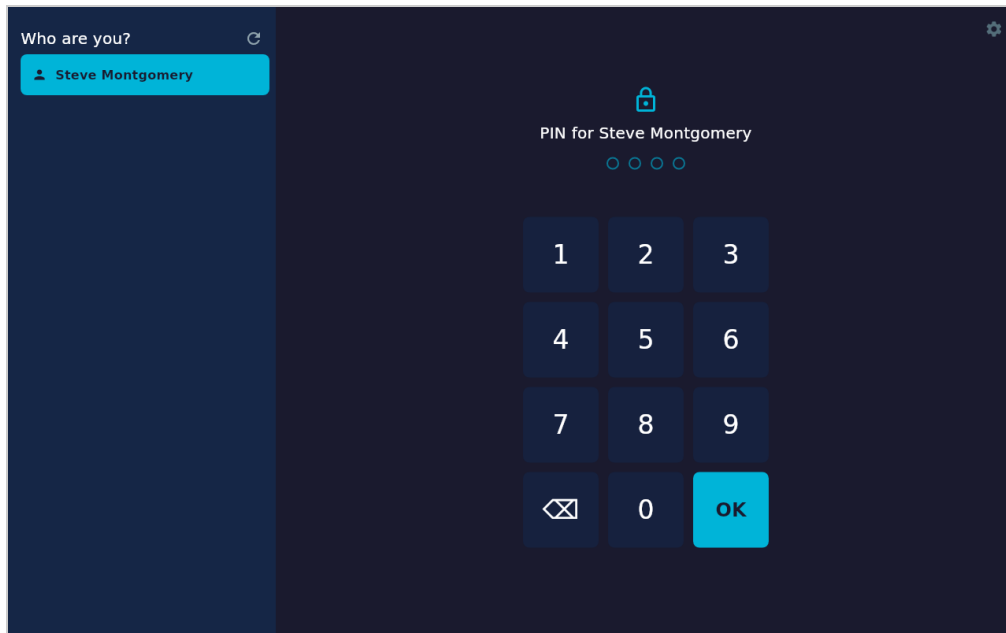
Warning. The **A** and **B** terminal labels on the RS-485 connectors are reversed. Wire the bus's **A (D+)** line to the terminal printed **B**, and **B (D-)** to the terminal printed **A**. Wired as printed, the bus idles in the wrong state: the controller polls, but no module ever answers, and every device on the port reports a timeout. With nothing transmitting, A must sit at least 200 mV above B. {.warning}

Likewise the controller's **Modbus TCP server** is always running on Ethernet; which tags it exposes, and at which addresses, is set per tag in the Tag Database (*Modbus Server and Map Viewer*).

4.8 Operator login at the panel

If the project has **Cloud PIN Login** turned on (Project Properties → HMI Settings), the panel starts locked and operators must identify themselves before they can act on a screen:

1. The lock screen lists the account's users under **Who are you?**. Tap your name.
2. The keypad heading changes to **PIN for *name***. Enter your Whisker.io PIN and tap **OK**.



Users and PINs come from your Whisker.io account; nothing is stored in the project. A wrong PIN shows how many tries remain; after too many the cloud locks that PIN for a period and tells you so (**This PIN is locked for ... minute(s). Other users can still log in.**). Every attempt is audited.

While logged in, the operator's name appears in the status bar next to a **Lock** button. Tapping it — or a **lockButton** widget you placed on a screen — ends the session. The session also ends after the project's **Session Timeout** of no touches, and whenever the screen goes to sleep.

What each role may do (buttons, setpoints, alarm acknowledgement and so on) is set in Project Properties under **Roles at the panel**, with per-widget overrides in the HMI designer. An operator who tries an action their role does not allow sees a short **Requires role** notice; with nobody logged in the notice is **Log in first**.

4.8.1 When the cloud is unreachable

The lock screen shows **Offline — recent users only**: operators who have logged in on this panel recently can still log in with their PIN. If the project's **Offline Fallback** switch is on, a long press on the lock screen title switches the keypad to **Maintenance PIN**, which accepts the project's Settings PIN and logs in as a local maintenance user. Three wrong maintenance PINs lock the keypad for 30 seconds. Leave Offline Fallback off unless your site needs it; on is less secure.

4.9 Screen sleep

After five minutes without a touch the backlight turns off. Touch the screen anywhere to wake it. If operator login is in use, waking brings you back to the lock screen.

4.10 Before you open the IDE

Check the basics once:

- The panel shows either your project's first screen or the "No HMI screens found" message — either means the runtime is up.
- Network Settings shows a non-zero **IP Address**.
- Your PC can ping that address.

If the ping fails, fix the network (subnet, cable, switch port, firewall) before trying the IDE — the IDE cannot reach a controller that ping cannot. When it succeeds, continue with *Connecting, Deploying and Emulating*, or with *Security, Provisioning and Enrolment* if you are taking the cloud path or setting up a site CA.

5 Target Hardware in the IDE

A **Target** is the kind of hardware — or, for WhiskerHMI, the kind of PC application — a Project runs on. This chapter describes the three targets the IDE supports, what each one gives you, and how the choice reshapes the IDE.

5.1 How the IDE knows a target

Each target is described to the IDE by a **hardware definition**: the memory areas (**arenas**) the controller has and their sizes, the ports it offers and the protocols each can carry, its display, which editors apply (ladder, HMI, Python) and how a program reaches it. When you pick a target the IDE reads that definition and shows only the tree nodes, editors and toolbar buttons that make sense for it. You never edit hardware definitions yourself.

The arena names appear in tag addresses and in Tag Monitor:

Arena	Holds	Address prefix
DI	Digital inputs	%I
DQ	Digital outputs	%Q
AI	Analog inputs	%IW
AQ	Analog outputs	%QW
MI	Memory integers	%MW
MR	Memory reals	%MF

5.2 Nexus.io Automation Controller (`Nexus.io AC`)

The Nexus.io Automation Controller is a panel-mounted PLC with an integrated 10-inch touchscreen. Part number **NIO-AC-11-D-120-0000-112N-N-6**.

Display	10-inch touchscreen, 1280 × 800; runs the HMI screens you design
Ethernet	Modbus TCP, HTTP and MQTT (cloud)
RS-485	Two ports, RS485A and RS485B (<code>rs485</code> and <code>rs485_2</code> in a project); Modbus RTU at 9600, 19200, 38400, 57600 or 115200 baud

Cellular	Optional, for cloud connectivity without a site network
Arenas	DI 256, DQ 256, AI 128, AQ 128, MI 512, MR 512
Program memory	64 MB

What a Nexus.io project can do:

- **Ladder logic** in as many tasks and function blocks as you need (*Ladder Logic Editor, Function Blocks, Faceplates and Tasks*).
- **Python** — an `app.py` with a `run(ctx)` entry point, run as a freewheel task alongside the ladder, and PYFB blocks that call Python from ladder (*Python on the Controller*).
- **Devices**: Modbus RTU modules on either RS-485 port, Modbus TCP devices, **ArenaTCP** (read another Nexus.io controller's tags, optionally over a NexusSec secure connection) and **EtherNet/IP** PLCs (*I/O Configuration and Devices*).
- **Modbus TCP server**, always running on port 502, exposing the tags you mark **Expose** (*Modbus Server and Map Viewer*).
- **CIP tag server**, off until the project turns it on, serving the tags you publish by name over EtherNet/IP on port 44818 to a Logix-style SCADA driver (*CIP Tag Server*).
- **HMI** on the touchscreen, including operator PIN login (*HMI Designer*).
- **Alerts** with latching and an alert-table widget (*Alerts Editor*).
- **Historian** — on-controller recording of selected tags for the trend chart widget and WhiskerHMI trends (*Historian and Trends*).
- **Cloud**: process variables published to Whisker.io, remote writes, and managed security with certificates from your account or your site CA (*Security, Provisioning and Enrolment*).

Editors in the tree: Design Spec, Modbus Map, Setup (Hardware Config, IO Config), Tags, Tasks, Functions, HMI Screens, Alerts, Python.

5.3 SmartController (Classic) (`SmartController Classic`)

D6 Labs' earlier controller, built on the Raspberry Pi RP2350 (Pico 2) module and programmed in **MicroPython** over a USB serial connection. It is kept in the IDE so that installed units can be maintained from the same tool.

Programming	MicroPython files edited in the Python editor and synced to the device
-------------	--

Connection	USB serial (Connect to SmartController Classic dialog: pick the port and a baud rate of 9600, 19200, 38400 or 115200)
RS-485	Modbus RTU master or slave, configured under Device Settings in Project Properties
Display	None

Editors and panels: the project's file tree (**New File, New Folder, Sync to Target**), Design Spec, Tag Database, the Python editor and the **SmartController Classic Console** (interactive REPL). There is no ladder editor, no HMI Designer and no I/O Configuration; I/O is addressed in Python. Build writes the application and configuration packages the factory load tool installs (`dist_app/` , `dist_config/`) and needs no connection; see *Python on the Controller*.

5.4 WhiskerHMI (`WhiskerHMI`)

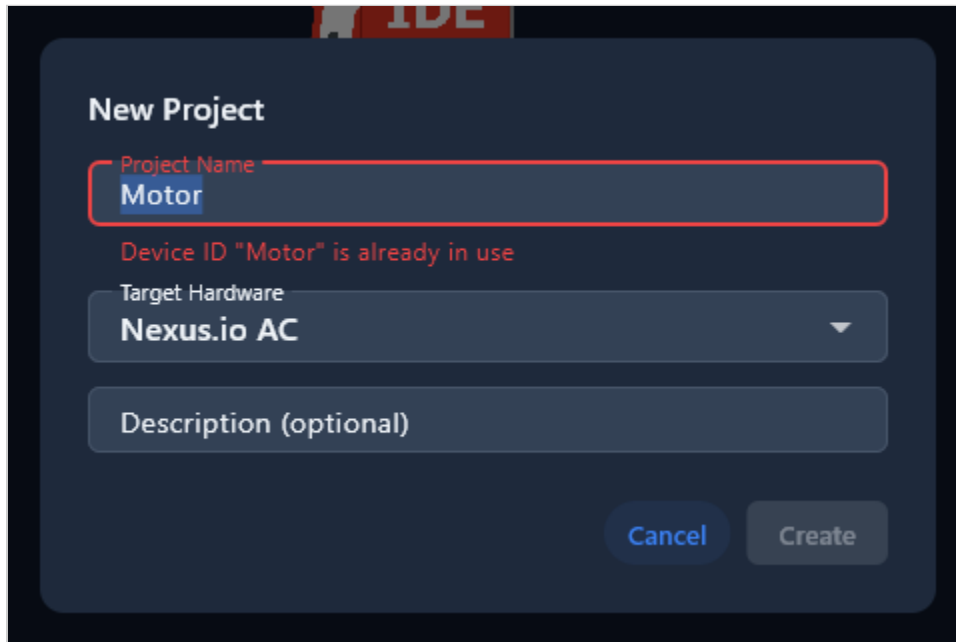
Not hardware: a **Windows application** the IDE generates from your screens. It connects to a Nexus.io controller over the network through an **ArenaTCP** device in its I/O configuration, shows the same widgets the controller's touchscreen can show, and adds PC conveniences — multiple windows across several monitors, a Trends dialog over the controller's historian, CSV export, operator login with Whisker.io users and roles, and a local audit log.

Runs on	Any Windows PC; installed from an installer the IDE generates
Design resolution	1920 × 1080 by default; set per project under Display
Arenas	DI 4096, DQ 4096, AI 2048, AQ 2048, MI 4096, MR 4096 — local tags plus the tags imported from the controller
Connection to the controller	ArenaTCP; the controller's address is a property of that device

Editors in the tree: Design Spec, Setup (Hardware Config, IO Config — where the ArenaTCP device and its **Import Tags** live), Tags, HMI Screens, Alerts. The toolbar offers **Build**, **Run** (start the application on this PC) and **Generate Installer**. See *The WhiskerHMI PC Application* and *Generating HMI Installers*.

5.5 Choosing a target

The **Target Hardware** dropdown appears whenever you add a project — in the **New Application** dialog and in **New Project**. Pick the one matching the hardware the project will deploy to.



The target can be changed afterwards in **Setup** → **Hardware Config** → **Select Target Hardware**, but nothing is converted when you do: ladder programs mean nothing to a WhiskerHMI project, and Python files nothing to a ladder one. Change it only on a project you have just created.

5.6 What changes with the target

	Nexus.io controller	SmartController (Classic)	WhiskerHMI
Tree nodes	Design Spec, Modbus Map, Setup, Tags, Tasks, Functions, HMI Screens, Alerts, Python	Files, Design Spec, Tags	Design Spec, Setup, Tags, HMI Screens, Alerts

	Nexus.io controller	SmartController (Classic)	WhiskerHMI
Toolbar, after the file buttons	Build (F5), Build & Emulate, Connect, Run Mode, Stop Mode, Debug Mode, Enable Monitoring	Build, Connect	Build, Run, Generate Installer
New Task / New Function buttons	yes	—	—
Build needs a connection	Yes — Build compiles and downloads to the connected controller	No	No
Deploy path	Over the network through the IDE port, as a signed bundle on a managed controller	USB serial (Sync to Target)	Run locally, or install the generated installer on the target PC
Bottom-panel tabs you will use	Output, Tag Monitor, Statistics, IO Stimulus	Output, SmartController Classic Console	Output, HMI Preview

In the Cloud edition, **Publish to Cloud** and **Security** appear for every target. The Tag Database, Alerts editor and HMI Designer work the same way wherever they appear; the parts a target does not support simply do not show.

6 Tag Database

The Tag Database is where every named variable in a project lives. Tags are the bridge between hardware (I/O channels, Modbus registers, peer controllers) and logic (ladder networks, Python, alerts, HMI bindings). This chapter is the reference for the Tag Database editor on a Nexus.io Automation Controller project. Tags on a WhiskerHMI project work the same way; a SmartController (Classic) project uses a reduced tag table described in *Python on the Controller*.

6.1 What a tag is

Every tag has a **Name**, a **Type**, a **Class**, an **Address** and, optionally, an initial value and a description. Names start with a letter or underscore and continue with letters, digits and underscores; they are case-sensitive. Rename a tag and every reference to it in ladder, HMI, alerts and Python is updated with it.

6.1.1 Data types

Type	Size	Range	Typical use
BOOL	1 bit	TRUE / FALSE	switches, lamps, alarm bits, run commands
INT	16 bits, signed	-32,768 to 32,767	counts, mode words, raw Modbus registers
DINT	32 bits, signed	±2,147,483,647	runtime seconds, totals, anything that could overflow INT
REAL	32 bits, floating point	about 7 significant digits	levels, pressures, flows, setpoints, gains

Projects saved by older versions of the IDE, in which INT was 32 bits, are converted when opened: their INT tags become DINT and their addresses move to the 32-bit area, so the program behaves as before.

6.1.2 Classes

Class	Meaning
VAR	An ordinary variable. Its initial value is applied every time a program is downloaded.
VAR_RETAIN	A retained variable. The controller keeps its last value across power cycles and program updates (see <i>Retained variables</i> below).
CONST	A constant. Give it an initial value and use it wherever a fixed number is needed; the program cannot overwrite it.

6.1.3 Addresses

An address names the memory area and the slot a tag occupies. The IDE assigns addresses; you choose the area and the next free slot is filled in for you.

Prefix	Area	Holds	Assigned by
%I	Digital Input	BOOL	I/O Configuration, one per device channel
%Q	Digital Output	BOOL	I/O Configuration
%IW	Analog Input	INT / DINT	I/O Configuration
%QW	Analog Output	INT / DINT	I/O Configuration
%MX	Memory Bit	BOOL	Tag Database
%MW	Memory Word (16-bit)	INT	Tag Database
%MD	Memory DWord (32-bit)	DINT	Tag Database
%MF	Retain Float (32-bit)	REAL	Tag Database

Memory bits are written as byte and bit: `%MX0.7` is bit 7 of byte 0, and the IDE allocates them in that form. The plain form `%MX7` names the same bit; both spellings are accepted everywhere. Word areas are indexed one slot per tag: `%MW0` , `%MW1` , `%MD3` , `%MF12` . The areas are independent, so `%MX0.0` , `%MW0` ,

`%MD0` and `%MF0` are four different locations. The IDE allocates up to 2,048 memory bits and 1,024 slots in each word area — far more than a typical project uses.

You cannot type a `%I` or `%Q` address into the Tag Database. Those tags exist only because a device channel exists: add the device in I/O Configuration, the IDE creates one tag per enabled channel, and you rename them (see *I/O Configuration and Devices*).

6.2 Opening the editor

In the Project Tree expand **Tags** and click **Tag Database**. The editor opens as a tab.



Name	Type	Class	Initial	Address	Source	PV	P...	Modbus	Expose	Hist	Description
active_level	REAL	VAR		%MR22		31	2000 / 42001				Active sensor level (%)
<i>sys_active_level Age</i>	DINT	VAR		%MD46							Seconds since this P...
ai_pressure_raw	INT	VAR	3000	%MW17							Pressure transmitter ...
ai_radar_raw	INT	VAR	3000	%MW18							Radar transmitter ra...
alarm_high_level	BOOL	VAR		%MX61		38	0 / 40001 coil 0 / 00001				High level alarm latc...
<i>sys_alarm_high_level Age</i>	DINT	VAR		%MD53							Seconds since this P...
alarm_io_comm	BOOL	VAR		%MX66		43	1 / 40002 coil 1 / 00002				IO comm fault latched
<i>sys_alarm_io_comm Age</i>	DINT	VAR		%MD58							Seconds since this P...
alarm_p1_fts	BOOL	VAR		%MX57		34	2 / 40003 coil 2 / 00003				P1 fail-to-start latched
<i>sys_alarm_p1_fts Age</i>	DINT	VAR		%MD49							Seconds since this P...
alarm_p1_seal	BOOL	VAR		%MX59		36	3 / 40004 coil 3 / 00004				P1 seal failure latched
<i>sys_alarm_p1_seal Age</i>	DINT	VAR		%MD51							Seconds since this P...

6.2.1 Columns

Column	What it shows
Name	The tag name. System tags carry a SYS badge and are shown in italics.
Type	BOOL, INT, DINT or REAL.
Class	VAR, VAR_RETAIN or CONST.
Initial	The initial value, if one is set (TRUE/FALSE for BOOL).
Address	The assigned address.

Column	What it shows
Source	For tags imported from another controller over ArenaTCP, the name of the source device; blank for local tags.
PV	A tick when the tag is a process variable published to the cloud (Cloud edition only).
PV#	The process-variable slot, 0–63 (Cloud edition only).
Modbus	The holding-register address on the controller's Modbus TCP server, shown as the raw address and the 4xxxxx form; BOOL tags also show a coil address. A dash means the tag is not exposed.
Expose	Whether the tag is exposed on the Modbus TCP server.
CIP	Shown only when the project's CIP tag server is on: a network icon, filled when the tag is published by name to EtherNet/IP clients. Click it to publish or withdraw the tag; hover to see whether it is read-only or writeable. See <i>CIP Tag Server</i> .
Hist	Whether the tag is recorded in the controller's historian. Click the icon to toggle it.
Description	Free text.

Click a column header to sort by it; click again to reverse the order. The search box at the top left, **Search tags...**, filters rows by name.

6.2.2 Toolbar

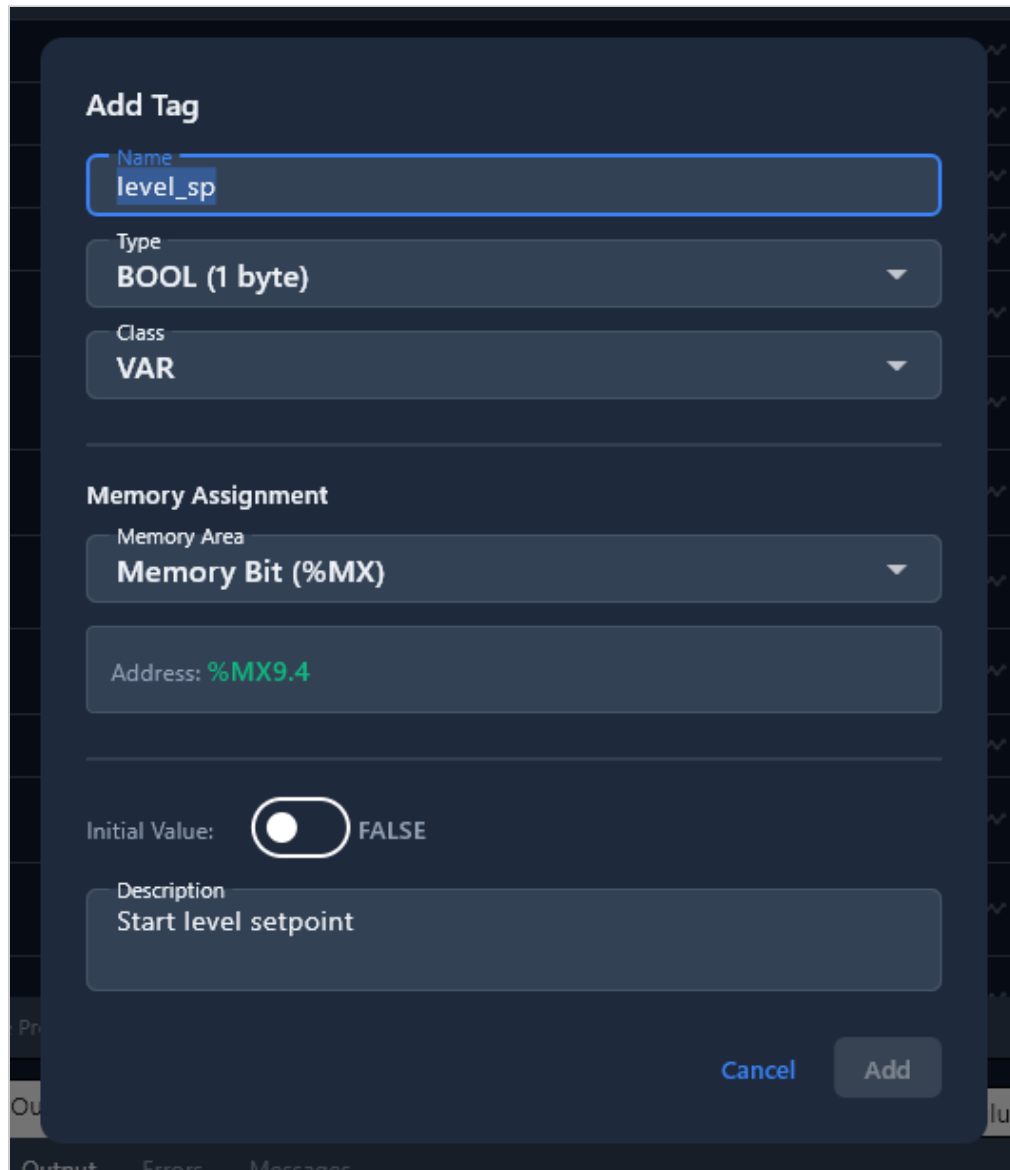
- **Add Tag** — opens the Add Tag dialog.
- **Import Tags...** / **Export Tags...** — read or write the tag list as a CSV file; see *Importing and exporting tags* below.
- **Remove Unused** — finds tags that no ladder network, Python file, HMI binding, alert or I/O point references and offers to delete them. The dialog lists them and has a **Skip IO-bound tags** checkbox so device channel tags are kept. System tags are never removed.

- **Delete Selected (n)** — appears when one or more rows are ticked with the checkbox at the left of each row. The confirmation lists every tag that will go and warns that references to them will break until you update them.

Each row also has a **Delete tag** icon at its right edge.

6.3 Creating a tag

Click **Add Tag**.



Add Tag

Name
level_sp

Type
BOOL (1 byte)

Class
VAR

Memory Assignment

Memory Area
Memory Bit (%MX)

Address: %MX9.4

Initial Value: ☐ FALSE

Description
Start level setpoint

Cancel Add

1. Type the **Name** — letters, digits and underscore, starting with a letter or underscore. The dialog checks as you type and says why a name cannot be used: it already exists, it differs from an existing tag only by case (the

controller treats `Pump1` and `pump1` as one tag), or it is a system tag's name. **Add** stays disabled until the name is acceptable. The same rule applies wherever a tag is created or renamed: the Properties panel, the undefined-tag sweep and the CSV import.

2. Pick the **Type**. The dialog lists each type with its size in bytes.
3. Pick the **Class** — VAR, VAR_RETAIN or CONST.
4. Under **Memory Assignment**, choose the **Memory Area**. The IDE suggests the natural area for the type (BOOL to `%MX`, INT to `%MW`, DINT to `%MD`, REAL to `%MF`) and shows the address it will assign as **Address:**. You do not type an address.
5. Set an initial value if the tag needs one, and a **Description**.
6. Click **Add**.

Tags are also created for you in three other places: I/O Configuration creates one tag per device channel; the Properties panel offers to create any tag name you type that does not exist yet; and the undefined-tag sweep creates tags in bulk before a build. All three are described below.

6.4 Importing and exporting tags

A tag list usually starts life in a spreadsheet, an instrument index or another controller's export. **Export Tags...** writes every tag except the system tags to a CSV file (UTF-8, comma separated, one header row) that Excel opens directly, and **Import Tags...** reads the same layout back in. The columns, in the order the export writes them:

Column	Holds	On import
Name	The tag name	Required. Letters, digits and underscore; must not be a system tag
Type	BOOL, INT, DINT or REAL	Required
Class	VAR, VAR_RETAIN or CONST	Blank = VAR
Initial	Initial value (TRUE/FALSE for BOOL)	Optional

Column	Holds	On import
Address	The assigned address	Blank = the next free address for the type, as Add Tag would assign it. A given address must be in the right area for the type and not already used by another tag
Description	Free text	Optional
Units	Engineering units	Optional
Min, Max, Decimals	Display range and decimal places	Optional
PV	Process-variable slot 0–63, or Y for the next free slot	Optional (Cloud edition)
ModbusExpose	Y when the tag is exposed on the Modbus TCP server	Optional
History	Y when the tag is recorded in the historian	Optional

Only **Name** and **Type** are needed; column order does not matter, because the header names the columns, and any column the IDE does not recognise is ignored — a spreadsheet with extra columns imports as it is.

Import shows a preview before anything changes: how many rows are **new tags**, how many **match an existing tag**, and every **refused row** with its line number and reason (an unknown type, an address in the wrong area or already taken, a name used twice in the file, a system tag name). For the rows that match an existing tag you choose once for the whole file: **skip them**, or **update them from the file**, which changes only the columns the file carries and leaves a blank cell's value as it is. Nothing is written until you click **Import**; the result line in the editor reports what was added, updated, refused and skipped. The cloud dashboard fields (gain, offset, display condition, select values) are not in the file and are never touched by an import.

6.5 Editing a tag

Select a row. Its parameters appear in the **Properties** panel on the right, which is where all editing happens: name, type, class, initial value, description, **Expose via Modbus** and the Modbus address, the **CIP tag server** publish and writeable switches, process-variable settings, and the history flag. A rename that collides with an existing name is refused and the field reverts.

Some tags have a fixed type and the type field is disabled with the reason shown beneath it: device channel tags (“type comes from the device definition”), tags imported from another controller, and system tags.

Tip. Select several rows with the checkboxes to delete them together; everything else is edited one tag at a time in the Properties panel. {tip}

6.6 Tags created from the Properties panel

Whenever you type a tag name into a Properties-panel field — a contact, a coil, a timer preset, a PID setpoint, an HMI binding — the IDE checks it against the Tag Database as soon as the field loses focus. A name that does not exist brings up the **Create New Tag** dialog, pre-filled with the name, a **Data Type** suggested from the pin you typed it into (BOOL for a contact or a Q output, REAL for a PID setpoint, DINT for other value pins), a **Variable Class** of VAR, the matching **Memory Area** and the address that will be assigned. Confirm the type — a wrong type compiles and runs but produces wrong numbers — and click **Create**. While you type, the field shows *New tag — you will be asked to create it*; a number shows *Numeric literal (constant value)* instead, because a number typed into a value pin is a constant, not a tag.

pressed

Create New Tag

Tag "fill_delay" does not exist. Create it now?

Name
fill_delay

Data Type
DINT (4 bytes) ▼

Variable Class
VAR ▼

Memory Assignment

Memory Area
Memory DWord (32-bit) (%MD) ▼

Assigned Address: %MD6

Initial Value
0

Description

Cancel Create

6.7 The undefined-tag sweep

Names can still reach the ladder without a tag behind them — from an older project, or because you cancelled the Create New Tag dialog. Before every **Build (F5)** and **Build & Emulate** the IDE sweeps the project and, if it finds any, shows one dialog, **N tags are used but not defined**, with a row per name: a checkbox, the name, a **Type** dropdown pre-filled from the pin, and where the name is used. **Create All** creates them and the build continues; **Cancel** stops the build. The

same sweep runs on **Save** and **Save As**, but there it never blocks: a **Save Anyway** button is always available, because saving is the operation that protects your work.

 **3 tags are used but not defined**

The ladder references these names, but the tag database has no entry for them. They must be defined before emulating.

The type is suggested from the pin each tag is wired to — confirm it, since a wrong type compiles and runs but produces wrong numbers.

TAG	TYPE	USED AT
☒ clear_alarms	BOOL	Main / rung 5 / contact (+2 more)
☒ float_start	BOOL	Main / rung 7 / contact
☒ float_stop	BOOL	Main / rung 8 / contact

Cancel
Create All

6.8 System tags

The IDE creates some tags itself and keeps them in step with what owns them. They carry the SYS badge, are read-only, cannot be deleted, and disappear when their owner is removed.

Owner	Tags	Meaning
Every I/O device	<device> Comm OK (BOOL), <device> Comm Age (DINT), <device> Error Type (DINT)	Device reachable and polling; seconds since the last good poll; last error (0 none, 1 timeout, 2 connection lost, 3 protocol, 4 device error, 5 configuration)
Every Nexus.io controller project	Cell Connected , Cloud Connected (BOOL); Cell RSSI , Cell Quality , Cloud Age (DINT)	Cellular modem attached; cloud session up; signal strength (dBm) and quality (%); seconds since the last cloud publish

Owner	Tags	Meaning
A project that requires a cloud PIN at the panel	HMI User Id (DINT), HMI Session Active (BOOL)	The Whisker.io user logged in at the panel (0 = locked); a session is open
Every process variable	<tag> Age (DINT)	Seconds since the PV was last written

Use them like any other tag: a contact on `uio Comm OK` makes a communications-loss alarm, and a numeric display bound to `Cell RSSI` shows signal strength on the panel.

6.9 Process variables

In the Cloud edition a tag can be published to your Whisker.io account as a **process variable**. Tick the PV setting in the Properties panel; the IDE assigns the next free slot (0–63) and shows it in the **PV#** column. The controller publishes the value and the cloud can write it back if the tag is marked writeable. A project can have 64 process variables.

6.10 Modbus addresses and renumbering

The controller always runs a Modbus TCP server, and every tag with **Expose via Modbus** ticked gets a holding-register address (and, for BOOL tags, a coil address). Addresses are assigned automatically when you tick the box, using the project's numbering mode; you can also type an address by hand, which marks it as manually assigned.

The numbering mode lives in the Properties panel when the project node is selected, under **Modbus Server**:

- **Auto-numbering mode — Alphabetical** (sorted by name, packed from 0), **Group by data type** (the default: each type has its own address range, editable in the table that appears), or **Creation order**.
- **Renumber all auto-assigned tags** — reassigns every exposed tag's address using the current mode. The confirmation, *Renumber Modbus addresses?*, warns that this changes the address every SCADA system reads and has a **Keep manually-assigned addresses** checkbox, ticked by default. Changing the mode itself triggers the same renumber and the same warning.

The generated map — every exposed tag with its register — is the **Modbus Map** node, which also exports CSV for SCADA import. See *Modbus Server and Map Viewer*.

6.11 History

Tick a tag's **Hist** icon (or the history setting in the Properties panel) and the controller records it in its on-board historian for the project's History Retention period. Tags used by a trend chart widget are recorded whether or not Hist is ticked. See *Historian and Trends*.

6.12 Retained variables

A tag of class **VAR_RETAIN** keeps its value across a power cycle and across program updates. When you download a new program the controller carries retained values over by name: a setpoint an operator entered at the panel last month is still there after today's deploy, even if you added or removed other tags and the layout moved. A retained tag's initial value is applied only when the tag is new or the controller starts cold with no saved snapshot.

A plain **VAR** works the other way round: its initial value is applied on every download, so a gain or a mode word you set in the Tag Database is always what the new program starts with. Use VAR for values the program owns and VAR_RETAIN for values operators own.

Function blocks that accumulate — the retentive timer, the totalizer, the motor's start counter and runtime meter, the drum's step number — keep their running value in a tag you nominate rather than inside the block. Declare that tag VAR_RETAIN and the total survives a power cycle; leave it VAR and it starts at zero each boot. See *Ladder Instruction Reference*.

Note. Retained values are stored on the controller, not in the project. A fresh controller starts from the initial values in the Tag Database.

6.13 Tag Cross Reference

In the Project Tree, under **Tags**, click **Tag Cross Reference**. The table lists every tag with its type and a chip for each place it is used — ladder networks (program and network number), HMI elements (screen and element type), Python files (file and line) and alerts. Double-click a chip to jump there: the ladder editor opens

scrolled to the network, the HMI Designer selects the element, the Python editor opens at the line. Tags nobody uses are marked *Not used*, which is the same information **Remove Unused** acts on.

Tag Name	Type	Usage (double-click to navigate)
Module1_DO7	BOOL	Not used
Module1_DO8	BOOL	Not used
Module2_DO6	BOOL	Not used
Module2_DO7	BOOL	Not used
Module2_DO8	BOOL	Not used
Module3_DO5	BOOL	Not used
Module3_DO6	BOOL	Not used
Module3_DO7	BOOL	Not used
Module3_DO8	BOOL	Not used
Module4_DO5	BOOL	Not used
Module4_DO6	BOOL	Not used
Module4_DO7	BOOL	Not used
Module4_DO8	BOOL	Not used
Module1_AI3	INT	Not used
Module1_AI4	INT	Not used

6.14 Naming conventions

The IDE does not enforce a naming style, but consistent names pay off in the cross reference and in every tag picker:

- `pb_` for pushbuttons, `sw_` for selector switches, `lamp_` for indicators
- `_cmd` for commands written by the HMI, `_sts` for status written by the ladder
- engineering units as a suffix on analog tags: `tank_level_pct`, `flow_gpm`, `temp_degf`
- `sp_` for setpoints, which are usually `VAR_RETAIN`

7 I/O Configuration and Devices

The **IO Config** editor is where you declare every device the project talks to: Modbus I/O modules on RS-485 or Ethernet, another Nexus.io Automation Controller read over ArenaTCP, or an EtherNet/IP PLC read by tag name. Each device brings its own tags into the Tag Database, and the controller's I/O scanner keeps those tags in step with the hardware every scan. This chapter is the reference for the editor, the Add I/O Device dialog and the device-specific dialogs.

7.1 Where I/O comes from

The Nexus.io controller has no field terminals of its own. Every physical input and output lives on an external module: a Modbus RTU device on the controller's RS-485 ports, or a Modbus TCP device on the Ethernet port. The IDE describes each module type with an **IODF** (I/O Device File), and adding a module is a matter of picking the IODF, naming the instance and entering its address.

Two more device kinds carry tags rather than physical points:

- **Arena TCP** — reads and writes the tags of another Nexus.io controller (or another WhiskerHMI project's source controller) through its arena agent. This is how a WhiskerHMI PC project gets its data, and how one controller can watch another.
- **EtherNet/IP PLC** — reads and writes named tags in a Logix-family PLC. Tags are entered by name after the device is created.

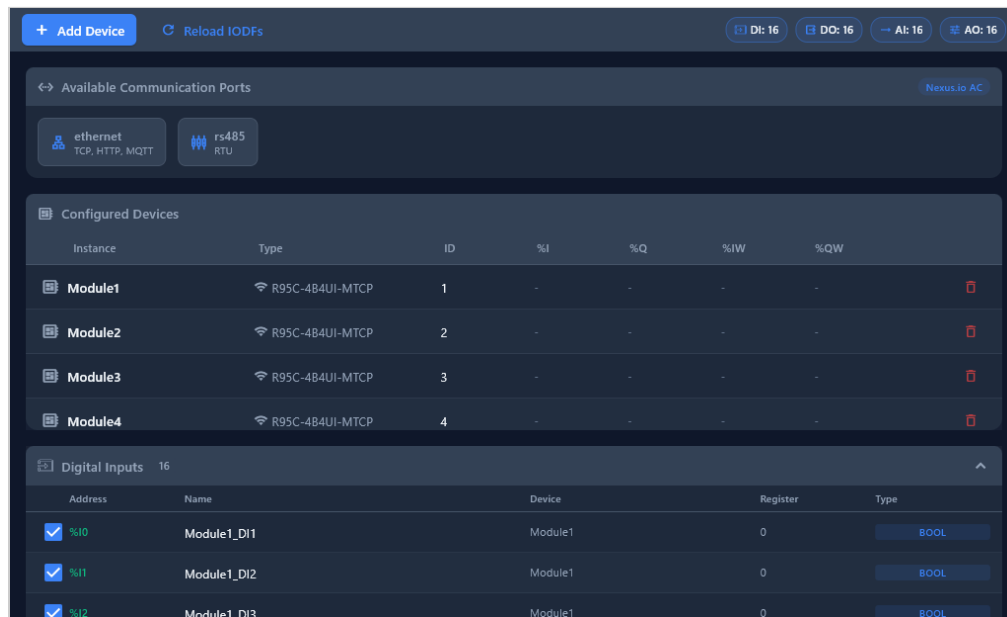
A WhiskerHMI project has an IO Config node too; it normally holds a single Arena TCP device pointing at the controller. A SmartController (Classic) project has no IO Config node.

7.2 Opening the editor

In the Project Tree expand the project, then **Setup** → **IO Config**. The editor opens as a tab with a toolbar and a stack of cards:

Card	Contents
Available Communication Ports	One chip per port on the target hardware: <code>ethernet</code> , <code>rs485</code> (the connector labelled RS485A) and <code>rs485_2</code> (RS485B) on the Nexus.io controller. Click a chip to edit its settings in the Properties panel. A WhiskerHMI project shows no ports.
Configured Devices	Every device declared in the project, with its start addresses and row actions.
Digital Inputs, Digital Outputs, Analog Inputs, Analog Outputs	The I/O points contributed by the configured devices, one collapsible table per class.

The toolbar has **Add Device**, **Reload IODFs** (re-reads the IODF folders after you drop a new file in) and four counters on the right: `DI` , `DO` , `AI` , `AO` .



The screenshot shows the D6 Labs I/O Configuration interface. At the top, there is a toolbar with a blue '+ Add Device' button, a 'Reload IODFs' button, and four status indicators: 'DI: 16', 'DO: 16', 'AI: 16', and 'AO: 16'. Below the toolbar, the interface is divided into three main sections:

- Available Communication Ports:** This section shows two chips: 'ethernet' (TCP, HTTP, MQTT) and 'rs485' (RTU). The 'rs485' chip is selected.
- Configured Devices:** This section displays a table of configured devices. The table has columns for Instance, Type, ID, %I, %Q, %IW, and %QW. There are four devices listed, all of type 'R95C-4B4UI-MTCP'.
- Digital Inputs:** This section shows a table of digital inputs. The table has columns for Address, Name, Device, Register, and Type. There are three inputs listed, all of type 'BOOL'.

7.2.1 Serial port settings

Modbus RTU devices on one RS-485 port share it, so its settings live on the port rather than on each device. The two ports are independent: put modules that run at different speeds on different ports. Click the `rs485` or `rs485_2` chip under **Available Communication Ports**; the Properties panel shows **Baud Rate** (the rates the hardware supports), **Parity** (None, Even, Odd) and **Stop Bits** (1 or 2). Every RTU device on that bus must be set to the same values on the device itself.

Which physical connector a port drives is fixed by the target hardware, not by the project. A project saved by an earlier version of the IDE, which knew only one port and placed it on the wrong connector, is corrected when you next build it; you do not need to edit it. If the modules on a port never answer, check the wiring first (see *Setting Up Your Nexus.io Automation Controller*: the A and B terminal labels are reversed).

The ethernet chip is read-only here; the controller's own IP address, subnet and gateway are set in the Network Configuration card of Hardware Config.

7.3 Adding a device

Click **Add Device**. The **Add I/O Device** dialog opens with a **Device Kind** choice at the top:

Device Kind	Use it for
Modbus / EtherNet-IP device (from IODF)	A physical I/O module or third-party device with a known register map
Arena TCP (another controller)	Another Nexus.io controller or its project; tags are imported from the source project
EtherNet/IP PLC (Allen-Bradley, by tag name)	A Logix-family PLC; tags are added by name after the device exists

Fill in the fields for the chosen kind and click **Add Device**. The confirm button stays disabled until the required fields are complete, and the dialog refuses an **Instance Name** that is already in use. The instance name is the prefix of every tag the device creates, so pick it with care (for example `FieldIO` rather than `Device1`).

Add I/O Device

Device Kind

☒ **Modbus / EtherNet-IP device (from IODF)**
Physical I/O module or third-party device with a known register map

☐ **Arena TCP (another controller)**
Connect to another PLC or HMI via its arena agent. Tags are imported from the source project.

☐ **EtherNet/IP PLC (Allen-Bradley, by tag name)**
CompactLogix / ControlLogix / Micro800 via CIP explicit messaging. Tags are added by name after the device is created.

Device Type

Select Device Type
S15C-B22-MQ

Choose from available IODF definitions

Manufacturer: **Banner Engineering**

Part Number: **223063**

Connection: **Modbus RTU**

I/O Points: **2 DI, 2 DO, 20 AI, 31 AO**

Bimodal to Modbus Converter - 2 discrete inputs/outputs with counting, timing, totalizer, and events-per-minute measurement. Pin 2 and Pin 4 channels independently configurable as NPN/PNP input or output. RS-485 Modbus RTU, default 19200 baud, no parity, slave address 1.

Instance Name
S15C_B22_MQ

Unique name for this device instance

Connection Settings

Port
COM1 - RS-485

Slave ID
1

[Cancel](#) [Add Device](#)

7.3.1 Modbus devices from an IODF

1. Under **Device Type**, open **Select Device Type** and choose the module.
The card below shows its **Manufacturer**, **Part Number**, **Connection** (Modbus RTU or Modbus TCP) and **I/O Points** summary (for example 4 DI, 4 DO, 4 AI, 4 AO).
2. Enter the **Instance Name**.
3. Complete **Connection Settings**. Whether the device is RTU or TCP is decided by the IODF:

Connection	Fields	Default
Modbus RTU	Port (rs485 or rs485_2 on the Nexus.io controller; shown as a fixed value when the target has only one), Slave ID (1–247)	1
Modbus TCP	IP Address, Subnet Mask, TCP Port, Unit ID (1–247)	255.255.255.0, 502, 1
Both	Max Delay (ms) — how long the scanner waits for a reply before counting a timeout	500

There is no scan-rate field. The controller polls Modbus devices every 500 ms and Arena TCP devices every 200 ms; Max Delay is the per-request timeout.

Note. If the dropdown is empty the dialog says **No IODF files found** and offers **Create Sample IODFs**. See *Where IODF files live* below.

7.3.2 Arena TCP devices

An Arena TCP device is a connection to another controller's arena agent. Enter:

- **Instance Name** — the connection label (the dialog suggests `plc`); this prefixes nothing by itself, but the Import Tags dialog offers it as the default tag prefix.
- **IP Address** — the source controller.
- **TCP Port** — the arena agent's remote listener, default 5000.
- **Max Delay (ms)** — timeout, default 500.

- **Secure connection (NexusSec — mutual TLS)** — tick it when the source controller runs its arena listener with TLS. The connecting station then needs a certificate the controller trusts; see *Security, Provisioning and Enrolment* and, for a PC panel, *The WhiskerHMI PC Application*.

The device carries no tags until you import them; the dialog reminds you to use **Import Tags** on the device row afterwards.

7.3.3 EtherNet/IP PLCs

Enter the **Instance Name**, the PLC's **IP Address** (the PLC or its Ethernet module), the **PLC Type** (CompactLogix, ControlLogix, Micro800, MicroLogix, SLC500 or PLC5), the **CIP Path** (shown for the Logix families only; `1,0` means backplane, slot 0 — change the second number when the CPU sits in another slot; Micro800, MicroLogix, SLC500 and PLC5 take no path), the **TCP Port** (default 44818) and a per-tag **Timeout (ms)** (default 2000). The IDE supports these device types through a generic tag-name protocol; check your PLC's documentation for the tag scopes it allows external clients to read and write. Tags are added afterwards with **PLC Tags** on the device row.

Both targets talk to the PLC the same way. On a Nexus.io controller the I/O scanner opens one connection per PLC and polls every mapped tag five times a second; on a WhiskerHMI PC the scanner service does the same. No extra software is installed on the PLC, but its Ethernet module must allow unconnected (class 3) messaging from the controller's address.

7.4 The Configured Devices table

Column	Meaning
Instance	The instance name
Type	The IODF name, or <code>ArenaTCP</code> / <code>EtherNetIP</code> for the virtual kinds
ID	Slave or unit ID
%I, %Q, %IW, %QW	The first address the device occupies in each memory area, or <code>-</code> when it has no points there

The action column holds icon buttons (hover for the tooltip):

- **Import tags from source PLC** — Arena TCP devices only; opens the Import Tags dialog.
- **PLC Tags** — add EtherNet/IP tags by name — EtherNet/IP devices only.

- **Delete** — asks *Delete Device* and removes the device together with its I/O points and device-created tags. Ladder or HMI references to those tags become undefined; the undefined-tag sweep reports them at the next Build.

Select a row to edit the device in the Properties panel. The fields follow the device kind: a Modbus device shows **Instance Name**, **Slave ID**, **Serial Port** (when the target has more than one), **IP Address**, **Subnet Mask**, **TCP Port** and **Max Delay (ms)**; an Arena TCP device shows **Instance Name**, **IP Address**, **Arena Port**, **Max Delay (ms)** and the **Security** section with the Secure connection checkbox; an EtherNet/IP PLC shows **Instance Name**, **IP Address**, **TCP Port**, **PLC Type**, **CIP Path** (Logix families only) and **Timeout (ms)** — the same settings as the Add I/O Device dialog, so a PLC can be moved to another address or slot after its tags exist. Changing the PLC Type resets the CIP Path to that family's default. Changes save as you type.

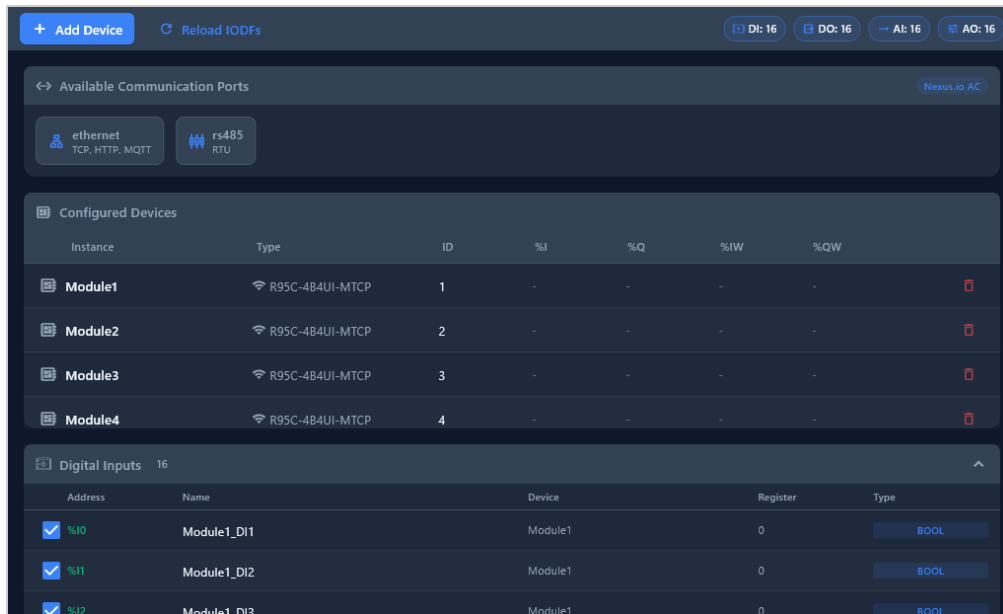
7.5 I/O points and their tags

When you add a Modbus device the IDE creates **one tag per channel** in the IODF, named `<Instance>_<Register>` — a module named `FieldIO` with a register `DI1` produces `FieldIO_DI1` — and assigns it the next free address in the matching area (`%I` for digital inputs, `%Q` for digital outputs, `%IW` for analog inputs, `%QW` for analog outputs). The four point tables show:

Column	Meaning
(checkbox)	Enable. Checked: <i>Polled by IO scanner</i> — <i>uncheck to disable</i> . Unchecked rows are dimmed and skipped by the scanner.
Address	The tag's address
Name	The tag name
Device	The instance the point belongs to
Register	The register name from the IODF
Type	BOOL, INT, UINT, DINT, UDINT or REAL
Range	Analog tables only. A dropdown when the IODF declares more than one input or output range (for example <code>0-10 V</code> or <code>4-20 mA</code>); otherwise the single range, or —

To rename a point's tag, click its row; the tag opens in the Properties panel and you edit the name there (it commits when you press Enter or leave the field). Renaming keeps the address and every ladder or HMI reference, so rename generously: `Tank1_HighFloat` reads better than `FieldIO_DI3`.

You cannot disable a point whose tag is still used. The editor answers *Cannot disable "... — it is used in ...*; remove the reference first.



7.6 System tags for every device

The moment a device is added, the IDE creates three read-only system tags for it in the Tag Database:

Tag	Type	Meaning
<Instance> Comm OK	BOOL	Device reachable and polling
<Instance> Comm Age	DINT	Seconds since the last good poll
<Instance> Error Type	DINT	Last error: 0 none, 1 timeout, 2 connection lost, 3 protocol, 4 device error, 5 configuration

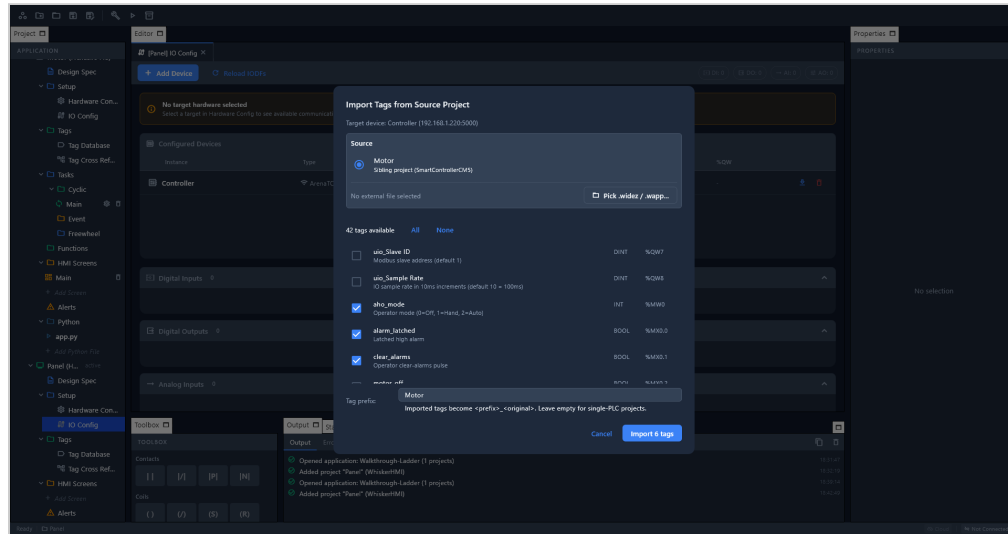
Use them in ladder and on HMI screens to alarm on a lost module. They are removed automatically when the device is deleted. See *Tag Database* for the other system tags.

7.7 Importing tags from another controller

Tags for an Arena TCP device come from the **source project**, not from the live controller: the IDE copies their names, types and addresses from the project file, so both sides stay in step. Click **Import tags from source PLC** on the device row. The **Import Tags from Source Project** dialog shows:

1. **Source** — one option per sibling project in the open Application, or **Pick .widez / .wapp...** to browse to another Application archive. If the archive holds several projects, pick one.
2. The tag list, with the count (`12 tags available`) and **All / None** buttons. Every tag is a checkbox row showing name, type and address; all are ticked to start with.
3. **Tag prefix** — imported tags become `<prefix>_<original>`. The field is pre-filled with the source project name; clear it when the panel reads a single controller so the names stay the same on both sides. Keep a distinct prefix per controller when one panel reads several.

Click **Import N tags**. Each imported tag gets a local address in the same memory area, remembers which device and source tag it came from (shown in the Tag Database **Source** column), and reads or writes the source controller's tag at runtime. The dialog refuses an import whose names collide with existing tags and suggests a different prefix.



Import Tags from Source Project dialog

7.8 Adding EtherNet/IP tags

Click **PLC Tags** on an EtherNet/IP device row. The **PLC Tags** — dialog has an entry row — **Local tag name**, **PLC tag path**, **Type** and **Add** — and a table of the tags added so far (**Local tag**, **PLC tag path**, **Type**, **Address**, with a *Remove* button per row).

- Enter the PLC tag path exactly as it appears in the PLC's programming software: a bare name for a controller-scoped tag (`TankLevel1`), `Program:Main.PumpSpeed` for a program-scoped tag, and `Pump[2].Speed` for an array element or member.
- **Type** is BOOL, INT, DINT or REAL. The local tag is created in memory (`%MX` , `%MW` , `%MD` or `%MF`) with the description `EtherNet/IP <instance>: <path>` .
- There is no browse function; tag names are typed. Every tag is readable, and an HMI may write any of them — the PLC decides what it accepts.

At runtime each PLC tag behaves like a remote I/O point. Every poll the scanner reads the PLC value into the local tag; when the ladder, an HMI or a cloud write changes the local tag, the new value is written to the PLC on the next poll and the PLC's copy follows. A tag the PLC does not have (a misspelt path, a program-scoped tag typed without its `Program:` prefix) is reported once in the controller log and retried every poll without disturbing the other tags. When no tag answers for three polls running, the device's **Comm OK** system tag drops

and **Error Type** shows 1 (timeout) or 2 (connection lost); the scanner then retries at 5, 10, 20, 40 and 60-second intervals until the PLC is back, and never pushes a stale value to it after a reconnect.

7.9 Banner wireless I/O

Banner Engineering wireless nodes and radios are supported as ordinary Modbus devices: the IDE ships IODFs for the R95C and S15C families and a universal I/O node. Add the radio's Modbus interface with **Modbus / EtherNet-IP device (from IODF)** exactly like any other module; the IODF already maps the node's registers to digital and analog points, including the selectable input ranges.

7.10 IODF files

An IODF is a JSON document (`"format": "iodf/1"`) that describes one device type: its name, manufacturer, part number, connection kind (`rtu` or `tcp`) and its registers, grouped into `holding`, `coils`, `input` and `discrete` tables. Each register has a name, a 0-based address, a data type, the I/O area it belongs to (`DI`, `DO`, `AI` or `AO`), optional bit position for packed words, a gain and units, and optional selectable ranges. The IDE turns that description into the point tables above, so adding a module never involves typing register addresses by hand.

Where the IDE looks:

Location	Purpose
<code>.whisker-ide\iodf</code> in your Windows user profile	The global library. The IDE copies its bundled IODFs here on startup and refreshes them when a new IDE version changes one; files you add are left alone.
<code>iodf</code> folder inside a project	Project-specific files. A project IODF overrides a global one of the same name.

To add a device type that is not shipped, write or obtain its IODF, drop it in either folder and click **Reload IODFs**. Use **Create Sample IODFs** in the Add I/O Device dialog to generate example files to copy from.

Tip. Put a project-specific IODF in the project's `iodf` folder so it travels inside the Application archive to every PC that opens it. {tip}

7.11 Building after I/O changes

Device and point changes take effect on the controller at the next deploy: connect, then press **Build (F5)**; the IDE compiles and downloads the program, including the I/O scanner configuration, to the connected controller. Once the program is running you can watch point values live in the Tag Monitor — see *Monitoring and Debugging*.

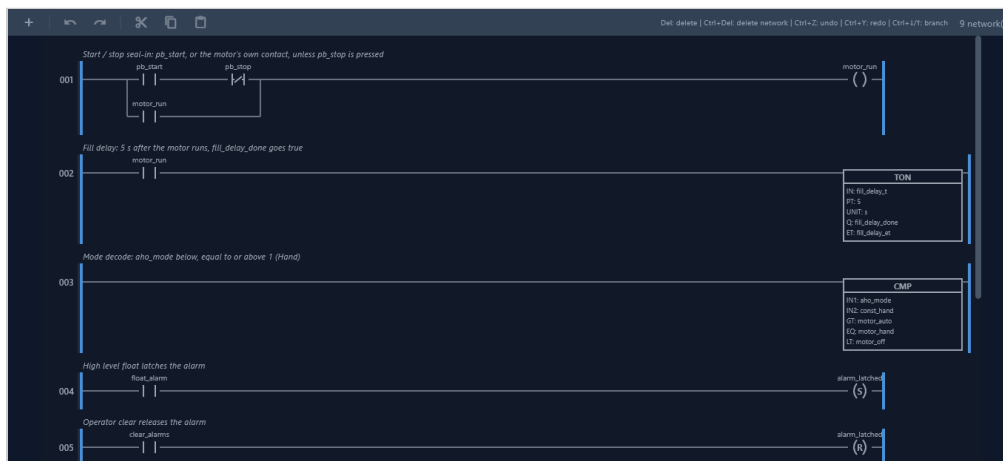
8 Ladder Logic Editor

The Ladder Logic Editor is where you write the program that runs on a Nexus.io Automation Controller. It uses graphical ladder logic: networks of contacts feeding an output, read left to right and scanned top to bottom. This chapter covers the editor itself — the canvas, placing elements, wiring branches, and setting properties. The instructions you can place are catalogued in *Ladder Instruction Reference*.

8.1 Programs, tasks and functions

Every ladder program belongs to a **task** or a **function** and has the same name. A new project comes with one task, **Main**, and its program. In the Project Tree expand **Tasks**; tasks are grouped under **Cyclic**, **Event** and **Freewheel**, and a single click on a task opens its program in the editor. Functions are listed under **Functions** and open the same way.

To add a program, add a task: click the toolbar button **New Task** (Ctrl+Shift+N). To add a reusable subroutine, click **New Function** (Ctrl+Alt+N). Both are described in *Function Blocks, Faceplates and Tasks*. A new program starts with five empty networks.



8.2 Anatomy of a network

A **network** is one line of logic. Other platforms call it a rung; the Whisker IDE uses “network” throughout, and numbers them 001, 002, ... down the left edge.

- The **main level** runs from the left rail to the output. It has **16 segments**, each of which can hold one contact or one inline comparison, or be a plain wire.
- **Branch levels** are extra rows below the main level. A branch leaves a segment on the level above and rejoins it further right, giving a parallel (OR) path.
- The **terminal** at the right end is the network’s single output: a coil, or a function block such as a timer, counter, comparison, math or device block.

The network is true when at least one path of closed contacts reaches the terminal. Contacts in series are AND; branches are OR.

One output per network. A network has exactly one terminal. To drive two outputs from the same condition, write two networks with the same contacts. Parallel contact paths feeding one output are fine; parallel outputs are not.

Networks in a program execute in order, once per scan of the task that owns the program. All networks in all tasks run every scan, at the rate each task is configured for.

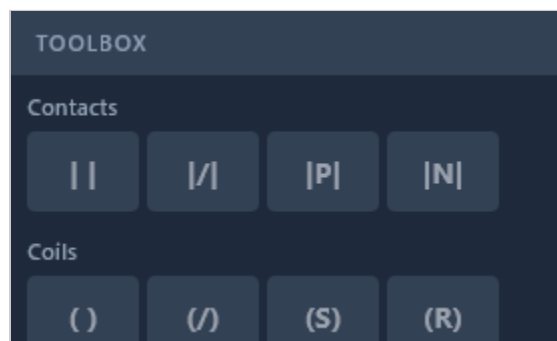
8.3 The Toolbox

The **Toolbox** panel lists every element you can place, in groups:

Group	Elements
Contacts	NO Contact, NC Contact, P Contact (rising edge), N Contact (falling edge)
Coils	Coil, Neg Coil, Set Coil, Reset Coil, Flash
Timers	TON, TOF, TP
Counters	CTU, CTD, CTUD
Edge Detection	R_TRIG, F_TRIG
Bistables	SR, RS
Compare	EQ, NE, GT, GE, LT, LE (inline comparisons), CMP (three-output block)

Group	Elements
Data Handling	MUX, PACK, UNPACK
Math / Data Movement	MOVE, ADD, SUB, MUL, DIV, MOD, MIN, MAX, ABS, SHL, SHR, ITOF, FTOI, LIMIT, SEL, SCALE
Bitwise	AND, OR, XOR, NOT, ROL, ROR, SWPB, SWPW
Advanced Math	SQRT, ROUND, TRUNC, EXPT, LN, LOG, EXP, SIN, COS, TAN, ASIN, ACOS, ATAN
Process	HYST, TONR, TOTAL, FILTER, ROC, PID, ALT, RTC, SCHED
Industrial	MEQ, DEBOUNCE, MOTOR, VALVE, FIRSTOUT, DRUM
Program Flow	PYFB, CALL, RET

Hover an item for a one-line description.



8.4 Placing elements

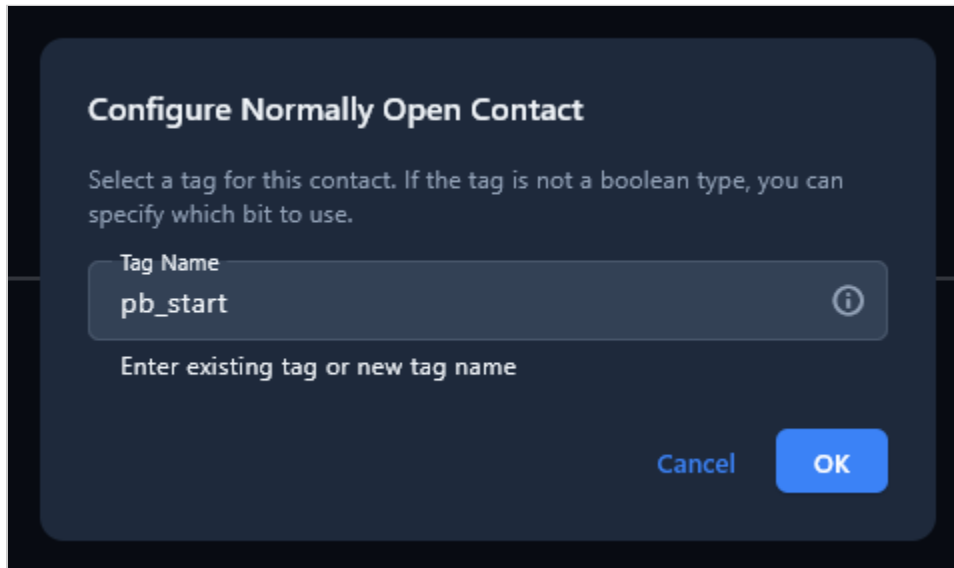
Elements are placed by **dragging** them from the Toolbox onto the canvas.

- **Contacts and inline comparisons** drop onto a **segment**. The segment highlights as you drag over it.
- **Everything else** — coils, timers, counters and every block — is a terminal. Drop it anywhere on the network and it replaces that network's output.

When you drop a contact or coil, a small dialog asks for the tag: **Configure Normally Open Contact** asks for a **Tag Name** (and a bit index if the tag is not a BOOL); a coil dialog asks for an **Output Tag**. Type a name or pick one from the

list and click **OK**. If the name is not in the Tag Database yet, the **Create New Tag** dialog follows so you can create it on the spot; cancelling it cancels the placement.

Timers and the other blocks drop with their pins empty and are configured in the Properties panel.



8.5 Selecting and the Properties panel

Click a segment or the terminal to select it; the **Properties** panel on the right shows its parameters. Click empty canvas or press **Esc** to clear the selection.

Double-click an element to bring the Properties panel to the front. This matters when you have docked the Toolbox and Properties in the same tab group: a single click selects, a double-click also raises the panel.

What the panel shows depends on the element:

- A contact: **Tag Name** and **Bit Index (for non-BOOL)** — a contact on a word tag can test one bit, shown on the canvas as `status_word.3`.
- An inline comparison: **Operator**, **IN1 (tag or number)** and **IN2 (tag or number)**.
- A coil: **Output Tag** and a bit index.
- A timer: **Instance Name**, **Preset (PT) tag or number**, **Time Unit (UNIT)** (Seconds or Milliseconds), and under **Output Tags** the **Done Output (Q)** and **Elapsed Time (ET, always ms)**.
- Other blocks: an instance name plus one field per pin, named as in *Ladder Instruction Reference*.

Every tag field is an autocomplete: start typing and matching tags appear with their type and address. A value pin (a preset, a setpoint, a comparison operand) accepts either a tag name or a number; a number becomes a constant in the compiled program and does not create a tag. When you leave a field holding a name that does not exist, the IDE offers to create the tag (see *Tag Database*).

Instance names. Timers, counters and every compute or device block need an instance name, which identifies the block's private state on the controller. An instance is not a tag: it has no address and cannot be monitored or forced. To watch a value, wire an output pin to a tag.

PROPERTIES

Timer On Delay (TON)

Instance Name

fill_delay_t

Preset (PT) tag or number

5

Numeric literal (constant value)

Time Unit (UNIT)

Seconds

Output Tags

Done Output (Q)

fill_delay_done

BOOL @ %MX6

Done Bit Index

For non-BOOL tags

Elapsed Time (ET, always ms)

fill_delay_et

DINT @ %MD0

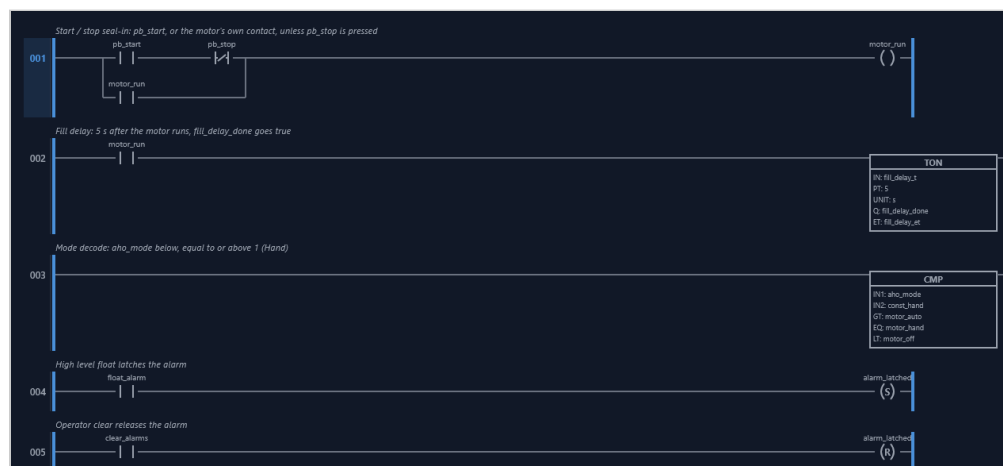
8.6 Branches

Branches are wired from the keyboard. Select the segment where the branch should start, then:

Key	Action
Ctrl+Down	Create a parallel branch below the selected segment, starting at that column. If a branch already starts there, the selection moves down onto it.
Ctrl+Up	Reconnect the branch to the level directly above, at the selected column. The level above must have wire at that column.
Ctrl+Shift+Up	Reconnect the branch straight to the main level, skipping empty levels in between.
Ctrl+Right	Extend the branch one segment to the right (moves the selection right along the branch).
Left / Right	Move the selection one segment.

A branch does not rejoin on its own — after placing the last element on it, press **Ctrl+Up** so both paths feed the output. An open branch is drawn as far as its last element so you can see what you have placed.

To build a start/stop seal-in: drop an NO `pb_start` contact on segment 1, an NC `pb_stop` on segment 2 and a `motor_run` coil as the terminal; select the `pb_start` segment, press **Ctrl+Down**, drop an NO `motor_run` contact on the new branch, and press **Ctrl+Up**.



Branches nest: a branch can itself have a branch below it, reconnecting either to its parent or (with Ctrl+Shift+Up) to the main level.

8.7 Deleting

Delete acts on the selected segment or terminal, peeling back one thing at a time: first the element in the segment, then an up-connection on that segment, then a down-branch starting there. On a terminal, Delete removes the block and leaves an empty coil in its place.

Ctrl+Delete deletes the whole network, as long as the program has more than one.

8.8 Networks

- **Add Network** — the **+** button at the top left of the editor appends an empty network at the bottom. The count is shown at the right of the toolbar as *n network(s)*.
- Networks are numbered automatically and run in the order shown; the editor has no move or insert-between commands, so plan the order as you add them.
- **Network comments.** Click a network's number (the three-digit label at the left of the rung, or an existing comment line) to select the whole network — the number turns blue and the Properties panel shows **Network N** with a **Comment** box. Type what the network does and press Enter (or click elsewhere); the comment appears in grey italics above the rung, to the right of its number, and is saved with the project. The box shows four lines; a longer comment scrolls inside it and a scrollbar stays visible to say so. A blank comment removes the line. The compiler copies each comment into the program listing.

8.9 Undo, redo and the clipboard

Ctrl+Z undoes and **Ctrl+Y** redoes edits on the canvas — placements, wiring, deletions and pastes — up to 50 steps back. The editor toolbar's **Undo** and **Redo** buttons do the same and grey out when there is nothing to undo or redo.

Copy (Ctrl+C), **Cut (Ctrl+X)** and **Paste (Ctrl+V)** work on whatever is highlighted:

Highlighted	Copy / Cut takes	Paste puts it
A cell with an element (contact, edge contact, inline comparison)	That one element and its tag — never its branch wiring	Into the highlighted cell if it is empty, otherwise into the next empty cell to the right on that row. A full row refuses with a message
The terminal (coil or block)	The block with all its pins	Into the highlighted network's terminal slot. If that slot already holds a real block or a coil with a tag, a dialog asks before replacing it
The network number (whole network)	The entire rung — cells, branches and terminal	Below the highlighted network, or at the end when nothing is highlighted; the pasted network is selected

The clipboard is the Windows clipboard, so a rung or a block can be copied between two projects or two IDE windows. A pasted block gets an instance name that is unique in the program (`T1` becomes `T1_2`) so the copy never shares its timer or counter state with the original; tags come along by name and the undefined-tag sweep offers to create any the target project lacks. Cut leaves an empty cell or a blank coil behind and, for a network, removes the rung; the last network cannot be cut. Right-click a network number for **Copy network**, **Cut network**, **Paste below** and **Delete network**.

The help strip along the editor's toolbar summarises the bindings: *Del: delete* | *Ctrl+Del: delete network* | *Ctrl+Z: undo* | *Ctrl+Y: redo* | *Ctrl+↓/↑: branch*.

8.10 Comparisons

There are two ways to compare values:

- An **inline comparison** (EQ, NE, GT, GE, LT, LE) sits in a segment like a contact and passes power when the test is true: `level > 80` . Its operands are tags or numbers. Use it for ordinary tests.
- The **CMP block** is a terminal that compares IN1 with IN2 and writes three BOOL tags at once — **GT**, **EQ** and **LT** — so one network decodes a mode word into three flags.

For start/stop levels with a deadband, use **HYST** instead of two comparisons.

8.11 Tags from the editor

You rarely need to open the Tag Database while writing ladder. Type the tag name you want in the drop dialog or the Properties panel and create it when asked; the suggested type comes from the pin. Before a build, the undefined-tag sweep catches any name that slipped through and offers to create all of them in one dialog — the build waits until every referenced tag exists. On Save the same dialog appears but never blocks. See *Tag Database*.

8.12 Building, emulating and monitoring

With a controller connected, press **Build (F5)**; the IDE compiles and downloads the program to the connected controller. Compile messages, including lint warnings such as a DRUM with nothing wired to its pattern output, appear in the **Output** panel. **Build & Emulate** compiles and runs the program on your PC without a controller. See *Connecting, Deploying and Emulating*.

Common compile errors:

- **Undefined tag** — the sweep normally prevents this; create the tag and build again.
- **PYFB has no Python function name** — a PYFB block was dropped but the function it calls was never entered in the Properties panel. A DRUM with no steps in its table is refused the same way.
- **Python syntax error in app.py** — the project's Python files are checked before the bundle is built.

Once the program is running, the editor shows live power flow and pin values (see *Monitoring and Debugging*). While any tag is forced, a **FORCES (n)** — **click to clear** chip appears in the editor toolbar; click it to release every force.

8.13 Keyboard reference

Key	Action
Esc	Clear the selection
Left / Right	Move the selection along the level
Ctrl+Down	New branch below the selected segment
Ctrl+Up	Reconnect the branch to the level above
Ctrl+Shift+Up	Reconnect the branch to the main level

Key	Action
Ctrl+Right	Extend the branch to the right
Delete	Delete the element, up-connection or branch at the selection; reset the terminal
Ctrl+Delete	Delete the network
Ctrl+Z / Ctrl+Y	Undo / Redo
Ctrl+C / Ctrl+X / Ctrl+V	Copy / Cut / Paste the highlighted element, block or network
F5	Build (compiles and downloads to the connected controller)

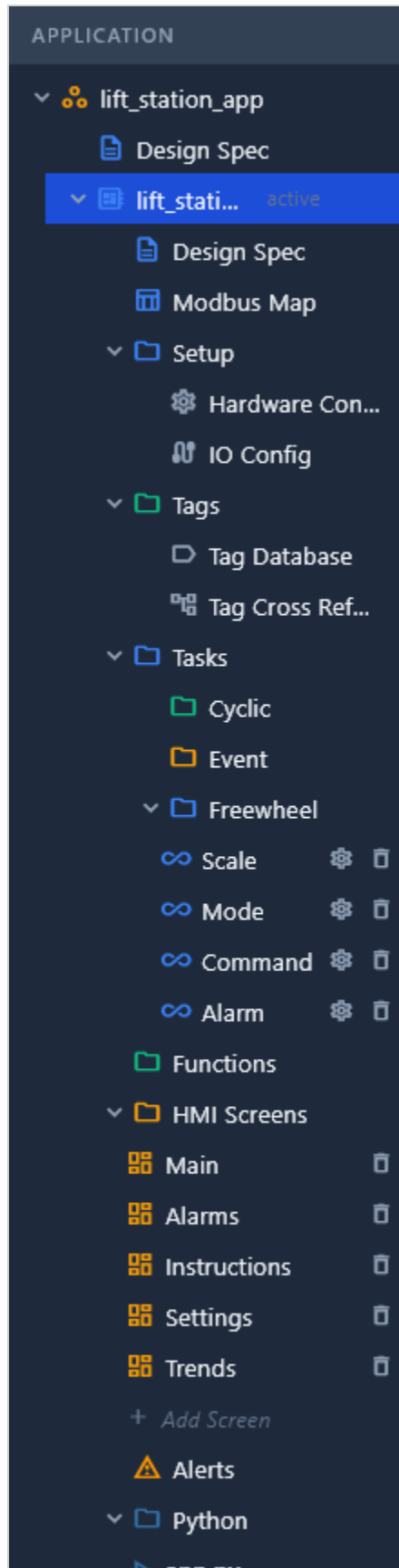
9 Function Blocks, Faceplates and Tasks

This chapter covers how a Nexus.io Automation Controller program is organised into **tasks**, how **functions** and **function blocks** package reusable logic, how a device block becomes an HMI **faceplate** in one click, and where Python fits in.

9.1 Tasks

A **task** is a scheduling unit. Each task owns exactly one ladder program with the same name, and the controller runs that program on the task's schedule. A new project has one task, **Main**, of kind **Cyclic** with a 10 ms period and priority 1.

In the Project Tree, **Tasks** has three sub-folders — **Cyclic**, **Event** and **Freewheel** — and each task appears under its kind. A single click on a task opens its ladder program. The gear icon on the task row opens **Task Configuration**; the trash icon deletes the task and its program after a confirmation.



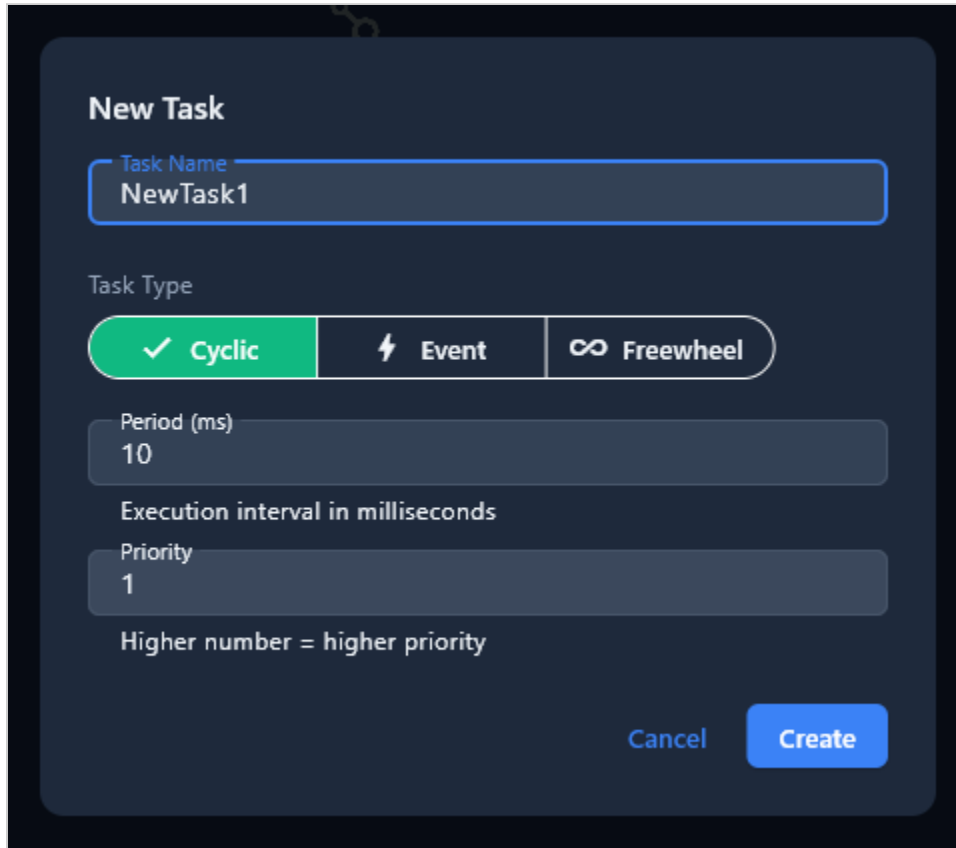
9.1.1 Task kinds

Kind	Behaviour
Cyclic	Runs on a fixed period, set in milliseconds. The right choice for a single-task project and for anything that needs a known rate.
Freewheel	Runs as fast as the controller can dispatch it. When several freewheel tasks exist, the ready task with the highest priority runs first; once it has run it waits until every other freewheel task has had its turn, then the cycle repeats. The result is round-robin scheduling in priority order. Freewheel tasks have no period.
Event	Runs once each time one bit changes: a digital input, a digital output or a memory bit, chosen by Memory Area (DI, DQ or Mb) and Index . Edge Type picks which change counts — Rising (off to on), Falling (on to off) or Both — and Debounce (ms) is how long the new level must hold before the change is accepted; a shorter blip is ignored. The bit is sampled once per scan tick, and a bit that is already on when the program starts is not a rising edge. The task runs at its priority alongside whatever cyclic tasks are due the same tick. To react to an analog value crossing a threshold, compare it in a cyclic task and set a memory bit; the event task watches that bit.

Priority is a whole number; **a higher number means higher priority**. The default is 1.

9.1.2 Creating a task

Click **New Task** in the toolbar (Ctrl+Shift+N).



New Task

Task Name
NewTask1

Task Type

☒ Cyclic
 ☐ Event
 ☐ Freewheel

Period (ms)
10

Execution interval in milliseconds

Priority
1

Higher number = higher priority

Cancel

- **Task Name** — must be unique among tasks and functions; it is also the program name.
- **Task Type** — Cyclic, Event or Freewheel.
- **Period (ms)** — Cyclic only; the execution interval, default 10.
- **Memory Area** (DI, DQ or Mb), **Index** (the bit number in that area), **Edge Type**, **Debounce (ms)** — Event only. The same four fields are edited later in Task Configuration.
- **Priority** — default 1.

Click **Create**. The task appears in the tree and its empty ladder program opens.

9.1.3 Task Configuration

Click the gear icon on a task in the Project Tree. The **Task Configuration** editor shows **Name**, **Type**, **Period (ms)** (Cyclic), **Priority**, and a read-only **Ladder Program** row that names the program (*1:1 - same as task*). Change what you need and click **Save Changes**. Switching a task away from Cyclic clears its period.

9.1.4 Structuring a program with freewheel tasks

A single cyclic Main task is enough for most projects. Larger programs read better split into freewheel tasks by responsibility, with priorities that fix the order they run in each cycle, for example:

Task	Priority	Contents
Scale	4	Convert raw analog inputs to engineering units
Mode	3	Decode operator modes and permissives
Command	2	Start/stop logic, timers, device blocks
Alarm	1	Alarm conditions and latches

Each cycle runs Scale, Mode, Command, then Alarm, so every downstream task sees this cycle's scaled values and modes. All tasks share the same tags; a task boundary is a scheduling boundary, not a data boundary.

9.1.5 Choosing a period

- Match the process. A tank that fills over minutes does not need a 5 ms scan; a pulse input that changes every 50 ms does.
- Timers and edge contacts resolve to one scan, so the period sets the resolution of everything time-based in the program.
- Ten milliseconds is the default and suits most control logic.

9.2 Functions

A **function** is a named ladder program that can be called from a task or another function. Use it to keep a long program readable — the pump logic in one place, the alarm logic in another.

Click **New Function** in the toolbar (Ctrl+Alt+N). The **New Function** dialog asks for a **Function Name** (a valid identifier, unique among tasks and functions) and an optional **Description**. Click **Create** and the function opens in the ladder editor; it appears in the Project Tree under **Functions**, where the gear icon opens **Function Configuration** (name, description, and the read-only program name) and the trash icon deletes it.

A function has no parameter list. It reads and writes the project's tags directly, exactly as a task does, so the caller and the function share data through the Tag Database.

To call a function, drag **CALL** from the **Program Flow** group of the Toolbox onto a network. The **Call Function (CALL) Configuration** dialog offers an optional **Instance Name** and a **Function to Call** dropdown listing the project's functions. When the network is true the function's networks run, top to bottom, and the caller continues with its next network; when it is false the function is skipped for that scan. A function runs every time it is called, so a function called from two networks runs twice per scan.

RET returns when its network is true: inside a function, back to the caller; inside a task, it ends that task's scan and the networks below it are skipped until the next scan. A RET whose network is false does nothing.

A function may call other functions, up to 31 levels deep, but not itself; the build refuses a CALL that names the function it sits in, one that names no function, or one whose function no longer exists, and the Output panel names the program and network.

9.3 Function blocks

Function blocks are the built-in instructions with state — timers, counters, comparisons, math, process and device blocks. Each placed block is an **instance** with its own private memory on the controller, named by the **Instance Name** in the Properties panel. Two timers named `t_fill` and `t_drain` never interfere; two blocks with the same instance name would.

Instance memory is cleared every time a program is downloaded. That is why every block that accumulates something — TONR's elapsed time, TOTALIZER's total, MOTOR's start counter and runtime, DRUM's current step — keeps that value in a **tag you nominate** rather than inside the block. The block reads the tag in and writes it back every scan. Make that tag **VAR_RETAIN** in the Tag Database and the value survives power cycles and downloads; a value written to the tag from the panel or over Modbus presets or resets the meter. Use one tag per block; two blocks sharing an accumulator tag fight over it. Block instances themselves cannot be retained or made constant; those classes apply to tags only.

The full catalogue, with every pin, is in *Ladder Instruction Reference*.

9.4 Faceplates

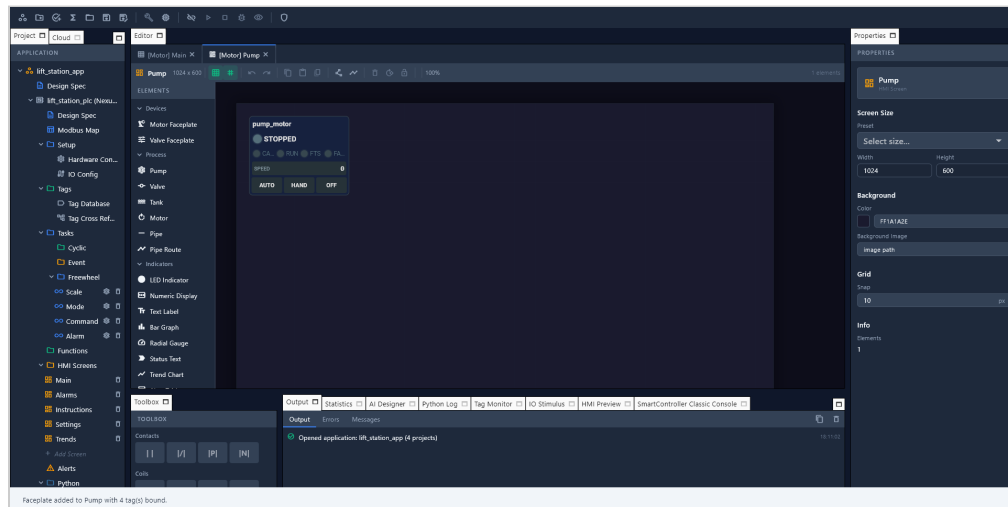
A **faceplate** is a single HMI element that shows one device — a motor or a valve — with its mode buttons, status lamp and annunciators, already bound to the tags of one MOTOR or VALVE block. Built by hand, the same view is a lamp, a

selector, several buttons and two displays with a dozen bindings to get right, repeated for every motor. Because MOTOR and VALVE are single blocks, the IDE knows every tag the device uses and can wire the faceplate for you.

9.4.1 Creating a faceplate from the ladder

Select a **MOTOR** or **VALVE** block on the canvas. Below its pin fields in the Properties panel is **Create HMI Faceplate**. The button is enabled once the block has at least one of its feedback, fault or mode pins wired and the project has an HMI screen; otherwise the reason is shown beneath it (*Add an HMI screen first.* or *Wire the block's outputs first ...*). Click it; if the project has more than one screen, choose one in **Add faceplate to which screen?**. A **Motor Faceplate** or **Valve Faceplate** element is placed below the existing elements on that screen, titled with the block's instance name, and the message **Faceplate added to with n tag(s) bound** confirms it.

The generator binds, for a motor: **Mode** to the MODE tag (read and write), **Running** to RUN_FB, **Failed to Start** to FTS, **Fault** to FLT and **Speed Setpoint** to SP (read and write). For a valve: **Mode** to MODE (read and write), **Open Limit** to OPEN_FB, **Closed Limit** to CLOSE_FB, **Failed to Open** to FTO and **Failed to Close** to FTC. Pins holding a number rather than a tag are skipped. Clicking the button again adds a second faceplate; it never replaces the first.



9.4.2 Placing faceplates from the HMI palette

Both faceplates are also ordinary HMI elements in the **Devices** category of the HMI Designer palette — **Motor Faceplate** (190 × 150) and **Valve Faceplate** (190 × 120). Place one and bind it in the Properties panel like any widget when you want a faceplate for a device that is not a MOTOR or VALVE block, or a second

layout of your own. The valve bindings are **Mode (MODE)**, **Open Limit (OPEN_FB)**, **Closed Limit (CLOSE_FB)**, **Failed to Open (FTO)** and **Failed to Close (FTC)**.

9.4.3 What the operator sees

- **Motor:** the device name; a RUNNING / STOPPED lamp; CALL, RUN, FTS and FAULT annunciators; a SPEED entry box when a speed setpoint is bound; and AUTO / HAND / OFF mode buttons that write 2, 1 or 0 to the MODE tag.
- **Valve:** the device name; a position lamp (green open, red closed, amber in between or with no limit switches); FTO and FTC annunciators; and OPEN / CLOSE / AUTO / OFF buttons that write 1, 2, 3 or 0 to MODE.

Properties on both: **Device Name**, **Show Meters** and **Body Color**. The mode values are the same numbers the ladder blocks use, so a faceplate and a selector switch bound to the same MODE tag always agree.

9.5 Python as a callable block

Ladder can call a function written in the project's `app.py` through the **PYFB** block: each rising edge of the network sends the block's arguments to Python and, when the function returns, the result lands on the block's OUT pin with DONE set. The Python side is one decorated function:

```
from whisker import pyfb

@pyfb("calc_dose")
def calc_dose(ctx, flow_gpm, target_ppm):
    return flow_gpm * target_ppm * 0.012
```

PYFB is the right tool for a curve fit, a lookup table or a calculation that is ten lines of Python and forty networks of ladder. It is not for per-scan math — a round trip is about 20 ms — and the function must return quickly. The block's pins are in *Ladder Instruction Reference*; the Python side, including how `app.py` is deployed, is in *Python on the Controller*.

10 HMI Designer

The HMI Designer is where you build operator screens: the touch screens shown on the display of a Nexus.io Automation Controller, and the windows of a WhiskerHMI PC application. The same designer, elements and properties serve both; the differences are the screen size and where the screens run. This chapter is the reference for the designer. For running screens on a PC see *The WhiskerHMI PC Application*; for trend data see *Historian and Trends*; for alarms see *Alerts Editor*.

10.1 Concepts

A **screen** is one page of HMI content. A project usually has several (Overview, Setpoints, Alarms). On the controller's display one screen is visible at a time and **Nav Buttons** move between them; on a WhiskerHMI PC each screen can be its own window.

An **element** is one visual component on a screen: a gauge, a push button, a tank, a label. You add elements from the **ELEMENTS** palette on the left of the designer and edit them in the Properties panel on the right.

A **tag binding** ties an element to a tag. Display elements read the tag; controls write it. The binding rows in the Properties panel are typed, so a `State (bool)` binding only accepts a BOOL tag and a `Value (real)` binding accepts a numeric one.

10.2 Screens

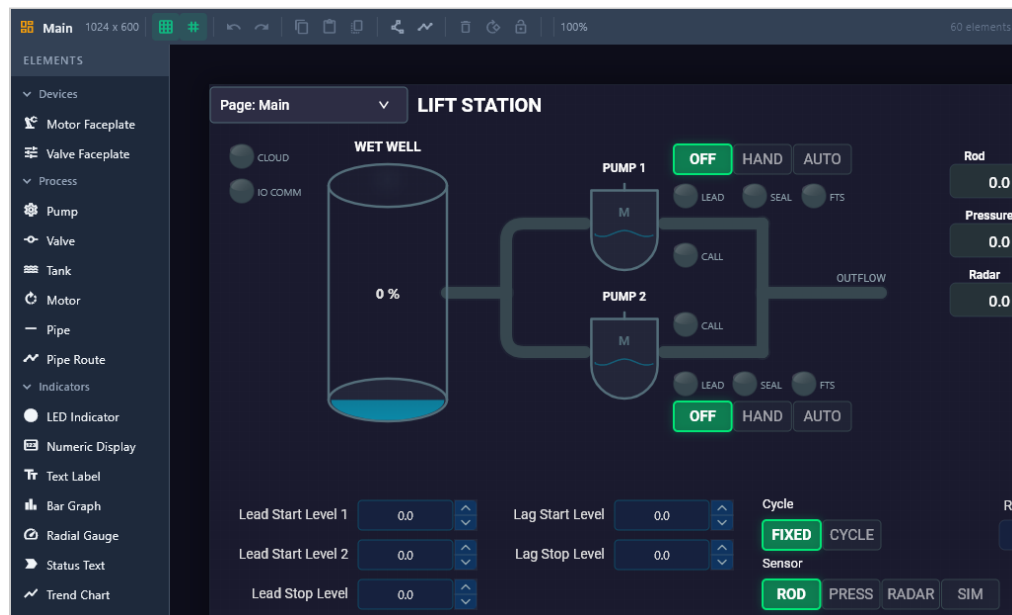
In the Project Tree, **HMI Screens** lists the project's screens. Click **Add Screen** under it; the **Add HMI Screen** dialog asks for a **Screen Name** (the suggestion is `Screen1`) and opens the new screen in the designer. The screen is created at the target's native size: **1280 × 800** for the Nexus.io controller's 10-inch display, **1920 × 1080** for a WhiskerHMI project.

Right-click a screen for **Copy** or **Delete**; right-click another project's HMI Screens folder for **Paste**. Pasting into a WhiskerHMI project that imported the source project's tags remaps every binding to the imported names (including any prefix) and reports how many were remapped; if the tags are not there yet the paste is refused with a list of what to import first. Screen names cannot be renamed after creation, so choose them carefully.

Click an empty part of the canvas to select the screen itself. Its Properties panel shows:

Section	Fields
Screen Size	Preset — 10" panel (1280×800) , 7" panel (1024×600) , 1920×1080 (FHD) , 1600×900 , 1366×768 , 1280×1024 , 1024×768 . Width and Height display the result and are not typed directly.
Background	Color (swatches or an AARRGGBB hex value) and an optional Background Image path
Grid	Snap — the snap spacing in pixels (1–50); this is a designer setting shared by all screens
Info	Elements — the element count

There is no theme system. The look of a screen is its background plus the colour properties of each element.



10.3 The ELEMENTS palette

The palette is grouped into five collapsible sections. Elements marked *writes* send operator input to their bound tag.

Group	Elements
Devices	Motor Faceplate, Valve Faceplate
Process	Pump, Valve, Tank, Motor, Pipe, Pipe Route
Indicators	LED Indicator, Numeric Display, Text Label, Bar Graph, Radial Gauge, Status Text, Trend Chart, Alert Table
Controls	Push Button (<i>writes</i>), Numeric Input (<i>writes</i>), Slider (<i>writes</i>), Selector Switch (<i>writes</i>), Page Selector, Lock Button
Shapes	Text, Rectangle, Ellipse, Line, Polyline, Image, Nav Button

Notes on the less obvious ones:

- **Motor Faceplate** and **Valve Faceplate** are the operator face of the MOTOR and VALVE ladder blocks: mode selection plus running, failed-to-start and fault feedback for a motor; open and closed limits plus failed-to-open and failed-to-close for a valve. Bind each row to the matching block pin tag. See *Function Blocks, Faceplates and Tasks*.
- **Trend Chart** plots historian data for the tags you list; see *Historian and Trends*.
- **Alert Table** shows the controller's alert history with ACK, CLR and RESET buttons; it needs no bindings. See *Alerts Editor*.
- **Lock Button** ends the operator's login session when tapped. The runtime status bar offers the same lock action, so placing one is optional.
- **Nav Button** switches to another screen; **Page Selector** shows the screen list.
- **Status Text** shows a word per integer value (default `{"0":"OFF","1":"RUN","2":"FAULT"}`).

10.4 Placing and arranging elements

Click an element in the palette. It appears near the top left of the canvas at the next free spot and is selected; **drag** it where you want it. Dragging and resizing snap to the grid when **Snap to Grid** is on. Four corner handles resize a selected

element; Polyline and Pipe Route show a handle per vertex instead. **Draw Polyline** and **Draw Pipe Route** put the canvas in drawing mode: click to add points, double-click or press Enter to finish, Esc to cancel.

Selection: click selects; Shift+click adds to the selection; drag on empty canvas draws a selection rectangle; Ctrl+click picks one element out of a group. Ctrl + mouse wheel zooms (25 % to 400 %); the current zoom is shown on the toolbar.

Designer toolbar, left to right: **Toggle Grid**, **Snap to Grid**, **Undo**, **Redo**, **Copy**, **Paste**, **Duplicate**, **Draw Polyline**, **Draw Pipe Route**, **Delete Selected**, **Rotate 90°**, **Lock/Unlock**, **Alignment Tools** (with two or more selected: Align Left / Center H / Right, Align Top / Center V / Bottom, and with three or more Distribute Horizontal / Vertical), **Group**, **Ungroup**, **Bring to Front**, **Send to Back**.

Keys	Action
Ctrl+Z, Ctrl+Y	Undo, redo
Ctrl+A, Ctrl+C, Ctrl+V, Ctrl+D	Select all, copy, paste, duplicate
Ctrl+G, Ctrl+Shift+G	Group, ungroup
Ctrl+L	Lock or unlock (locked elements cannot be moved, resized or deleted)
Delete	Delete selected
R	Rotate 90°
] and [	Bring forward, send backward; with Ctrl: bring to front, send to back
Arrow keys	Nudge 1 pixel (Shift: 10 pixels), ignoring snap
Esc, Enter	Cancel drawing or deselect; finish a polyline

10.5 Element properties

Select an element and the Properties panel shows, in order: **Position & Size (X, Y, Width, Height, Rotation (degrees))**, **Appearance** (the element's own properties), **Tag Bindings**, and for gated controls a **Security** section (see *Operator login* below).

Each binding row is labelled with its purpose and type, for example `Level1 (real)` or `Output (bool)`; type a tag name or pick one from the list. For a numeric binding the panel adds **Raw Range**, **Scaled Range** and **Units** so a 0–10000 register can display as 0–100 %.

Appearance properties of the most used elements:

Element	Appearance	Bindings
Radial Gauge	Min, Max, Units, Needle, Track, Active Arc	Value
Bar Graph	Orientation, Min, Max, Units, Bar Color, Track Color, Show Scale, Alarm High, Alarm Low	Value
Tank	Shape (rectangle, cylinder, horizontal, v_bottom, standpipe), Body Color, Liquid Color, Show Level %, Capacity	Level, High Alarm, Low Alarm
LED Indicator	Shape, On Color, Off Color, Label, Label Position, Label Width	State
Numeric Display	Font Size, Format, Units, Background, Text Color	Value
Text Label / Text	Text, Font Size, Bold, Color, Align	—
Push Button	Label, Mode (momentary or toggle), Confirm, Color, Active Color, Text Color	Output (writes)
Selector Switch	Positions (2–8), Labels (comma-separated), Orientation, colours	Position (writes an INT)
Numeric Input	Min, Max, Step, Format, Units, Background	Value (writes)
Slider	Orientation, Min, Max, Step, Show Value, Track, Active	Value (writes)
Pump / Motor / Valve	Style or Type, colours, Orientation or Direction	Running, Faulted, limits, Position %

Element	Appearance	Bindings
Pipe / Pipe Route	Pipe Color, Flow Color, Diameter	Flowing
Nav Button	Target Screen (or + New Screen . . .), Label, Color, Text Color	—
Trend Chart	Title, Tags, Time Range (s), Refresh (ms), Y Min, Y Max	—
Image	Image Path, Fit, Opacity	—

A push button in **momentary** mode writes TRUE while pressed and FALSE on release; **toggle** flips the tag on each tap. **Confirm** asks the operator to confirm before writing. Changing an element's Orientation between horizontal and vertical swaps its width and height.

PROPERTIES

 **Numeric Display**
disp_rod

Position & Size

X

750

Y

80

Width

110

Height

34

Rotation (degrees)

0

Appearance

Font Size

16.0

Format

0.0

Units

%

Background

FF263238

Text Color

FFFFFF

Tag Bindings

Value (real)

level_rod

×

Raw Range

Min

to

Max

Scaled Range

0.0

to

100.0

Units

%

10.6 Navigation between screens

Place a **Nav Button** and set **Target Screen**. The dropdown lists the other screens and offers **+ New Screen...**, which creates a screen without leaving the designer. A **Page Selector** gives the operator a list of every screen. On a WhiskerHMI PC, screens assigned a window role open as windows instead; see *The WhiskerHMI PC Application*.

10.7 Previewing screens

The **HMI Preview** tab in the bottom panel group shows any screen of the project at a chosen zoom. Until monitoring is on it draws design-time defaults and ignores presses (the chip reads *not live*). Connect to the controller or run **Build & Emulate**, turn on monitoring, and the chip changes to *LIVE — touch enabled*: the preview shows live tag values and presses write to the tags, so you can exercise a screen from the IDE before an operator ever sees it. See *Monitoring and Debugging* for the monitoring gate.

On the controller, screens run after a deploy: connect and press **Build (F5)**; the IDE compiles and downloads the program, screens included, to the connected controller.

10.8 Operator login at the panel

The Nexus.io controller can require operators to log in before they touch a screen. Operators are the users of your Whisker.io account, each with a personal PIN, so this needs the Cloud edition and an account whose administrator has enabled PIN codes for HMI access.

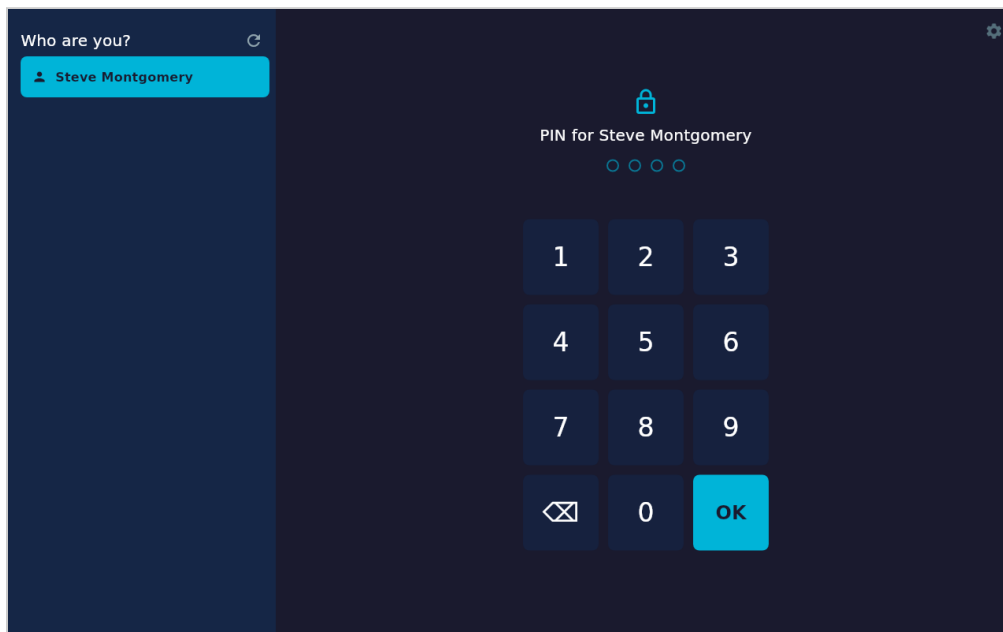
Open Project Properties and find the **HMI Settings** group:

- **PIN** — the project's 4–6 digit PIN that opens the controller's settings screen (network settings and the like). This is a device PIN, not a user login.
- **Cloud PIN Login** — turn on to require a user login at the panel. The row explains what is missing if it cannot be enabled (not logged in to Whisker.io, or PIN codes not enabled for the organization).
- **Session Timeout (s)** — idle time after which the panel re-locks (300 by default); screen-off always re-locks.

- **Offline Fallback** — when on, the project PIN above opens a *Local maintenance* session if Whisker.io cannot be reached. Off is the secure default.

With Cloud PIN Login on, a **Roles at the panel** table appears with one row per action — View screens, Buttons and switches, Setpoints and sliders, Acknowledge alarms, Clear alarms, Save trend presets, Export history, Open windows, Audit log and settings — and a **Required role** for each: *No login needed* (viewing and acknowledging only), *Any logged-in user*, or one of the account's roles. Roles load from Whisker.io; use the refresh button after the account administrator changes them. To make one element stricter than its action, select it and set **Required role** in its **Security** section (push buttons, selector switches, numeric inputs, sliders, faceplates and the Alert Table).

At the panel the operator taps their name in the user picker, then enters their PIN. A wrong PIN is answered with *Not recognised*; after repeated failures that user is locked for a few minutes while other users can still log in. A tap on a control the operator's role does not allow shows a brief *Requires* message. The session ends after the timeout, when the screen turns off, or when the operator taps a **Lock Button** or the lock icon in the status bar. Every login attempt and every gated action is recorded in the audit trail; see *Security, Provisioning and Enrolment*.



11 Alerts Editor

Alerts are alarm rules evaluated on the controller: each one watches tags, drives an output tag while its rule is true, and writes FIRED and CLEARED events to the alert history that the **Alert Table** widget shows on the panel and in a WhiskerHMI PC application. This chapter is the reference for defining them. Placing the Alert Table is covered in *HMI Designer*.

11.1 What an alert is

An alert has:

- a **name** — what operators see in the Alert Table (High Pressure , Pump 2 Overload);
- a **rule** — one condition, or a group of conditions joined by AND or OR. A condition compares a tag with a constant or with another tag;
- an **output tag** — a BOOL tag, or one bit of an INT tag, that is TRUE while the alert is active. Ladder logic and HMI elements can use it like any other tag;
- a **latch** flag — off: the output follows the rule and the alert clears itself; on: the alert stays active after the rule goes false until a **Clear tag** goes true.

There is no severity, no message text and no free-text expression language. Alerts exist in Nexus.io Automation Controller projects and in WhiskerHMI projects; a WhiskerHMI project's alerts are evaluated on the PC from the tags it reads.

11.2 Opening the editor

In the Project Tree, click **Alerts** under the project. The editor is a table with one row per alert:

Column	Contents
Name	The alert name
Output Tag	The output tag, or -
Bit	The bit index when the output is a bit of an INT tag, or -

Column	Contents
Latch	Yes for a latched alert
Rule Summary	The rule in one line, for example Tank_Level > 95.0 or Motor_Run == True AND Flow_OK == False

The last column holds a **Delete Alert** button, which asks for confirmation. Everything else is edited in the Properties panel: click a row to select the alert.

+ Add Alert				
Name	Output Tag	Bit	Latch	Rule Summary
P1 VFD Fault	alarm_p1_vfd_fault	-	Yes	di_p1_vfd_fault == 1.0
P2 VFD Fault	alarm_p2_vfd_fault	-	Yes	di_p2_vfd_fault == 1.0
P1 Seal Failure	alarm_p1_seal	-	Yes	di_p1_seal_failure == 1.0
P2 Seal Failure	alarm_p2_seal	-	Yes	di_p2_seal_failure == 1.0
Rod Sensor Fault	alarm_rod_fault	-	Yes	di_rod_sensor_fault == 1.0
Pressure Sensor Fault	alarm_pressure_fault	-	Yes	sensor_pressure_error == 1.0
Radar Sensor Fault	alarm_radar_fault	-	Yes	sensor_radar_error == 1.0
Sensor Disagreement	alarm_sensor_disagree	-	Yes	state_disagree_debounced == 1.0

11.3 Creating an alert

1. Click **Add Alert** above the table. The **Add Alert** dialog asks for an **Alert Name**; the name must be unique in the project. Click **Add**.
2. The new alert is selected. In the Properties panel, fill in the sections below.

11.4 Alert properties

11.4.1 Alert

Name — edit and press Enter to rename. A duplicate name is not accepted.

11.4.2 Output

Tag — the tag the alert drives. Pick a BOOL tag, or pick an INT tag and enter a **Bit Index (0–15, empty for direct BOOL)** to pack several alerts into one word (the Alert Table and any WhiskerHMI panel read the same bit). Leave the tag at - if nothing needs to react to the alert: the compiler then gives it a private bit in an internal alerts word so it still reaches the alert history. Up to 32 alerts can share that internal word; give the rest an output tag of their own.

11.4.3 Behavior

Latch — a switch. Off: *Clears when condition clears*. On: *Stays active until alarm reset*, and a **Clear tag** dropdown appears: the alert clears when that tag goes true. If you leave Clear tag empty, the alert clears when a tag named `alarm_reset_request` goes true — the shared reset convention that the Alert Table's **RESET** button uses (see *Alert history at runtime*). A ladder network can drive the same tag with a Reset coil from a push button.

Tip. When the condition tag is itself a latched ladder bit (a coil set with a Set coil and cleared by a reset network), make the alert latched too and give it the same clear tag. The alert and the ladder then agree about when the fault is over. {tip}

11.4.4 Rule

The rule starts as a single condition card:

1. **Left tag** — the tag to test.
2. The **operator** — `>`, `>=`, `<`, `<=`, `==` or `!=`. For a BOOL left tag only `==` and `!=` are offered.
3. The right-hand side. By default it is a constant whose editor follows the left tag's type: a **True / False** dropdown for BOOL, a whole number for INT and DINT, a decimal value for REAL. Click the toggle button (*Switch to tag comparison*) to compare with another tag instead (*Switch to constant* changes it back).

Below the card, **+ Condition** adds a second condition and **+ Group** adds a nested group; either turns the rule into a group. Conditions in a group are separated by an **AND** or **OR** chip; click the chip to switch the whole group between AND and OR. Each condition and each nested group has a remove button; a group that drops to one member unwraps itself. This lets you express, for example, *(Level > 95 OR High_Float == True) AND Pump_Running == False* without writing any text.

Tags are chosen from dropdowns of the Tag Database, so an alert cannot reference an undefined tag. If you delete a tag an alert uses, the field shows blank; the Tag Cross Reference lists alerts by name under the tags they use, and a tag used only by an alert is never offered by Remove Unused Tags.

PROPERTIES

Alert

Name

P1 VFD Fault

Output

Tag

alarm_p1_vfd_fault

Bit Index (0-15, empty for direct BOOL)

Direct BOOL

Behavior

Latch

☒

Stays active until alarm reset

Clear tag

Tag that resets the latch

Rule

di_p1_vfd_fault

=

True

+ Condition

+ Group

11.5 How alerts run

Alerts are compiled with the rest of the project: connect and press **Build (F5)**; the IDE compiles and downloads the program to the connected controller. On the controller a separate alert task evaluates every rule continuously, independent of the ladder scan:

- A non-latched alert's output equals its rule.
- A latched alert's output is set when the rule becomes true and cleared when its Clear tag (or `alarm_reset_request`) is true. If the rule is still true when the reset arrives, the alert re-fires on the next evaluation.

Alerts have no compile-time checks of their own. An alert with an empty rule is always true, and an empty group counts as true, so finish every rule before you build.

In a WhiskerHMI project the same rules are written to the PC application's configuration and evaluated there.

11.6 Alert history at runtime

The controller watches every alert's output and appends an event to its alert history each time the output changes: **FIRED** when it goes true, **CLEARED** when it goes false. An alert already active when the controller starts is recorded as FIRED. Latched and non-latched alerts are logged the same way; the difference is only that a latched alert stays FIRED until it is reset. The history keeps the most recent 1000 events on the controller.

The **Alert Table** widget shows that history with columns **ID**, **Timestamp**, **Alert Name**, **State** and **Ack**, newest first, with **All / Active / Cleared** filter chips and a name search. Active rows are red until acknowledged and amber afterwards; cleared rows are grey. Its buttons:

Button	Effect
ACK	Marks the selected events acknowledged (a tick in the Ack column). Acknowledging does not change the alert's output.
CLR	Removes the selected events from the history.

Button	Effect
RESET	Pulses <code>alarm_reset_request</code> , clearing every latched alert that uses the shared reset.

On the controller's own display all three buttons act on the controller. In a WhiskerHMI PC application the Alert Table is a viewer of the controller's history: acknowledge and clear marks made on the PC are not sent to the controller and disappear at the next refresh, and RESET has no effect; reset latched alerts from the panel or from a ladder-driven push button.

With operator login enabled, **Acknowledge alarms** and **Clear alarms** are separate actions in the project's role table, and the Alert Table can carry its own **Required role**; see *HMI Designer*.

Alarms ALARMS					
☐	ACK	CLR	RESET	All	Active Cleared
ID	Timestamp	Alert Name	State	Ack	
☐ 20	10/03 18:56:33	P1 Seal Failure	FIRED	-	
☐ 19	10/03 18:56:33	P2 VFD Fault	FIRED	-	
☐ 18	10/03 18:56:33	P1 Seal Failure	CLEARED	-	
☐ 17	10/03 18:56:33	P2 VFD Fault	CLEARED	-	
☐ 16	10/03 18:56:33	P1 VFD Fault	CLEARED	-	
☐ 15	10/03 18:56:22	P1 Seal Failure	FIRED	✓	
☐ 14	10/03 18:56:22	P2 VFD Fault	FIRED	-	
☐ 13	10/03 18:56:22	P1 VFD Fault	FIRED	-	
☐ 12	10/03 18:37:02	Radar Sensor Fault	CLEARED	-	
☐ 11	10/03 18:37:02	Pressure Sensor Fault	CLEARED	-	

11.7 Removing an alert

Click **Delete Alert** on the row and confirm. There is no disable switch; an alert you do not want evaluated must be deleted (keep a copy of its rule in the project's Design Spec if you expect to restore it).

12 Modbus Server and Map Viewer

A Nexus.io Automation Controller runs a Modbus TCP **server** on port 502. It publishes the tags you choose as coils and holding registers so that SCADA software, a building management system or another PLC can read and write them. This chapter covers marking tags for exposure, controlling their addresses, the read-only **Modbus Map** view, and the CSV exports that hand the map to whoever configures the SCADA side.

Note. This is the controller as a Modbus *server*. Polling Modbus I/O modules, where the controller is the client, is described in *I/O Configuration and Devices*. Both run at the same time. A WhiskerHMI PC application reads the controller over ArenaTCP and does not need Modbus.

12.1 How the server works

There is no enable switch and no port or unit-ID setting in the project: the server listens on port 502 and answers any unit ID. It is part of the controller's standard software; every unit built from the current image runs it from first boot (a unit commissioned by hand from the production guide gets it in that guide's Modbus server step). The server is TCP only. The controller's RS-485 port is a Modbus RTU *master* for I/O modules (see *I/O Configuration and Devices*); the controller cannot be polled as a Modbus RTU slave. It supports Read Coils (FC 01), Read Holding Registers (FC 03), Write Single Coil (FC 05), Write Single Register (FC 06), Write Multiple Coils (FC 15) and Write Multiple Registers (FC 16). Discrete inputs and input registers (FC 02 and FC 04) are not used; everything is a coil or a holding register.

Reading an address that is not mapped returns 0, so SCADA range scans do not fail. Writing an unmapped address is rejected. A write to a mapped register changes the tag on the controller immediately — including a field input, which the I/O scanner overwrites again on its next poll. Nothing in the server refuses writes, so give the SCADA side the map's Access column and keep write permissions on that side.

The map is compiled with the program: connect and press **Build (F5)**; the IDE compiles and downloads the program, and the server picks up the new map without dropping its clients.

12.2 Exposing a tag

Tags are exposed one at a time. Open the **Tag Database** and look at its two Modbus columns:

Column	Shows
Modbus	The holding-register address as raw / Modicon (for example 100 / 40101), and for a BOOL a second line coil 50 / 00051 . A dash when the tag is not exposed.
Expose	A share icon, filled when the tag is exposed

Click a tag to select it, then in the Properties panel find the **Modbus Server** section:

- **Expose via Modbus** — tick to publish the tag. The IDE assigns the next free address at once. Untick and the panel reads *Hidden — not visible to the Modbus server*; the tag remembers its last address and gets it back if it is still free when you re-expose it.
- **HR** — the holding-register address. Type either the raw 0-based address (100) or the Modicon form (40101).
- **Coil** — BOOL tags only: the coil address, raw (50) or Modicon (00051).

Under each field the panel shows both forms of the address and the word *manual* once you have typed one yourself. Manual addresses are never moved by renumbering unless you ask.

If you type an address another tag already owns, the **Modbus HR address in use** (or **Modbus Coil address in use**) dialog names the owner and offers **Swap**: the two tags exchange addresses and both become manual. Two-word tags can only swap with two-word tags.

PROPERTIES

active_level
REAL %MR22

Data Type
REAL (4 bytes) ▼

Variable Class
VAR ▼

Initial Value
0.0

Description
Active sensor level (%)

Modbus Server
☒ Expose via Modbus

HR 2000
2000 / 42001

Process Variable
☒ Enabled

PV Slot (0-63)
31

Report on Change
☐ Disabled

History (on-controller, 60-day)
☐ Not kept

12.3 Address layout and data types

Tag type	Coil space	Holding-register space
BOOL	one coil	one register (0 or 1)
INT	—	one register
DINT	—	two consecutive registers
REAL	—	two consecutive registers, IEEE 754 single precision

A BOOL is published in both spaces, so a SCADA that only reads registers can still see it. Two-word values are stored **low word first** (word order CDAB, each word big-endian); select “word swap” or “CDAB” in the SCADA driver. A two-word value must be written with FC 16 in one request starting at its first register; FC 06 on either half is refused so a DINT or REAL can never be half-updated.

Addresses in the IDE are 0-based and are shown alongside the Modicon convention (holding register 0 is 40001, coil 0 is 00001). Field inputs (%I, %IW) are listed as read-only in the map and exports; everything else is read/write.

12.4 Auto-numbering

Select the project node in the Project Tree and open the **Modbus Server** section of its properties. **Auto-numbering mode** decides how **Renumber** lays out addresses:

Mode	Layout
Alphabetical	Tags in name order, packed from address 0 in both spaces
Group by data type (<i>default</i>)	Each type gets its own address range, tags in name order within it
Creation order	Tags in the order they were created, packed from 0

With *Group by data type* an **Address ranges per data type** table appears with a **start** and **size** for **Coil bool**, **HR bool**, **HR int** and **HR real** (defaults: coils and BOOL registers from 0, INT from 1000, REAL from 2000, 1000 addresses each). Ranges are useful when the SCADA engineer wants all the analog values in one block.

Click **Renumber all auto-assigned tags** to apply the mode. The **Renumber Modbus addresses?** dialog warns that this changes the address of every value the SCADA reads and offers **Keep manually-assigned addresses** (ticked by default). Changing the mode in the dropdown triggers the same confirmation (**Change auto-numbering mode?**); cancelling it leaves the mode unchanged. Tags you deleted leave gaps; the *Remove Unused Tags* dialog reminds you that a renumber closes them.

Warning. Renumbering after the SCADA has been configured breaks every address it reads. Do it before hand-off, or export a new map and re-import it on the SCADA side. {.warning}

12.5 The Modbus Map view

As soon as one tag is exposed, a **Modbus Map** node appears in the Project Tree under the project, next to Design Spec. Click it to open a read-only view built from the Tag Database as it stands — no build is needed and there is nothing to edit here.

The view has two sections, **Holding Registers** (FC 03 / 06 / 16 · base 40001) and **Coils** (FC 01 / 05 / 15 · base 1), each a table with the columns **Tag**, **Description**, **Type**, **IEC Address**, **Modbus (0-based)**, **Modicon**, **Width**, **Word Order** (CDAB for two-word values), **Access** (R or RW), **Units**, **Gain**, **Offset**, **Min** and **Max**. The last five come from the tag's process-variable settings and travel into the exports as scaling.

Toolbar, left to right: a **Show / hide columns** button (the choice is remembered per project), **Export...**, and a search box (*Search tag, description, address...*) with a row counter on the right. Hover a row for **Copy row to clipboard**, which copies every column tab-separated for pasting into a spreadsheet.

Export...

Search tag, description, ad...

261 row(s)

Holding Registers - FC 03 / 06 / 16 - base 40001 - 162 tag(s)

Tag	Description	Type	IEC Address	Modbus (0-based)	Modicon	Width	Word Order	Access	Units	Gain	Offset	Min	Max
alarm_high_level	High level alarm latched	BO...	%MX61	0	40001	1	—	RW				0.0	100...
alarm_io_comm	IO comm fault latched	BO...	%MX66	1	40002	1	—	RW				0.0	100...
alarm_p1_fts	P1 fail-to-start latched	BO...	%MX57	2	40003	1	—	RW				0.0	100...
alarm_p1_seal	P1 seal failure latched	BO...	%MX59	3	40004	1	—	RW				0.0	100...
alarm_p1_vfd_fault	P1 VFD fault latched	BO...	%MX55	4	40005	1	—	RW				0.0	100...
alarm_p2_fts	P2 fail-to-start latched	BO...	%MX58	5	40006	1	—	RW				0.0	100...
alarm_p2_seal	P2 seal failure latched	BO...	%MX60	6	40007	1	—	RW				0.0	100...
alarm_p2_vfd_fault	P2 VFD fault latched	BO...	%MX56	7	40008	1	—	RW				0.0	100...
alarm_pressure_fault	Pressure sensor fault latched	BO...	%MX63	8	40009	1	—	RW				0.0	100...
alarm_radar_fault	Radar sensor fault latched	BO...	%MX64	9	40010	1	—	RW				0.0	100...
alarm_reset_request	Operator alarm reset button	BO...	%MX67	10	40011	1	—	RW				0.0	100...
alarm_rod_fault	Rod sensor fault latched	BO...	%MX62	11	40012	1	—	RW				0.0	100...
alarm_sensor_disag...	Sensor disagreement latched	BO...	%MX65	12	40013	1	—	RW				0.0	100...
cmd_p1_run	P1 internal run command	BO...	%MX39	13	40014	1	—	RW				0.0	100...
cmd_p2_run	P2 internal run command	BO...	%MX40	14	40015	1	—	RW				0.0	100...

The project's **Design Spec** document also carries a generated *Modbus Registers* section with the same holding-register and coil tables, so the map is part of the printed design record.

12.6 Exporting the map for the SCADA engineer

Click **Export...** in the Modbus Map view. The **Export Modbus Map** dialog asks for:

- **Format** — one of the four below.
- **Output file** — with **Browse...**; the suggested name is `<project>_<format>.csv`.
- **Device name** — the OPC path prefix (Ignition) or the access name (AVEVA) the SCADA will use for this controller.
- **Ignition tag-path prefix** (Ignition only, default `Tags`).
- **Default scan rate (ms)** (KEPServerEX only, default 100).

Format	What it produces
Generic CSV	One row per register or coil: Tag, Description, Data Type, IEC Address, Register Type, Modbus (0-based), Modicon, Width (words), Word Order, Access, Units, Gain, Offset, Min, Max. Import it into anything, or keep it as the hand-off document.
Ignition CSV	Ignition tag-import format: Path, Name, Tag Type, Data Type, OPC Server, OPC Item Path ([device]HR40101 , HRF for REAL, HRI for DINT, C for coils), Documentation, Engineering Units, Read Only and linear scaling columns
KEPServerEX CSV	KEPServerEX tag import: Tag Name, six-digit Modicon Address (400101 , 000051), Data Type, Client Access, Scan Rate, Scaling, Eng Units, Description
AVEVA System Platform CSV	A DBLoad file with :IODiscrete and :IOReal sections, Group WhiskerPLC , AccessName = the device name, item names HRxxxxx / Cxxxxx

All four are UTF-8 with a byte-order mark, sorted holding registers first then coils by address. The last-used values of the dialog are remembered per project. The Output panel reports *Exported N Modbus map rows to*

Tip. Keep the exported CSV with the project's documentation and re-export it after every renumber. When the SCADA side reports a wrong value, the CSV is the record of what it should be reading. {*.tip*}

12.7 When the SCADA cannot connect

- The SCADA must reach the controller's address on TCP port 502; check firewalls between them.
- The map on the controller is the one from the last Build (F5). Exposing a tag in the IDE does nothing until the program is downloaded.

- Check the addressing convention in the SCADA driver: the IDE shows both 0-based and Modicon addresses, and KEPServerEX expects the six-digit form.
- For DINT and REAL values, set the driver's 32-bit word order to low-word-first (CDAB).

See *Troubleshooting* in the appendices for more.

13 Python on the Controller

Both controller families run Python, in different ways. A **Nexus.io Automation Controller** project carries an `app.py` that runs beside the ladder program, reads and writes the same tags, and can serve function calls from ladder. A **SmartController (Classic)** project is MicroPython: its files are edited in the IDE and synchronised to the device over USB serial. This chapter covers both, starting with the editor they share.

13.1 The Python editor

Click a Python file in the Project Tree and it opens in the Python editor: syntax highlighting, line numbers, and the editing behaviour you expect from a code editor.

- **Ctrl+F** opens the find bar (**Find...**, **Replace...**, Enter for next, Shift+Enter for previous, Esc to close). Search is case-insensitive plain text.
- **Tab** and **Shift+Tab** indent or outdent the selected lines by four spaces; with no selection, Tab inserts four spaces and Shift+Tab outdents the current line.
- **Enter** keeps the previous line's indent and adds a level after a line ending in `:`, `(`, `[` or `{`.
- **Ctrl+/** toggles a `#` comment on the line or selection.
- Pasted text has tab characters converted to four spaces, so mixed-indent errors cannot creep in from another editor.
- Edits are saved into the project automatically as you type; **Ctrl+S** saves the project.

```

app.py
Ctrl+F to find

1  """Lift-station wet-well generator (optional demo aid).
2
3  `level_sim` is the SIM sensor level, and by default it is a MAINTENANCE
4  INPUT: the operator sets it by hand from the slider on the Main screen
5  (bound readWrite) to exercise the station without a real transmitter. So
6  this app must not touch it unless it has been asked to - anything writing
7  `level_sim` on a timer overwrites the slider before the operator's value can
8  take effect.
9
10 Two switches gate it, both on the Settings page:
11
12     sp_pump_simulation    overall simulation mode
13     sp_level_sim_auto    sub-mode: generate the level here instead of by hand
14
15 Both must be ON. With the sub-switch OFF the app writes nothing at all and
16 the slider has sole control.
17
18 When generating, the model is a wet well with a varying influent and two
19 fixed-speed pumps:
20
21     level% += (inflow - pumps_running * PUMP_RATE) * dt
22
23 Influent swings on a slow cycle so the station walks its whole sequence in a
24 couple of minutes: at dry-weather flow the lead pump alone draws the well
25 down; at peak it cannot keep up, the level climbs to the lag setpoint and the
26 second pump joins. Peak influent stays below the two-pump capacity on purpose
27 - a high-level alarm every cycle is a nuisance in a demo, not a feature.
28 Raise INFLOW_MAX above 2 * PUMP_RATE to show the alarm and standby sequence.
  
```

13.2 Part 1 — Python on the Nexus.io controller

13.2.1 app.py and run(ctx)

Every Nexus.io controller project has a **Python** node in the Project Tree containing `app.py`. It cannot be deleted; **Add Python File** adds further modules (`helpers.py`, `utils/math.py`) that `app.py` can import. A new project's `app.py` is a template:

```

def run(ctx):
    ctx.log.info("App started")
    while not ctx.shutdown_requested:
        # Your application logic here
        ctx.wait(1.0)
    ctx.log.info("App stopped")
  
```

The controller's Python runtime calls `run(ctx)` once, on its own thread, alongside the ladder program. There is no fixed scan: your loop decides its own rate with `ctx.wait()`. When `run` returns, the application has finished; it is started again only when a new bundle arrives. An exception inside `run` is reported to the Python Log with its traceback and the application stops.

A ladder program and `app.py` run at the same time and share every tag. A Python-only project — no ladder logic at all — is equally valid; the ladder VM simply has nothing to do.

13.2.2 The ctx API

`ctx` is the controller's API. Tag access by name is the normal way to work; the arena calls are there for offset-addressed access when you need it.

Member	Description
<code>ctx.read_tag(name)</code>	Read a tag by its Tag Database name. Returns <code>bool</code> , <code>int</code> or <code>float</code> according to the tag's type. Raises <code>KeyError</code> for an unknown name.
<code>ctx.write_tag(name, value)</code>	Write a tag by name; the value is converted to the tag's type. Returns <code>True</code> on success. Raises <code>KeyError</code> for an unknown name and <code>ValueError</code> for a digital or analog input, which is read-only.
<code>ctx.tags</code>	The list of tag names available to this program.
<code>ctx.wait(seconds)</code>	Sleep for up to <code>seconds</code> (a float). Returns <code>True</code> as soon as the controller asks the application to stop — break out of your loop when it does — and <code>False</code> when the full time elapsed. Use it instead of <code>time.sleep()</code> , which cannot be interrupted.
<code>ctx.shutdown_requested</code>	<code>True</code> once a stop has been requested (service stop, a new deploy).
<code>ctx.log</code>	A standard Python logger: <code>ctx.log.info(...)</code> , <code>.warning(...)</code> , <code>.error(...)</code> . Lines appear in the IDE's Python Log panel and in the controller's log files.
<code>ctx.di_read(offset, count=1)</code> / <code>ctx.dq_read(...)</code>	Read digital inputs / outputs as a list of <code>bool</code> .
<code>ctx.dq_write(offset, values)</code>	Write digital outputs from a list of <code>bool</code> .
<code>ctx.ai_read(offset, count=1)</code> / <code>ctx.aq_read(...)</code>	Read analog inputs / outputs as a list of 16-bit <code>int</code> .
<code>ctx.aq_write(offset, values)</code>	Write analog outputs.

Member	Description
<code>ctx.mb_read(offset, count=1) /</code> <code>ctx.mb_write(offset, values)</code>	Memory bits (%MX) as <code>bool</code> .
<code>ctx.mw_read(offset, count=1) /</code> <code>ctx.mw_write(offset, values)</code>	16-bit memory words (%MW).
<code>ctx.md_read(offset, count=1) /</code> <code>ctx.md_write(offset, values)</code>	32-bit memory words (%MD).
<code>ctx.mr_read(offset, count=1) /</code> <code>ctx.mr_write(offset, values)</code>	Memory reals (%MF) as <code>float</code> .
<code>ctx.pv_read(slot) /</code> <code>ctx.pv_read_all() /</code> <code>ctx.pv_write(slot, value)</code>	Process-variable slots 0–63 (Cloud edition projects). <code>pv_read_all</code> returns 64 floats, NaN for unassigned slots.

`ctx.wait(seconds)` sleeps for the time you give it from the moment you call it; it does not subtract the time your loop body took. A loop that works for 40 ms and then waits 50 ms repeats every 90 ms. The runtime measures the time between successive `wait()` calls and streams it to the IDE's **Statistics** panel, so you can see your loop's real period.

A typical loop:

```
SCAN_S = 0.05 # 50 ms

def run(ctx):
    ctx.log.info("Lift station app started")
    while not ctx.shutdown_requested:
        level = ctx.read_tag("wet_well_level")
        if ctx.read_tag("mode_auto"):
            ctx.write_tag("call_lead", level >
                ctx.read_tag("sp_lead_start"))
        if ctx.wait(SCAN_S):
            break
    ctx.log.info("Lift station app stopped")
```

The standard library is available — `datetime` , `json` , `statistics` , `re` , `collections` — and extra modules you add to the project's Python node are importable by file name. There is no package installer: what you can import is the standard library plus your own files.

13.2.3 Deploying Python

Python is part of the normal build. With a controller connected, press **Build (F5)**; the IDE compiles and downloads the program to the connected controller. Before bundling, every Python file in the project is syntax-checked; an error stops the build with the file, line and message in the Output panel. The Python files, the tag map and any Python function blocks are packaged as one bundle inside the deploy. On the controller, the runtime notices the new bundle within a couple of seconds, stops the running `run(ctx)` (your loop sees `shutdown_requested`), replaces the files and starts the new application. Nothing needs restarting by hand.

13.2.4 Testing without a controller

Build & Emulate runs the same bundle on your PC. The controller's Python runtime starts beside the ladder emulator and runs your application unmodified — `run(ctx)`, the alert rules and any Python function blocks — against the emulated memory, so a value you set in the **IO Stimulus** tab is what `ctx.read_tag` returns and what `ctx.write_tag` writes shows in the Tag Monitor and on the HMI Preview. The Python Log panel connects to the emulated runtime automatically. See *Connecting, Deploying and Emulating*.

13.2.5 The Python Log

The **Python Log** tab at the bottom of the workspace streams the controller's application log live: every `ctx.log` line with its time and level, plus the runtime's own messages about starting and stopping the application. It connects automatically when the IDE connects to a Nexus.io controller, and to the emulated runtime during **Build & Emulate**.

- An exception in `run(ctx)` or in a Python function block appears as a red row with the exception and a **source · phase** line; click the row to expand the traceback. Unread errors are counted in a red badge in the panel header; click the badge to clear it.
- Header buttons: **Connect / Disconnect**, **Copy all log entries to clipboard**, **Clear Log**, and **Auto-scroll ON / OFF**. Any text in the list can be selected and copied.

Use `ctx.log` rather than `print()`: the log is what the panel shows.

```

PythonLog
INFO:10:14:00.777 INFO: Lift station app stopped
INFO:10:14:00.777 INFO: Lift station app stopped
INFO:10:15:02.123 INFO: Lift station app ready (level generator: Pump Simulation + Auto Level; OK data: on)
INFO:10:15:02.123 INFO: Lift station app ready (level generator: Pump Simulation + Auto Level; OK data: on)
INFO:10:15:02.124 INFO: Level generator on - taking over at 49.8 S
INFO:10:15:02.124 INFO: Level generator on - taking over at 49.8 S
  
```

13.2.6 Python function blocks (PYFB)

A ladder network can call a Python function and use its result. On the Python side, decorate a function in `app.py` :

```

from whisker import pyfb

@pyfb("calc_dose")
def calc_dose(ctx, flow_gpm, target_ppm):
    return flow_gpm * target_ppm * 0.012

def run(ctx):
    # optional when the file only serves function blocks
    while not ctx.shutdown_requested:
        if ctx.wait(1.0):
            break
  
```

The `whisker` module is provided by the runtime; you do not add a file for it. The function receives `ctx` — the same object as `run(ctx)` , so it can read other tags and log — followed by the block's wired arguments in pin order, already converted to `int` , `float` or `bool` . Return an `int` , `float` or `bool` ; it is written to the block's OUT pin, typed by the tag bound there. Returning `None` leaves OUT unchanged.

On the ladder side, the **PYFB** block's Properties panel takes the **Instance Name**, the **Python function** (**@pyfb name in app.py**), arguments **A1** to **A4** (tags or numbers; only wired arguments are passed), a **Timeout TO** in milliseconds (blank = 1000) and the output tags **Result (OUT)**, **Done (DONE)**, **Error (ERR)** and **Error code (ECODE)**. Each rising edge of the network sends one request;

DONE goes true when the reply arrives and stays true until the next request. ECODE reports 1 for a timeout, 2 when the function raised, 3 when no function of that name is decorated, 5 for a program/runtime mismatch. On any error OUT keeps its last good value, so a dosing loop never slams to zero because Python hiccuped. An exception's traceback appears in the Python Log.

Rules that keep this reliable:

- Keep the function quick. Functions run one at a time on a dispatcher thread, so a slow one delays the others; a round trip measures about 20 ms at a 10 ms scan. Long work belongs in `run(ctx)` with a tag handshake.
- Do not call `ctx.wait()` inside a decorated function.
- Arguments and results are numbers only — no strings or lists.
- A project with PYFB blocks but no decorated function of that name still builds; the block reports ECODE 3 on its first call and the runtime logs the missing name once.

The block's pin table is in *Ladder Instruction Reference*.

13.3 Part 2 — SmartController (Classic)

The SmartController (Classic) runs MicroPython. Its logic is Python, edited in the IDE and copied to the device over **USB serial**. A Classic project shows only what applies to it: the **File System** tree, **Design Spec**, **Tags** (Tag Database and Tag Cross Reference) and, when tags are exposed, **Modbus Map**. There is no ladder editor, HMI Designer, Alerts editor or Setup node.

13.3.1 Project files

A new Classic project has three files, none of which can be deleted or renamed:

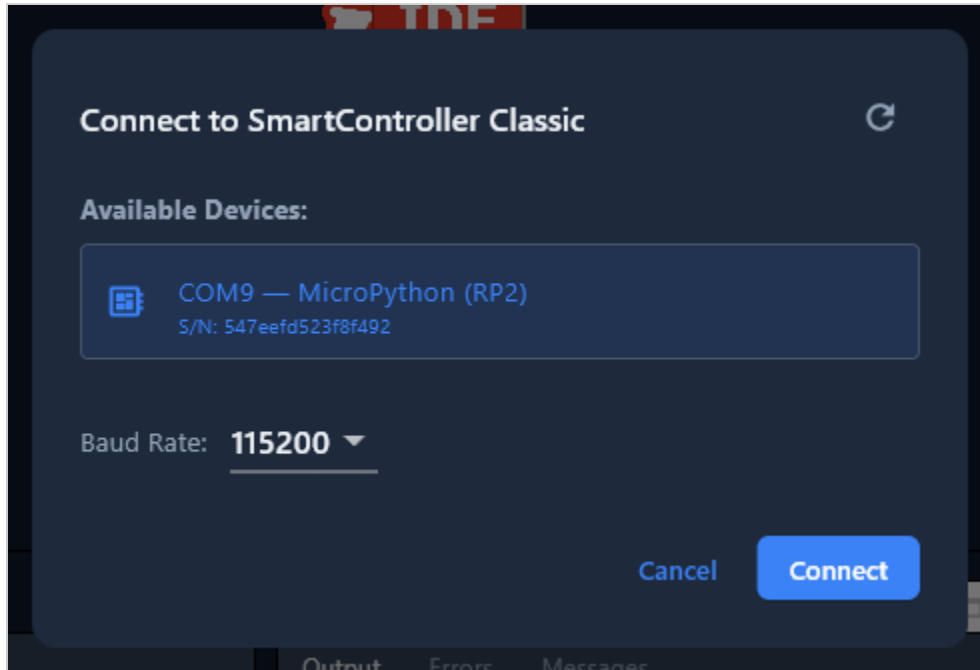
File	Purpose
<code>app.py</code>	Your application: <code>setup(...)</code> runs once at start, <code>getAppStatus()</code> returns two status lines for the display, and <code>async def loop(...)</code> is the main loop, run as a MicroPython asyncio task. Near the top, <code>appName</code> and <code>appVersion</code> name the application: the controller shows them on its screen, and the IDE uses them as the name and version of the packages it installs (below). Change <code>appVersion</code> whenever you release a change.
<code>app_vars.py</code>	Generated by the IDE from the Tag Database — a constant per process variable so <code>app.py</code> can read and write values by name. Read-only in the editor; regenerated when tags change.
<code>config.json</code>	The controller's runtime configuration — update period, time zone, logging, Modbus settings and the process variable list — generated from the project's Classic settings and its Tag Database. Do not edit it by hand; it is regenerated before every Sync and Build.

Add further modules with **New File** in the File System tree.

A Classic project must have **at least one process variable** (a tag with a PV slot in the Tag Database). Without one the controller's display task stops and the controller restarts every 30 seconds with a blank screen, so **Sync** and **Build** refuse such a project and say why.

13.3.2 Connecting over serial

Plug the controller into a USB port and click **Connect** in the toolbar. The **Connect to SmartController Classic** dialog lists the serial devices it finds under **Available Devices**; a Classic shows as *MicroPython (RP2)* with its serial number and is selected for you (or the port you used last time). Set the **Baud Rate** if needed — 115200 is the default — and click **Connect**. The toolbar then reads *Connected (COMn)*.



Connect to SmartController Classic

Connecting **stops the program** on the controller so the IDE can read and write its files, and keeps the controller at the `>>>` prompt. The controller has a hardware watchdog that would otherwise restart it a few seconds after its program stops; while the IDE holds it, the IDE keeps the watchdog satisfied.

Disconnect restarts the controller so its program runs again. If the controller restarts or is unplugged while connected, the toolbar reports the connection lost — click **Connect** again.

Warning. Disconnect before closing the IDE. If the IDE exits while a serial connection is open, Windows keeps the port locked until the USB driver releases it, which can take a minute or more, and the controller stays stopped until it is restarted or power-cycled. {.warning}

13.3.3 The File System tree

At the top of the Project Tree, **File System** has two sections:

- **Local** — the files in your project. Buttons: **Sync all local files to target**, **New File**, **New Folder**, **Delete selected**. Double-click a file to edit it.
- **Target** — the application files on the controller, shown while connected: the files its manifest lists for the installed application and configuration, and any other Python file in the root of its file system. The controller's own firmware is not shown. **Refresh** re-reads it. Double-click a target file to

view it read-only in a tab titled with its name in square brackets. A folder shown as **app (legacy)** is left over from an older installation; it hides the installed application and the next Sync removes it.

13.3.4 Sync

There is no per-file upload. **Sync** installs the project the same way the factory load tool does:

- every project file goes to the **root** of the controller's file system;
- files the previous application installed that are not in your project are removed — the controller's firmware, `ident.json` and `main.py` are never touched;
- a legacy `/app` folder is removed;
- when `config.json` changes, the saved process variable values (`pv.dat`) are cleared, because they are restored by position and would otherwise land on the wrong variables;
- each uploaded file is checked on the controller, and the controller's **manifest** (`/manifest.json` , its record of what is installed) gets an **app** package (your code) and a **config** package (`config.json`), both named and versioned from `appName` and `appVersion` , next to the firmware's own package.

The **Sync to Target** dialog lists the new and modified files, everything it will delete and the package name and version, and **Sync** carries it out. Files are compared by their SHA-256 hash, so an unchanged project reports *Target is up to date — no changes*. After a Sync, restart the controller (below) to run the new program.

13.3.5 The console

The **SmartController Classic Console** tab at the bottom of the workspace is an interactive MicroPython REPL over the same serial port. Type at the `>>>` prompt; Up and Down recall previous commands. Its buttons are **Interrupt** (**Ctrl+C**) — Ctrl+C in the input does the same — **Restart controller** and **Clear**.

Restart controller restarts the controller and follows it: the connection is picked up again as soon as the controller is back, without stopping it, so its start-up messages and anything the program prints appear in the console while it runs.

Interrupt stops the running program and returns to the `>>>` prompt.

```
SmartController Classic Console
SmartController Classic Console

import os
>>> os.listdir()
['sd', 'app.py', 'app_vars.py', 'chemtank.py', 'config.json', 'crypto', 'diagnostics', 'drivers_exp', 'ident.json', 'main.py', 'manifest.json', 'pv.dat', 'smartcontroller', 'whiskerblue']
>>>
SD card mounted
SD card filesystem scanned
SD card filesystem healthy
[2021-01-01 00:00:02] [uimanager] [info] starting up
[2021-01-01 00:00:02] [uimanager] [info] Free RAM is 393328 before display setup
[2021-01-01 00:00:03] [uimanager] [info] Free RAM is 401328 after display setup
[2021-01-01 00:00:03] [uimanager] [info] Free RAM is 400432 after loading UI
[2021-01-01 00:00:03] [bg95] [info] loading bg95
Setting picroRTC 10/4/2026 16:11:44
well_monitor: started
well_monitor: cycle 25, level 45.0 %, up 11 s
setting time 10/4/26 21:12:13
RTC values: 13 12 21 0 4 10 26
Setting picroRTC 10/4/2026 16:12:13
well_monitor: cycle 50, level 50.0 %, up 23 s
well_monitor: cycle 75, level 55.0 %, up 29 s
well_monitor: cycle 100, level 60.0 %, up 35 s

>>>
```

Classic console

For a more detailed log in the console, set the project's **Logger Profile** to `debug` (see *Project Properties Reference*), Sync and restart.

13.3.6 What is different from the Nexus.io controller

- No compile step: MicroPython runs the `.py` files directly. **Build** for a Classic project needs no device connection: it asks for a folder and writes the two packages Sync installs — `dist_app/` (the code) and `dist_config/` (`config.json`), each with its `manifest.json` — the form the factory load tool installs from.
- The `ctx` API, `run(ctx)` and Python function blocks belong to the Nexus.io controller. A Classic `app.py` uses `app_vars.py` and the `asyncio` loop shown above.
- The **Python Log** panel is for the Nexus.io controller's application log; on a Classic, use the console.
- Cloud publishing of a Classic's process variables is configured in the Tag Database (**Cloud Type**, **Units**, **R/W** and the display settings) and pushed with the project's cloud configuration; see *Project Properties Reference*.

14 Connecting, Deploying and Emulating

This chapter takes you from a saved project to a Nexus.io Automation Controller running it: finding the controller on the network, connecting to it in whatever security posture it is in, letting the IDE reconcile your open project with what the controller already holds, deploying with **Build**, and running the program on your PC with **Build & Emulate** when there is no controller at hand.

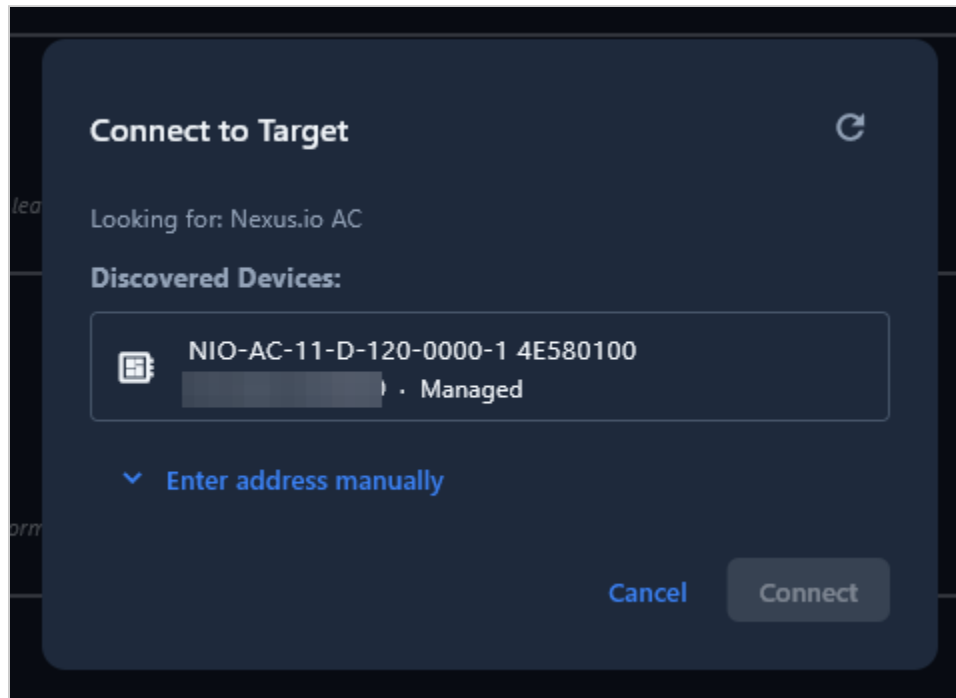
Nothing here needs a Whisker.io account except connecting to a cloud-enrolled controller, which needs the account that enrolled it (see *Security, Provisioning and Enrolment*).

14.1 Before you connect

- The project builds cleanly or you are prepared to fix what the Output panel reports.
- The controller is powered and on the same network as your PC, or reachable from it.
- You know how the controller was commissioned: **standalone** (as shipped: plain connections, unsigned deploys, or a unit a site chose to leave that way), **managed** (enrolled into a cloud account or provisioned into a site CA) or **awaiting enrolment** (a locked-down configuration some sites order: nothing but enrolment until it is managed). The Connect dialog shows which.

14.2 Discovering controllers

Click **Connect** on the toolbar. The **Connect to Target** dialog opens and scans for about five seconds. Every Nexus.io controller announces itself on the local network as an mDNS service of type `_ladder-vm._tcp`; the IDE lists what it hears. If mDNS finds nothing, the IDE probes the addresses of your PC's own subnet directly, so a controller on the same network segment still appears even when multicast is filtered.



Each row shows the controller's model and serial, its address and port, and its posture:

Row label	Meaning	How the IDE connects
Standalone	The shipping posture: no identity, plain connections from any IDE, unsigned deploys. Also a controller a site chose to leave that way.	Plain connection
Managed	Enrolled into a cloud account or provisioned into a site. Accepts mutually authenticated TLS only and signed deploys only.	Mutual TLS with your identity

Row label	Meaning	How the IDE connects
Awaiting enrolment	A locked-down configuration some sites order: the controller accepts a connection, status queries and the enrolment or site-identity upload, nothing else, until it is made managed. Its data port is closed. A decommissioned controller returns here too.	Plain connection

Click a row, then **Connect**. The refresh icon in the title bar (tooltip **Scan for devices**) runs the scan again. When a project is open, the dialog says *Looking for:* followed by the project's target type and lists only matching controllers.

Note. A PC on a VPN, or on a different subnet from the controller, cannot see mDNS announcements and the subnet probe scans the wrong subnet. Use **Enter address manually**.

14.2.1 Connecting by address

Expand **Enter address manually** and fill in **Target Address** (the controller's address or host name) and **Port** (9000 unless you have changed it). Because no discovery record tells the IDE the controller's posture, it tries a TLS connection with your identity first and falls back to a plain connection if that is refused. Connecting by address is the normal path over a VPN or a routed network. For a standalone controller — which is how a new one arrives — that is the whole procedure: connect, then deploy. A managed controller reached by address still needs the identity that enrolled or provisioned it (see *Security, Provisioning and Enrolment*).

14.3 Which identity the IDE presents

A managed controller only accepts a certificate issued by the authority it trusts. The IDE picks its identity in this order:

1. **Your account identity** when you are logged in to Whisker.io. The IDE requested this certificate from the account's certificate authority at login; a controller enrolled into the same account accepts it.

2. **A site identity** imported from a site administrator's package. The project's **Site identity** setting (Project Properties, Security) selects which one; with a single site imported the choice is automatic.
3. **The local keystore**, used only for bench units provisioned from this PC.

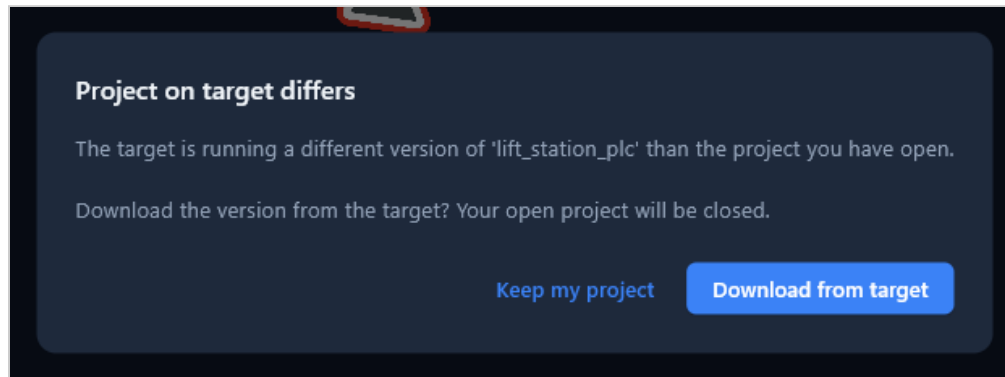
If the handshake is refused, the Output panel and the connection message name the reason; the usual causes are listed in *Troubleshooting*.

14.4 What happens after connect

The toolbar shows a green **Connected** badge and the status bar reads **Connected**, with **(Running)** or **(Debug)** appended once the controller reports its mode. The IDE also opens the controller's Python log stream (the **Python Log** panel) if the program has one.

Straight after connecting, the IDE asks the controller which project it holds and compares it with the project you have open. This is the **connect-time reconcile**:

You have open	Controller holds	What the IDE does
Nothing	A project with stored source	Offers Download from target . Decline, and the connection is cancelled.
Nothing	No stored project	Connects and says there is nothing to download. Monitoring stays off.
A project	The same project, same version	Connects silently. Full monitoring once you have deployed in this session.
A project	The same project, a different version	Asks whether to download the controller's version. Keep yours, and monitoring is limited.
A project	A different project	Asks whether to download it. Keep yours, and monitoring is off until you deploy.
A project	No stored project (older deploy, or source storage off)	Connects and says it cannot confirm what is running. Monitoring stays off until you deploy.



The dialog is titled **Project on target differs** and has two buttons: **Keep my project** and **Download from target**. Downloading closes the open project (you are asked to save unsaved edits first) and opens the controller's copy; that copy has no file location yet, so the first **Save** asks where to put it.

A controller can only offer a download if the project was deployed with **Store editable source on target** switched on (Project Properties, Deployment section). With it off, the controller keeps nothing an engineer could pull back, which some customers require for intellectual-property reasons; the IDE then cannot confirm what the controller is running either.

14.5 Deploying with Build

For a Nexus.io project the **Build (F5)** button is enabled only while a controller is connected, because Build compiles the project and, on success, downloads it to the connected controller in one action. The Output panel calls the pair "Build + Download". There is no separate Upload or Download button.

Press **Build (F5)**. In order, the IDE:

1. Runs the **undefined-tag sweep**. If the ladder references a name the tag database does not hold, the dialog *N tags are used but not defined* lists them with a suggested type; **Create All** creates them and the build continues, **Cancel** stops it.
2. Compiles the project and prints the compiler's output. Errors end the build with **Build failed**.
3. Applies the **deploy-overwrite guard**. If the controller holds a different project, the dialog **Replace project on target?** appears; **Replace** continues, **Cancel** leaves the controller untouched.
4. If the project requires cloud PIN login at the panel, requests a fresh device credential from the cloud for this controller's serial (see *Security, Provisioning and Enrolment*).

5. Stops the controller's program.
6. Sends the program. To a **managed** controller it packages the build into a **signed bundle** and stages it; the controller verifies the signature against the authority it trusts and only then installs the files. To a **standalone** controller it sends the build artifacts one by one, unsigned.
7. Reloads the program, starts it running and turns monitoring on. The last line is **Program downloaded and running**.

```

Output  Errors  Messages
[0] [W007] coilSet on 'alarm_p2_reset' with no coilSet - tag can never be set (networks: 6, 28) 18.03.22
[0] [W007] coilSet on 'alarm_high_level' with no coilSet - tag can never be set (networks: 7) 18.03.22
[0] [W007] coilSet on 'alarm_red_fault' with no coilSet - tag can never be set (networks: 8) 18.03.22
[0] [W007] coilSet on 'alarm_pressure_fault' with no coilSet - tag can never be set (networks: 9) 18.03.22
[0] [W007] coilSet on 'alarm_radar_fault' with no coilSet - tag can never be set (networks: 10) 18.03.22
[0] [W007] coilSet on 'alarm_sensor_disagree' with no coilSet - tag can never be set (networks: 11) 18.03.22
[0] [W007] coilSet on 'alarm_io_comm' with no coilSet - tag can never be set (networks: 12) 18.03.22
[0] [W007] coilSet on 'okc_reset_all' with no coilSet - tag can never be set (networks: 13) 18.03.22
[0] [W007] coilSet on 'okc_p1_reset_all' with no coilSet - tag can never be set (networks: 17) 18.03.22
[0] [W007] coilSet on 'okc_p2_reset_all' with no coilSet - tag can never be set (networks: 27) 18.03.22
[0] [W009] coilSet on non-alarm tag 'okc_p1_comm_fault' - should this be a plain coil? (networks: 19) 18.03.22
[0] [W009] coilSet on non-alarm tag 'okc_p1_comm_fault' - should this be a plain coil? (networks: 20) 18.03.22
[0] [W009] coilSet on non-alarm tag 'okc_p1_comm_fault' - should this be a plain coil? (networks: 21) 18.03.22
[0] [W009] coilSet on non-alarm tag 'okc_p2_comm_fault' - should this be a plain coil? (networks: 29) 18.03.22
[0] [W009] coilSet on non-alarm tag 'okc_p2_comm_fault' - should this be a plain coil? (networks: 30) 18.03.22
[0] [W009] coilSet on non-alarm tag 'okc_p2_comm_fault' - should this be a plain coil? (networks: 31) 18.03.22
[0] [W009] coilSet on non-alarm tag 'okc_ww_highhigh' - should this be a plain coil? (networks: 39) 18.03.22
[0] [W009] coilSet on non-alarm tag 'okc_ww_high' - should this be a plain coil? (networks: 41) 18.03.22
[0] [W009] coilSet on non-alarm tag 'okc_ww_low' - should this be a plain coil? (networks: 43) 18.03.22
[0] [W009] coilSet on non-alarm tag 'okc_ww_lowlow' - should this be a plain coil? (networks: 45) 18.03.22
[0] [W011] Variable 'okc_supply_dc' is read but never written by ladder (networks: 36, 37) 18.03.22
[0] [W011] Variable 'okc_ww_hi_hi_on' is read but never written by ladder (networks: 39) 18.03.22
[0] [W011] Variable 'okc_ww_hi_hi_off' is read but never written by ladder (networks: 40) 18.03.22
[0] [W011] Variable 'okc_ww_hi_off' is read but never written by ladder (networks: 42) 18.03.22
[0] [W011] Variable 'okc_ww_lo_on' is read but never written by ladder (networks: 43) 18.03.22
[0] [W011] Variable 'okc_ww_lo_off' is read but never written by ladder (networks: 44) 18.03.22
[0] [W011] Variable 'okc_ww_lo_on' is read but never written by ladder (networks: 45) 18.03.22
[0] [W011] Variable 'okc_ww_lo_off' is read but never written by ladder (networks: 46) 18.03.22
[0] [W011] Variable 'level_sim' is read but never written by ladder (networks: 36) 18.03.22
[0] Build succeeded: 15 files generated 18.03.22
[0] Stopping VM... 18.03.22
[0] Signing bundle for managed controller (local keystore identity)... 18.03.22
[0] Signed by (SHA256Digest7bcb37f151...) 18.03.22
[0] Uploading signed bundle to target... 18.03.22
[0] Reloading program... 18.03.22
[0] Starting VM... 18.03.22
[0] Starting monitoring... 18.03.22
[0] Monitor state enabled=true, tcpOk=true 18.03.22
[0] Program downloaded and running 18.03.22
  
```

14.5.1 The signed bundle

A managed controller never accepts a plain program file. The IDE packs the build directory into one archive, signs it with your certificate, and sends the archive, the signature, a manifest naming the signer and the payload hash, your certificate and the authority's chain. The controller checks that the payload matches the manifest, that the signature is valid, that your certificate chains to its trusted authority and is not on its revocation list. If any check fails the deploy is refused with the reason in the Output panel and **the previous program keeps running**.

The first signed deploy to a controller that has no trust anchor yet binds it to your authority; the Output panel says so. From then on only bundles signed under that authority are accepted.

14.5.2 Retained values

Variables marked retained keep their values across a deploy: the controller migrates them by name, so a setpoint an operator entered at the panel survives a program update. Renaming a retained variable makes it a new one. See *Tag Database*.

14.6 Run, Stop and Debug

Three toolbar buttons control the program on a connected controller: **Run Mode (F7)**, **Stop Mode (Shift+F7)** and **Debug Mode (F9)**. The keys do the same as the buttons and, like them, work only while a Nexus.io controller is connected.

- **Run Mode** — the runtime scans every task, reads inputs and drives outputs.
- **Stop Mode** — the runtime halts. Outputs hold their last state.
- **Debug Mode** — Run Mode with live monitoring switched on (subject to the gate below). The ladder editor shades energised elements and shows live values.

The active mode is highlighted on the toolbar and named in the status bar.

Warning. Stop Mode freezes outputs where they are. Confirm with operations before stopping a controller that drives live equipment. {.warning}

14.7 Why monitoring is off until you deploy

Live monitoring reads the controller's memory and labels it with the open project's tag names. If the controller is running anything else, every value shown would be plausible and wrong. So the IDE fails closed: the monitoring button (tooltip **Enable Monitoring**) is disabled until this session has built and downloaded the open project to this controller, and it disables again the moment you edit the project. The disabled button's tooltip states the reason, for example *Monitoring unavailable — This project has not been downloaded to the target yet. Use Build + Download to enable monitoring.*

The connect-time reconcile applies the same rule from the other side: keeping your project when the controller holds a different one leaves monitoring off; keeping it when the controller holds a different version of the same project gives limited monitoring, where names align but values may be unreliable until you deploy.

14.8 Build & Emulate

Build & Emulate (tooltip *Build & Emulate (run on this PC, no target needed)*) compiles the project and runs it in an emulator on your PC. It needs no controller and no account. It does need a Python 3.9 or newer interpreter on your PC's PATH; the emulator is a Python program installed with the IDE.

When you press it, the IDE:

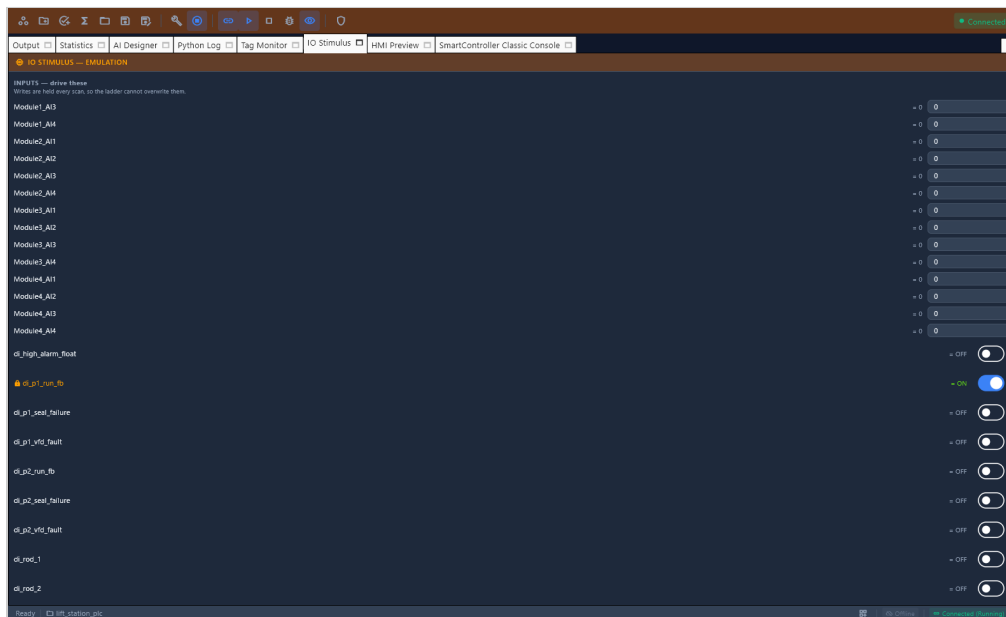
1. Runs the undefined-tag sweep and compiles, exactly as Build does.
2. Disconnects from any controller — the IDE keeps one connection, and two would leave you unsure which one the ladder editor is showing.
3. Starts the emulator on the build it just produced and connects to it on this PC.
4. Turns the toolbar amber for as long as emulation is running (the IO Stimulus tab's header turns amber too), and prints *Emulation running — the IDE is connected to the emulator on this PC, not to a controller.*

The emulator runs the ladder program and the controller's memory areas on your PC. The connect-time reconcile and the monitoring gate are satisfied automatically, because the emulator runs exactly the build it was handed and accepts no other. Press **Enable Monitoring** (or **Debug Mode**) and the IDE behaves as it does against a controller:

- **Ladder editor** — energised elements and wires are shaded, live values appear above the elements.
- **Tag Monitor** — every addressed tag with its live value; the header counts the emulator's scans.
- **IO Stimulus** tab — the panel for driving the program by hand. Its header reads *IO STIMULUS — EMULATION*. Under *INPUTS — drive these* every tag bound to a physical input is listed: digital inputs as switches, analogue inputs as a number box or, when the channel has an IODF range, a slider labelled in engineering units with the raw count beside it. A value you set is held every scan so the program cannot overwrite it; a lock icon marks the tag as forced. Under *OUTPUTS — what the ladder produced* the physical outputs are shown read-only. A project whose tags are all memory (no physical I/O) shows nothing here; drive those from the Properties panel's write and force controls instead.
- **HMI Preview** tab — renders the project's panel screens (screen drop-down, zoom) with live values and marks itself *LIVE — touch enabled* while monitoring is on, so push buttons, selector switches and numeric inputs on the preview write into the emulated program exactly as the panel would. Without monitoring it shows design-time defaults and ignores presses.
- **Properties panel** — the **ONLINE — WRITE / FORCE** section for the selected network works as on a controller (see *Monitoring and Debugging*).

- **Python application** — when the project has Python (an `app.py`, alerts, or Python function blocks), the controller's own Python runtime starts on your PC beside the emulator and runs it unmodified: `run(ctx)` is called, `ctx.read_tag` and `ctx.write_tag` go to the emulated memory, alerts are evaluated, and a **PYFB** block in the ladder gets its answer from your `@pyfb` function. The Output panel confirms it with *Python application running under emulation*, and the **Python Log** panel connects to it exactly as it does to a controller, so `ctx.log` lines, tracebacks and the loop statistics appear there. Its own start-up messages are prefixed `[py]`. If the runtime cannot start, the Output panel says *Python application not running*: with the reason and the ladder emulates without it.

Physical devices, the historian and the cloud service are not run by the emulator; the program — ladder, Python and alerts — is what you are testing.



The same button becomes **Stop Emulation** while the emulator runs. Stopping it ends the Python runtime (your `run(ctx)` loop sees `shutdown_requested`) and the emulator, drops the connection to them and restores the toolbar and your previous target address. If a start fails, the last lines of the emulator's own output appear in the Output panel prefixed `[emu]`; a port already in use usually means an emulator from an earlier session is still running.

14.9 Disconnecting

Click the toolbar **Disconnect** button (the same button as **Connect**, highlighted while connected). Monitoring stops, the Python log stream closes and the status bar returns to **Not Connected**. Any forces you left on the controller stay active until the controller's runtime restarts — check the ladder editor's amber **FORCES** chip before you leave (see *Monitoring and Debugging*).

Opening or creating another project also drops the connection, so the toolbar never reports a connection that belongs to a different project.

14.10 A typical session

1. Open the application and select the project.
2. Click **Connect**, pick the controller, connect. A standalone controller connects plainly; a managed one uses your account or site identity. If the reconcile dialog appears, decide whether the controller's copy or yours is the one to keep.
3. Edit tags, ladder, HMI and alerts.
4. Press **Build (F5)**. Create any undefined tags the sweep reports. Confirm **Replace** if the controller held a different project.
5. Watch the Output panel through *Program downloaded and running*. Monitoring is now on.
6. Use **Debug Mode**, the ladder editor's live shading, the Tag Monitor and the Properties panel's write and force controls to check the logic.
7. Edit again; monitoring switches off. Press **Build (F5)** again to deploy and re-enable it.
8. Clear any forces, then **Disconnect**.

If no controller is available, replace steps 2 and 4 with **Build & Emulate** and work against the emulator; the rest of the loop is the same.

15 Monitoring and Debugging

Once a program is running on a Nexus.io Automation Controller (or in the emulator), the IDE becomes a live window onto it: the ladder editor shades what is energised, the Tag Monitor lists every addressed tag's value, the Properties panel lets you write and force values, and the Output and Statistics panels tell you what the runtime is doing. This chapter covers each tool and ends with diagnostic recipes for the problems that come up most.

Nothing in this chapter needs a Whisker.io account.

15.1 The monitoring gate

Live values are only ever shown for a program this IDE session has deployed. The rule:

Monitoring is available only after **Build (F5)** has compiled and downloaded the open project to the connected controller in this session, and it switches off again on any edit.

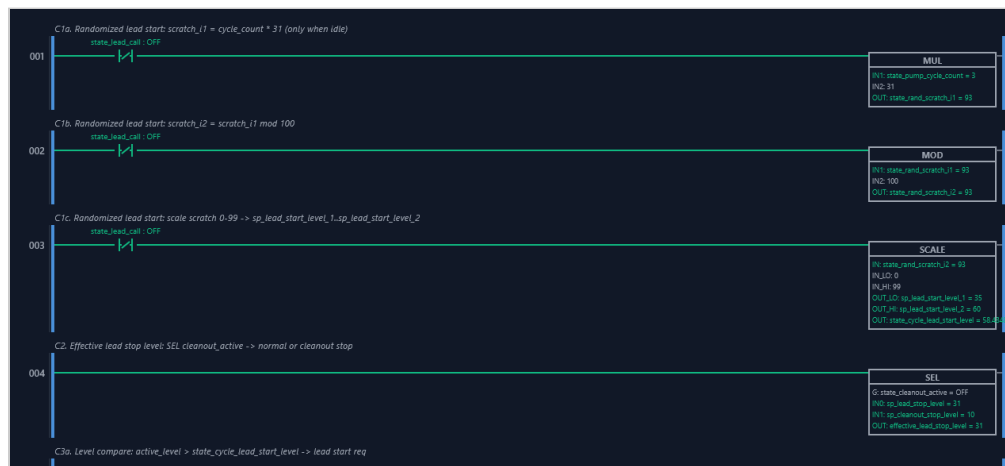
The reason is safety, not tidiness: monitoring reads the controller's raw memory and labels it with the open project's tag names. Against a different program every value would look plausible and be wrong. The toolbar's monitoring button (tooltip **Enable Monitoring / Disable Monitoring**) is disabled while the gate is closed, and its tooltip names the reason:

Tooltip reason	What to do
<i>Not connected to a target.</i>	Connect.
<i>No project open.</i>	Open the project the controller runs.
<i>This project has not been downloaded to the target yet. Use Build + Download to enable monitoring.</i>	Press Build (F5) .
<i>A different project was downloaded to this target. Download the open project to enable monitoring.</i>	Press Build (F5) with the right project open.
<i>The target is running a different project. Download this project to enable monitoring.</i>	You kept your project at the connect-time reconcile; deploy it.

Build (F5) turns monitoring on automatically when it finishes. **Debug Mode** turns it on as well, when the gate allows. Under **Build & Emulate** the gate is satisfied by construction, because the emulator runs exactly the build it was given.

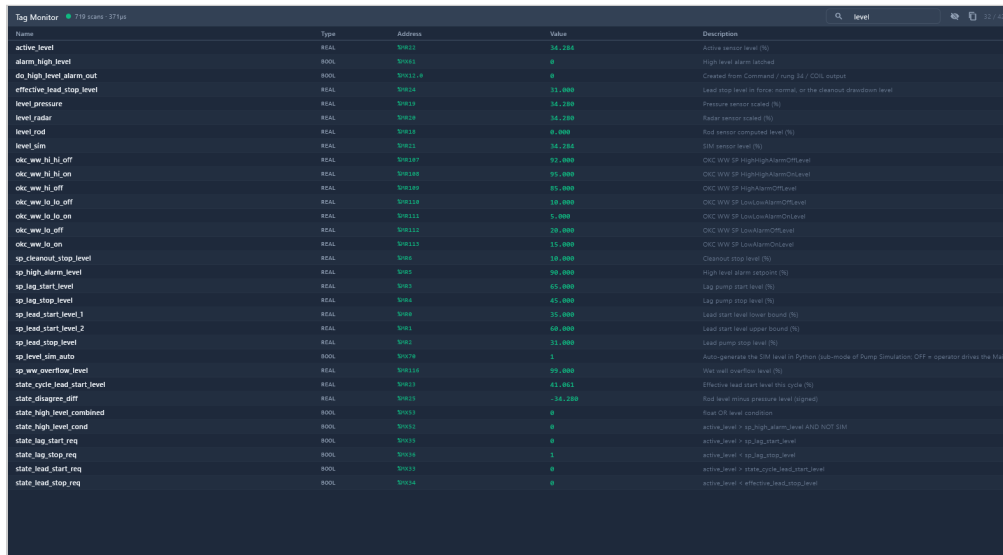
15.2 Where live values appear

- **Ladder editor.** Contacts and coils that are energised change colour; wires carry the power flow across each network; the label above every element shows the tag's current value. Compare segments (CMP) are evaluated against live values.
- **Tag Monitor panel.** A table of every addressed tag and its value.
- **Properties panel.** With a network selected, an amber **ONLINE** — **WRITE / FORCE** section lists the tags on that network with their live values and write/force controls.
- **IO Stimulus panel.** Every tag bound to a physical input as a switch, number box or slider you can drive (values are held every scan), and every physical output read-only. Its header reads *IO STIMULUS — TARGET* on a controller and *IO STIMULUS — EMULATION* under Build & Emulate; see *Connecting, Deploying and Emulating* for the controls.
- **HMI Preview panel.** The project's panel screens rendered with live values; *LIVE* — *touch enabled* while monitoring is on, so presses on the preview write to the controller as the panel would.
- **Statistics panel.** Scan timing, program size and connection state.



15.3 The Tag Monitor panel

The **Tag Monitor** tab sits in the bottom panel group next to Output. It lists every tag that has an address; local temporaries without an address have no memory slot to read and are not shown.



Name	Type	Address	Value	Description
active_level	REAL	\$W023	34.284	Active sensor level (%)
alarm_high_level	BOOL	\$W003	0	High level alarm detected
do_high_level_alarm_out	BOOL	\$W022.0	0	Created from Command / Pump 34 / CO2 output
effective_lead_stop_level	REAL	\$W024	31.000	Lead stop level in fence, normal, or the overflow shutdown level
level_pressure	REAL	\$W020	34.280	Pressure sensor output (%)
level_radar	REAL	\$W020	34.280	Radar sensor output (%)
level_rtd	REAL	\$W020	0.000	Rtd sensor computed level (%)
level_sim	REAL	\$W023	34.284	Sim sensor level (%)
clk_ww_hl_hl_off	REAL	\$W0207	92.000	CLK WW OF HighAlarmOffLevel
clk_ww_hl_hl_on	REAL	\$W0208	95.000	CLK WW OF HighAlarmOnLevel
clk_ww_hl_off	REAL	\$W0209	85.000	CLK WW OF HighAlarmOffLevel
clk_ww_hl_hl_off	REAL	\$W0208	10.000	CLK WW OF LowAlarmOffLevel
clk_ww_hl_hl_on	REAL	\$W0209	9.000	CLK WW OF LowAlarmOnLevel
clk_ww_hl_off	REAL	\$W0210	70.000	CLK WW OF LowAlarmOffLevel
clk_ww_hl_on	REAL	\$W0210	15.000	CLK WW OF LowAlarmOnLevel
sp_cherout_stop_level	REAL	\$W04	10.000	Cherout stop level (%)
sp_high_alarm_level	REAL	\$W04	90.000	High level alarm request (%)
sp_lag_start_level	REAL	\$W04	65.000	Lag pump start level (%)
sp_lag_stop_level	REAL	\$W04	45.000	Lag pump stop level (%)
sp_lead_start_level.1	REAL	\$W04	35.000	Lead start level upper bound (%)
sp_lead_start_level.2	REAL	\$W04	20.000	Lead start level lower bound (%)
sp_lead_stop_level	REAL	\$W04	25.000	Lead pump stop level (%)
sp_level_sim_auto	BOOL	\$W078	1	Auto-generate the SIM level in Python (sub-mode of Pump Simulation: OFF = operator drives the tag)
sp_ww_overflow_level	REAL	\$W0104	99.000	Wat wast overflow level (%)
state_cycle_lead_start_level	REAL	\$W023	41.063	Effective lead start level this cycle (%)
state_diagnose_diff	REAL	\$W023	-34.280	Rtd level minus pressure level (signed)
state_high_level_combined	BOOL	\$W003	0	Rtd OR level condition
state_high_level_cond	BOOL	\$W003	0	active_level > sp_high_alarm_level AND NOT sim
state_lag_start_req	BOOL	\$W020	0	active_level > sp_lag_start_level
state_lag_stop_req	BOOL	\$W020	1	active_level > sp_high_alarm_level
state_lead_start_req	BOOL	\$W020	0	active_level > state_cycle_lead_start_level
state_lead_stop_req	BOOL	\$W024	0	active_level < effective_lead_stop_level

The header shows the connection dot, the scan counter and scan time (for example $1240 \text{ scans} \cdot 812\mu\text{s}$), and three controls:

- **Filter** — type part of a name, address or description to narrow the list.
- **Eye toggle** — system-generated tags are hidden by default (tooltip *System-generated tags hidden — click to show*). Click to include them; this is where the per-device **Comm OK**, **Comm Age** and **Error Type** tags live.
- **Copy** — tooltip *Copy visible tags to clipboard (TSV)*. Copies the visible rows as tab-separated text (Name, Type, Address, Value, Description) ready to paste into a spreadsheet or a support e-mail.

The right-hand count shows *visible / total* when a filter or the eye toggle is hiding rows.

Columns are **Name**, **Type**, **Address**, **Value** and **Description**. Values are **1 / 0** for BOOL, integers for INT and DINT, three decimals for REAL. Before monitoring starts the Value column shows a dash and the panel says *Not connected — no live values*; once monitoring runs, a **?** means the tag's address could not be read from the data the controller sends.

15.4 Debug Mode shading in the ladder editor

Debug Mode (toolbar; tooltip *Debug Mode (F9)*) puts the controller in Run and switches monitoring on. In the ladder editor:

- an energised contact, coil or output pin is drawn in the active colour;
- a normally closed contact is drawn active when its tag is false, since that is when it conducts;
- wires are coloured from the left rail as far as power flows, so the first de-energised segment on a network is visible at a glance;
- the label above each contact and coil shows the tag's live value.

Monitoring refreshes about five times a second. The picture is a sampled view, not a scan-by-scan trace; a pulse shorter than the refresh interval may never be seen shaded.

15.5 Writing and forcing values

Select a network in the ladder editor while connected with monitoring on. The Properties panel shows the **ONLINE — WRITE / FORCE** section with one row per addressed tag on that network.

PROPERTIES

⚡ ONLINE — WRITE / FORCE
 1 forced

Tags on this rung. "= value" is what the TARGET holds (needs monitoring on); the box below it is what you are about to write. Switch / ± apply immediately; typed values apply on Enter or with the pencil. Pin = force (held every scan), open-lock = release.

🔒 state_cleanout_... = OFF
 📌 🔒

☐ OFF

sp_lead_stop_lev... = 31
 ✎ 📌 🔒

value to write
 31
 REAL

sp_cleanout_sto... = 10
 ✎ 📌 🔒

value to write
 10
 REAL

effective_lead_st... = 31
 ✎ 📌 🔒

value to write
 31
 REAL

other tag... Add

⚠️ Clear All Forces (1)

Select (SEL)

Instance Name

sel_eff_stop

Input Tags

G (Selector, BOOL)

state_cleanout_active ✕

 BOOL @ %MX41

IN0 (When G=0)

sp_lead_stop_level ✕

 REAL @ %MR2

IN1 (When G=1)

sp_cleanout_stop_level ✕

 REAL @ %MR6

Output Tags

Each row shows = *value* (what the controller holds now) and a control for the value you are about to write: a switch for BOOL, plus and minus for integers, a text box for anything typed.

- **Write** (the pencil, or Enter in the text box) writes the value once. The program can overwrite it on the next scan, which is what you want for a setpoint or a one-shot test.
- **Force** (the pin; tooltip *Force the typed value (held every scan)*) writes the value every scan so the program cannot overwrite it. Use it to hold an input high or pin an analogue value while you watch the logic.
- **Release** (the open lock; tooltip *Release force*) removes the force on that tag.
- **other tag...** with **Add** brings any other addressed tag in the project into the section.
- **Clear All Forces (N)** releases every force on the controller.

Function-block instances (a timer or counter name) are not tags and cannot be written; force their pins instead.

While any force is active the ladder editor's header shows an amber chip, **FORCES (N)** — **click to clear**, and the Statistics and Properties panels count them. Forces live on the controller: they survive the IDE disconnecting and reconnecting, and are cleared when the controller's runtime restarts.

Warning. A forced output on a controller driving live equipment stays forced after you disconnect. Clear every force before you leave, and never force outputs on a running plant without operations' agreement. {.warning}

15.6 The Output panel

The **Output** tab collects everything the IDE does on your behalf. Each line has an icon for its kind — information, warning, error or success — and a timestamp.

Copy All copies the log with [INFO], [WARN], [ERROR] and [OK] prefixes;

Clear empties it; right-click a line to copy just that one.

What you will see there:

- **Build lines** from the compiler, then the deploy sequence (*Stopping VM..., Signing bundle for managed controller... or Uploading program to target..., Reloading program..., Starting VM..., Program downloaded and running*).

- **Connect lines**: the connect-time reconcile's outcome, and on a site-managed controller whether the revocation list was pushed.
- **Enrolment and security lines** from Provision New Device, Approve Re-Enrollment and the Security dialog.
- **Emulator output** prefixed `[emu]` when a Build & Emulate start fails.
- **Warnings** the deploy carried, such as an optional artifact an older controller did not accept.

15.7 The Statistics panel

The **Statistics** tab is a dashboard of cards that rearrange to the width of the panel:

- **Controller** — connected or not, the mode (**RUN**, **STOP** or **DEBUG**), the controller's name and address, uptime and whether monitoring is on.
- **Scan time** — a gauge of the current scan time, scaled to the slowest scan seen and coloured green, amber or red as it nears it, with the scan **Rate**, **Min** and **Max** below.
- **Scan trend** — the last two minutes or so of scan-time samples, with the average.
- **Program on controller** — **Code size**, **Tasks**, **FB instances** and **Scans**.
- **Python app** — when the controller runs a Python application, a gauge of its loop time with rate, average, maximum and loop count.
- **Project** — the open project's tags (total and by type), programs, networks and HMI screens.

A rising **Max** with a steady scan time points at an occasional slow path — a device that times out, or a Python function block doing too much. The trend shows when it happens.

15.8 The Python Log panel

When the connected controller runs a Python application, the **Python Log** tab streams its `ctx.log()` output live. The stream opens automatically on connect. For the SmartController (Classic) the **SmartController Classic Console** tab is the interactive REPL; see *Python on the Controller*.

15.9 Historian and trends

Values you need over hours or days belong in the on-controller historian, not in a Tag Monitor you have to keep open. Tick **Hist** for the tag in the Tag Database, deploy, and read it back with a trend chart on the panel or in WhiskerHMI. See *Historian and Trends*.

15.10 Diagnostic recipes

15.10.1 Start with the system tags

Every device in I/O Configuration gets three tags the IDE maintains for you:

`<device> Comm OK`, `<device> Comm Age` and `<device> Error Type`. They are the first thing to check when a value looks wrong. Show them in the Tag Monitor with the eye toggle, or filter on `Comm`.

- **Comm OK** false with a rising **Comm Age** — the controller is not hearing from that device. Check the address, unit ID, cabling and that the device is powered.
- **Comm OK** true but the value is stale — the device answers, but the register or channel you bound is not the one changing. Check the point binding in I/O Configuration.
- **Error Type** non-zero — the device answered with an exception. Compare the register map with the device's documentation.

15.10.2 “The program is deployed but nothing happens”

1. Check the status bar reads **Connected (Running)**. If it says only **Connected**, press **Run Mode**.
2. Turn on monitoring (or press **Debug Mode**). If the button is disabled, read its tooltip — you probably edited since the last deploy. Press **Build (F5)**.
3. Look at the first network that should fire. Follow the wire colour from the left rail; the first grey segment is the contact that is false. Check that tag's value and where it comes from.
4. If the input never changes, check the device's system tags as above.

15.10.3 “A value is wrong by a scale factor”

The device's raw counts are not the engineering value. Check the IODF range chosen for the channel in I/O Configuration and the SCALE block's parameters in the ladder.

15.10.4 “An alert will not clear”

Latched alerts stay active until acknowledged at the panel even after the condition clears; see *Alerts Editor*. For a non-latched alert, watch its condition tags in the Tag Monitor: the alert follows them exactly.

15.10.5 “Monitoring stops by itself”

You edited the project. The gate closes on any edit and reopens after the next **Build (F5)**.

15.10.6 “The connection drops”

Read the Output panel’s last lines. A controller that restarts into managed mode after enrolment or site provisioning drops the connection on purpose — reconnect from **Connect**. A drop during a deploy’s reload step is followed by an automatic reconnect. Repeated drops with nothing in the log point at the network: a second IDE connected to the same controller, a PC that keeps switching between Wi-Fi and cable, or a VPN that resets idle sessions.

15.10.7 “The panel HMI shows old values”

The panel reads the controller’s memory directly and does not depend on the IDE’s monitoring. If the panel is stale while the Tag Monitor is live, the HMI screen is bound to a different tag than the ladder writes. Check the widget’s tag in the HMI designer.

See *Troubleshooting* for connection, enrolment and deploy failures.

16 Security, Provisioning and Enrolment

A Nexus.io Automation Controller leaves the factory **standalone**: it has no identity and accepts plain connections and unsigned deploys from any IDE that can reach it. A controller connected to the cloud must be managed, and enrolment makes it so. A controller that is not connected to the cloud is managed only if you choose to make it so, and the IDE, the controller and a WhiskerHMI panel all respect that choice. This chapter explains the three security postures a controller can be in, the two paths — a Whisker.io account (Path A), or standalone with the optional site certificate authority you run yourself (Path B) — the **Security** dialog that manages both, the local keystore for bench units, PIN login at the panel, and what a managed controller refuses.

What needs what:

Feature	Needs
Cloud enrolment, Rotate, Decommission, Approve Re-Enrollment, the Account section	Cloud edition, logged in to a Whisker.io account
Site CA (make a site, issue identities, Provision unit (site), CRL push)	Either edition; no account (in the Cloud edition, logged in or Continue Offline)
Local keystore for bench units	Either edition; no account
Cloud PIN login at the panel	Cloud edition, logged in, and PIN codes enabled for your organisation in Whisker.io

The **Security** dialog opens from the toolbar shield button (tooltip **Security**) in both editions. The Cloud edition shows it whether or not you are logged in; the Offline edition shows the same dialog without the Account section, since there is no account to show.

16.1 The three postures

Posture	Discovery shows	Accepts	Deploys
Standalone	<i>Standalone</i>	Plain connections from any IDE; a WhiskerHMI panel connects plainly	Unsigned
Managed	<i>Managed</i>	Mutually authenticated TLS only, from a certificate issued by the one authority it trusts and not on its revocation list — an IDE identity, or a WhiskerHMI panel identity	Signed bundles only, verified before install
Awaiting enrolment	<i>Awaiting enrolment</i>	A plain connection; status and security queries; the enrolment bootstrap (Path A) or a site identity (Path B). Its data port is closed.	None

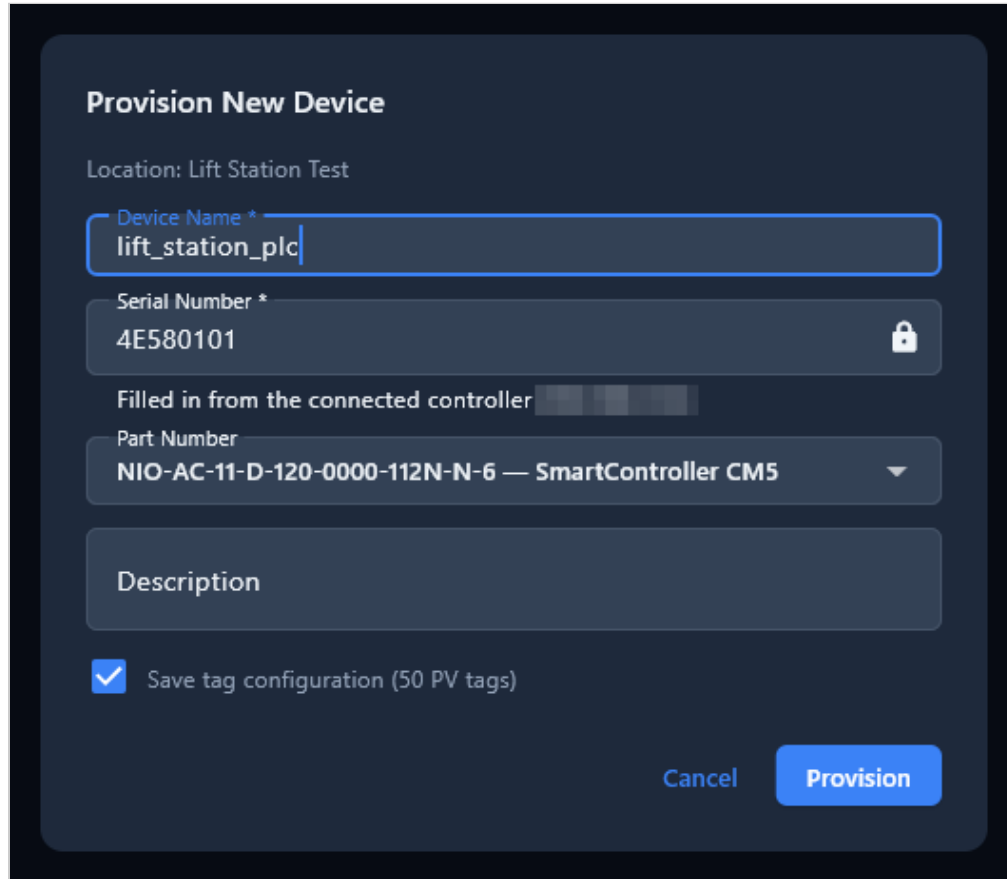
Every controller shipped to a customer arrives **standalone** and stays there until you decide otherwise. Enrolling it into a cloud account (Path A) or provisioning it into a site (the optional step of Path B) makes it **managed**; it then trusts exactly one authority: the account's, or the site's. **Awaiting enrolment** is not how a delivered controller starts. It is a locked-down configuration a site can order for units that must be managed before they do anything, and the posture a decommissioned controller returns to; in it the controller accepts nothing until it has been enrolled or provisioned.

16.2 Path A — cloud provisioning and enrolment

Path A binds the controller to your Whisker.io account's certificate authority. You need the Cloud edition, logged in, and a location in the Cloud Explorer to put the device under. At login the IDE requested its own user certificate from the account's authority, so nothing has to be set up in a keystore first.

16.2.1 Provision New Device

1. Click **Connect** and connect to the controller while it shows **Standalone** (or **Awaiting enrolment**, on a unit configured that way).
2. In the **Cloud** panel (the tab beside Project), right-click the location and choose **Provision New Device**.

A dark-themed dialog box titled "Provision New Device". It contains a "Location" field with the value "Lift Station Test". Below this are three input fields: "Device Name *" with the value "lift_station_plc", "Serial Number *" with the value "4E580101" and a lock icon, and "Part Number" with a dropdown menu showing "NIO-AC-11-D-120-0000-112N-N-6 — SmartController CM5". There is also a "Description" field. At the bottom, there is a checkbox labeled "Save tag configuration (50 PV tags)" which is checked. Two buttons, "Cancel" and "Provision", are at the bottom right.

1. Fill in **Device Name** (pre-filled from the project name) and, if you like, **Description**. **Serial Number** is filled in from the connected controller and locked; the lock icon (tooltip *Edit the serial by hand (not recommended)*) is only for provisioning a unit that is not connected. **Part Number** is NIO-AC-11-D-120-0000-112N-N-6 for a Nexus.io controller. Tick **Save tag configuration** to create the device's cloud tag template from the project's process variables at the same time.
2. Click **Provision**. The status line walks through *Checking for duplicates*, *Creating tag template*, *Provisioning device*, *Generating bootstrap nonce*, *Sending bootstrap.json to the controller...* and *Bootstrap delivered — waiting for the controller to enrol....*

What happens on the controller: the IDE writes a small bootstrap file over the connection it already has. The controller's enrolment service reads it, contacts the cloud with the one-time nonce, receives a certificate issued by the account's authority, installs it with the authority's chain and revocation list, and restarts its services in managed mode. Enrolment usually completes within a few seconds; the IDE polls for up to two minutes.

16.2.2 The reconnect moment

Restarting into managed mode closes the plain connection the IDE was using. The dialog reports *Controller enrolled ... It restarted into managed mode — reconnect from the Connect dialog*. Click **Connect** again; the row now says **Managed** and the connection is mutual TLS with your account identity. From here on every deploy to this controller is signed.

16.2.3 The one-time nonce dialog

If the IDE was not connected to the controller when you clicked **Provision**, or the controller refused the bootstrap, it cannot deliver the nonce itself. The **Device Bootstrap Nonce** dialog then shows the nonce once, with **Copy nonce** and **I've recorded it**. You do not need the nonce in normal use: connect to the controller (its row still shows the posture it had before) and choose **Approve Re-Enrollment** on the device in the Cloud panel. The IDE mints a fresh nonce, delivers it over the connection and waits for the controller to enrol, exactly as Provision New Device does.

16.2.4 Device actions in the Cloud panel

Right-click an enrolled device:

- **Rotate Device Cert** — revokes the current certificate and opens a five-minute window in which the controller may re-enrol with its old identity. An online controller does so on its next revocation-list check, within fifteen minutes, and comes back with a new certificate without a visit. If it is offline past the window, use Approve Re-Enrollment.
- **Decommission Device** — revokes the certificate and disables the device in the cloud. The controller drops its identity, closes its data port and waits in the awaiting-enrolment posture until an operator approves re-enrolment. Use it for a unit leaving service, changing hands or suspected compromised.
- **Approve Re-Enrollment** — mints a new nonce for a decommissioned device, or for one whose enrolment never completed, and delivers it if the IDE is connected to that controller.

Each action first shows the device's current enrolment state and fingerprint and asks you to confirm. The same three actions, under the same names, are in the row menu of the Security dialog's controllers table.

16.2.5 Revocation and renewal

The account's **certificate revocation list (CRL)** is the list of certificates the authority no longer accepts — rotated or decommissioned controllers, re-issued or withdrawn IDE certificates. Every enrolled controller fetches it from the cloud every fifteen minutes and consults it at every connection and every signed deploy. The IDE refreshes its own copy daily at login.

Certificate **renewal is automatic**. A cloud-issued device certificate is valid for 30 days; the controller's renewal service asks the cloud for a new one once fewer than 7 days remain and restarts its services with it. Nothing has to be scheduled, and a controller that is offline for the whole week simply renews when it is next connected, provided its certificate has not yet expired; past expiry, use **Approve Re-Enrollment**. The IDE's own certificate is re-issued at login when fewer than 30 days remain.

16.3 The Security dialog — Account section

Security

Account 579 — cloud certificate authority

The one PKI controllers trust. Units enrol against it; this IDE signs and connects with a certificate it issues.

Your IDE certificate (mTLS + deploy signing)

Re-issue

Issued 2026-09-29 · C

issued by D6 Labs Inc. Cloud

SHA256:76fc1f02ddcc411c0f21862121e3b13e0bbb6b38d...

Valid

Expires 2028-09-28

Account certificate authority chain

Issuing CA

D6 Labs Inc. Cloud

issued by Whisker Root CA (D6 Labs Inc.)

SHA256:1225adfffb4fc15a543b9d176d2275537d68a4f7b2...

Valid

Expires 2031-09-28

Root CA

Whisker Root CA (D6 Labs Inc.)

SHA256:ad4cf3ed63552531ee22ae336fcfb4e9f4da8f7be...

Valid

Expires 2036-09-26

Revocation list (CRL)

Refresh

Fetchd 2026-10-03, next update 2026-10-10 — refreshed daily on login. Enrolled controllers pull the same list every 15 minutes.

Controllers enrolled in this account

Serial	Device / location	State	Fingerprint	Enrolled
4E580101	lift_station_plc lift_station_test	Enrolled	SHA256:3cf50544dcf8bc1cb24... 2026-09-29	

Logged in, the first section is headed **Account N — cloud certificate authority** and holds:

- **Your IDE certificate (mTLS + deploy signing)** — the certificate the cloud issued you, with subject, issuer, fingerprint, status and expiry. **Re-issue** requests a new one; controllers accept it at once because they trust the authority, not the individual certificate.
- **Account certificate authority chain** — the **Issuing CA** and **Root CA** cards.
- **Revocation list (CRL)** — when it was fetched and when the next update is due; **Refresh** fetches it now.
- **Controllers enrolled in this account** — a table with **Serial**, **Device / location**, **State**, **Fingerprint** and **Enrolled** date. State is one of **Enrolled**, **Awaiting**, **Rotating**, **Decommissioned**, **Not enrolled** or **Error**; hover for an explanation. The row menu offers **Rotate Device Cert** and **Decommission Device** for an enrolled controller, **Approve Re-Enrollment** otherwise.
- **Panels (WhiskerHMI PC identities issued by this account)** — the panel identities this IDE has issued from the account's authority, with **Station**,

fingerprint, expiry, status and a **Revoke** icon per row, and the **Issue panel identity...** button (below).

Not logged in, the section says so and the rest of the dialog still works.

16.3.1 Panels — identities for WhiskerHMI PCs

A WhiskerHMI panel that connects securely to an enrolled controller must present a certificate issued by the account's authority; the build of the panel ships only the account's CA chain and revocation list. Each panel PC therefore needs its own **panel identity**, issued here and imported on that PC. Click **Issue panel identity...**, enter the **Panel PC name** (the station name becomes `HMI-<NAME>`), the **Validity (days)** and a **Panel passphrase** (confirmed), click **Issue...** and save the `.whisker-panel` package. The cloud issues the certificate; the IDE packages it with the account chain and revocation list. Copy the file to the panel PC and have the operator import it from the panel's **Panel ▶ Import panel identity...** menu with the passphrase, sent separately. **Revoke** on a panel's row puts its certificate on the account revocation list; enrolled controllers pull the list every fifteen minutes and refuse the panel from then on. The full procedure, including the messages the panel shows, is in *The WhiskerHMI PC Application*.

16.4 Path B — standalone, and the optional site CA

Path B is for plants without a cloud account. By default it needs nothing from this chapter: the controller stays standalone, you connect by discovery or by address and deploy, and that is the whole procedure. Management is your choice. A plant with several engineers that wants per-engineer identities, signed deploys and revocation sets up a **site CA** and gets the same managed posture as Path A: one certificate authority for the site, every engineer's IDE holding a certificate it issued, every controller trusting its chain, one revocation list. The difference from Path A is where the authority lives: on a PC the customer controls, not in the cloud. The rest of this section describes the site CA.

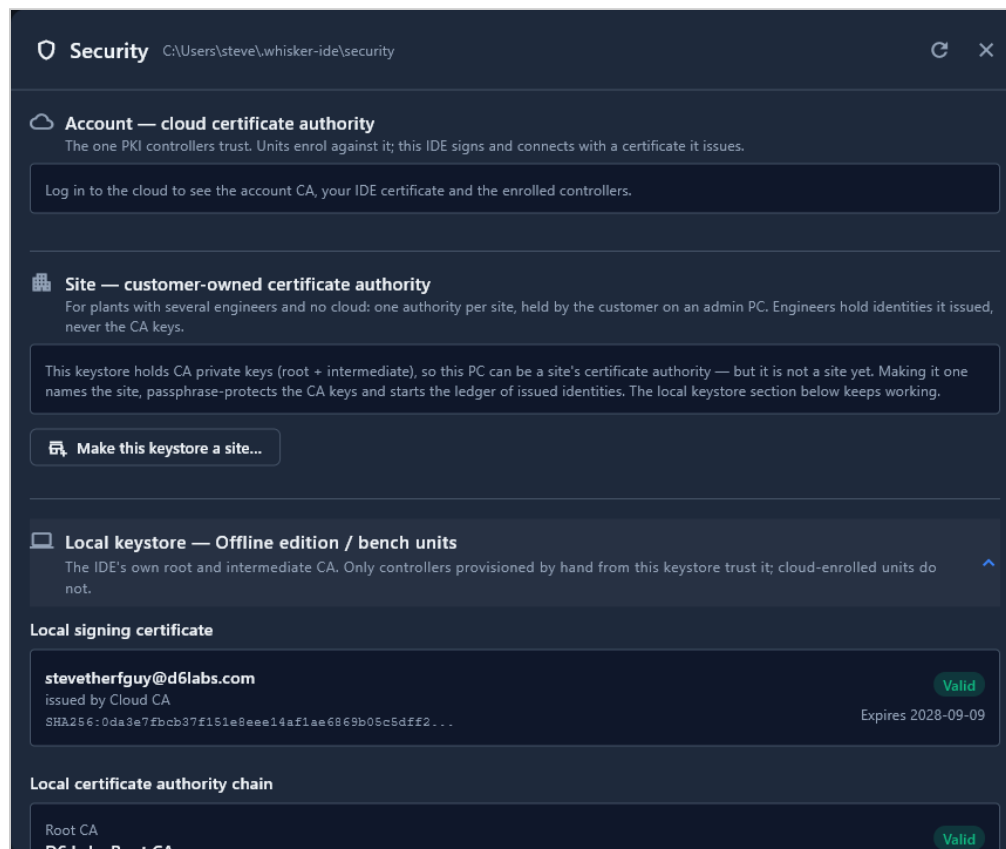
Warning. The customer owns the site CA. D6 keeps no copy and no escrow. Back the admin PC's keystore folder up to an encrypted USB drive and keep it in a safe: losing it loses the site, and every controller would have to be decommissioned and provisioned again. The Security dialog shows the folder's location in its title bar. {warning}

16.4.1 Who is the administrator

What makes a PC the site's administrator is possession of the authority's private keys in its keystore, protected by a **CA passphrase** that the IDE asks for on every issue, revoke and unit provisioning and never stores. There is no login or licence involved. Copying the keystore folder to another PC makes that PC an administrator too, so treat the folder as you would the keys to the plant. The administrator's PC is also an ordinary engineer PC: its own signing certificate comes from the same authority.

16.4.2 Make this keystore a site

On the PC that will hold the authority, open **Security** and find the **Site** section. If the keystore has no certificate authority yet, run the local keystore wizard first (below). Then click **Make this keystore a site...**, enter a **Site name** and a **CA passphrase** (at least eight characters, confirmed) and click **Create site**. The authority's keys are encrypted with the passphrase and a ledger of issued identities starts. The section heading becomes **Site name — this PC is the certificate authority**.



16.4.3 Issue engineer identity

Click **Issue engineer identity...** Enter the engineer's **e-mail**, the **Validity (days)** (365 by default), an **Identity passphrase** for the engineer (confirmed) and your **CA passphrase**, then click **Issue...** and choose where to save the package. The result is a `.whisker-identity` file named after the e-mail and the site, holding the site's chain and current revocation list plus the engineer's certificate and private key, the key encrypted with the identity passphrase.

The file and the passphrase must travel separately: e-mail or a share for the file, a phone call or in person for the passphrase. The file alone is useless without it. The **Engineers** table records every identity issued, with fingerprint, expiry and status.

16.4.4 Import site identity on the engineer's PC

On each engineer's PC, open **Security**, and in the Site section click **Import site identity...**, pick the `.whisker-identity` file and enter the passphrase. The dialog reports the site, the identity and the revocation list date. A card for the site now shows **Your identity**, its expiry and **CRL from** the list's date. Managed connections and signed deploys use this identity from now on. With more than one site imported, set **Site identity** in each project's properties (Security section) to the site the project belongs to; **Automatic** works when a single site is imported.

16.4.5 Provision unit (site)

The administrator binds a controller to the site:

1. Click **Connect** and connect to the controller while it shows **Standalone** (or **Awaiting enrolment**, on a unit configured that way). A managed controller cannot be re-provisioned; revoke and decommission it first.
2. In the Site section click **Provision unit (site)...** The confirmation names the controller's serial and the site; click **Provision** and enter the CA passphrase.
3. The IDE issues a device certificate for that serial, sends it with the site's chain and revocation list, and the controller verifies the serial, the key and the chain before installing. A controller already bound to another authority refuses.
4. The controller restarts into managed mode and the connection drops. Reconnect; the row now says **Managed** and the connection uses your site identity.

The **Units** table lists every controller the site has provisioned. A site-issued unit certificate is valid for 365 days and is **not** renewed automatically — there is no cloud for the unit to ask — so the administrator re-provisions the unit before it expires (the table shows each certificate's expiry). Engineer and panel identities issued by the site are also 365 days by default; the Issue dialogs accept 1 to 3650.

16.4.6 Panels — identities for WhiskerHMI PCs

A WhiskerHMI panel that connects securely to a site-managed controller must present a certificate issued by the site's authority; the build of the panel ships only the site's CA chain and revocation list. The **Panels (WhiskerHMI PC identities issued by this site)** sub-section, below the Units table, issues one identity per panel PC. Click **Issue panel identity...**, enter the **Panel PC name** (the station name becomes `HMI-<NAME>`), the **Validity (days)**, a **Panel passphrase** (confirmed) and your **CA passphrase**, click **Issue...** and save the `.whisker-panel` package. The identity is recorded in the ledger and listed in the table with **Station**, fingerprint, expiry and status. Copy the file to the panel PC and have the operator import it from the panel's **Panel > Import panel identity...** menu with the passphrase, sent separately. Revoking a panel works like revoking an engineer (next section): the list is regenerated at once and reaches the controllers with the next engineer's connect or **Push CRL to unit**. The full procedure, including the messages the panel shows, is in *The WhiskerHMI PC Application*.

16.4.7 Revoke, the CRL and the seven-day warning

Click the **Revoke** icon on an engineer's row (or a unit's, or a panel's) and confirm with the CA passphrase. The certificate goes on the site's revocation list, which is regenerated at once. A site has no cloud to distribute the list, so it travels with the engineers: **every IDE that connects to a managed controller with a site identity compares its list with the controller's and pushes the newer one** before doing anything else. The Output panel logs the result. A revoked engineer can no longer connect, so cannot push; any other engineer's next visit closes the gap. A controller nobody connects to keeps accepting the revoked certificate until someone does — walk the plant with a laptop after a revocation that matters. **Push CRL to unit** in the Site section sends the list to the connected controller on demand.

To get the new list to the engineers, the administrator clicks **Export site trust package...** (CA passphrase required) and distributes the resulting `.whisker-site` file, which holds only public material. Each engineer imports it with **Update**

trust package.... When an engineer's copy of the list is more than seven days old the site card and the Output panel warn: *your revocation list is N days old — update the site trust package.*

Revoking a unit's identity is followed by provisioning it again with a new one; revoking a panel's, by issuing a new identity and importing it on the PC if it stays in service.

16.5 Local keystore — Offline edition and bench units

The last section of the Security dialog, **Local keystore — Offline edition / bench units**, is the IDE's own certificate authority on this PC. It exists for two reasons: it is what a site CA is built on, and it lets you provision a bench unit by hand for development. **Set up local certificates...** runs a wizard in five steps — *Welcome, Root certificate authority, Intermediate certificate authority, Your signing certificate, Confirm and create*. Once it has run, the section shows your local signing certificate, the chain, and a **Bench device certificates (issued here)** table with **Issue bench device cert...** (enter the serial) and a **Revoke** icon per row.

A controller enrolled into a cloud account or provisioned into a site does not trust anything issued here. The local keystore is never the way to reach a customer's controller, and a WhiskerHMI build for such a controller takes nothing from it: the panel PC imports an identity issued by the account or the site instead (the Panels sub-sections above).

16.6 PIN login at the panel

Operators can be required to log in at the panel with their Whisker.io user and PIN before they can act on the HMI. It is a project setting, not a controller setting:

1. Your organisation enables PIN codes for HMI access in the Whisker.io Control Panel (Account Info). Until then the project's **Cloud PIN Login** row explains why it is unavailable.
2. In Project Properties, switch **Cloud PIN Login** on (the row reads *Required at the panel*). Set the **Session Timeout (s)** after which an idle session re-locks and the **Offline Fallback** behaviour. The **Roles at the panel** table maps each action (commands, setpoints, acknowledging and deleting alarms) to a role from your account.

3. Deploy. Every deploy mints a fresh device credential for the connected controller's serial and sends it with the program, which is why the controller must be registered to your account. Without a Whisker.io login the deploy stops with *This project requires a PIN at the panel, which needs a Whisker.io login to mint the device credential*. Operator login at the panel is a cloud feature: for a controller on Path B, switch **Cloud PIN Login** off. The project's Settings **PIN** still guards the panel's settings screen.
4. For a **WhiskerHMI PC**, the credential travels in the PC's panel identity package instead: with the setting on, **Issue panel identity...** in the Account section's **Panels** sub-section mints a station credential for `HMI-<NAME>` and puts it in the package the operator imports on that PC (see *The WhiskerHMI PC Application*). The installer never carries it, because one installer serves every PC and a credential belongs to one. The **Panels** table marks such rows `[PIN]`, and revoking the panel revokes its credential too. A site-managed PC has no cloud account and therefore no PIN login.

At the panel the operator picks their name from the list of users who hold the login role and have a PIN, then enters the PIN. Five wrong PINs lock that user for five minutes, doubling on each further cycle up to thirty; the panel says *This PIN is locked for N minutes. Other users can still log in*. Every attempt is audited. A controller that loses its cloud connection accepts, for 24 hours, the users who logged in while it was online. The **Lock Button** widget and the status-bar lock icon end a session; see *HMI Designer* for the widgets.

16.7 Signed deploys and what a controller refuses

A **managed** controller installs a program only from a signed bundle whose signature verifies against its trusted authority. It refuses, leaving the running program untouched, and the Output panel says why:

Refused	Reason shown
A plain, unsigned program	<i>Managed controller accepts signed deploys only</i>
A bundle altered after signing	<i>payload hash mismatch</i>
A bundle signed under a different authority (another account, another site, a local keystore)	signer certificate not signed by the trusted CA

Refused	Reason shown
A bundle signed with a revoked certificate	certificate revoked
A connection without a certificate, or with one it does not trust	the TLS handshake fails before any command is accepted
A new site identity while already managed	<i>already provisioned — decommission it first</i>
A revocation list from another authority	issuer not in the trusted chain

A **standalone** controller accepts an unsigned program from any IDE that can reach it. That is the shipping posture, and the right one for a bench and for a site that has chosen not to manage its controllers. It also means anyone who can reach the controller on the network can program it; if that is not acceptable at your site, enrol it (Path A) or provision it into a site CA (Path B).

Note. The first signed deploy to a controller that has an identity but no trust anchor yet binds it to the signer's authority; the Output panel reports *trust anchor installed*. A controller enrolled into an account or provisioned into a site already has its anchor, and a standalone controller receives its deploys unsigned, so this only concerns bench units given a certificate from the local keystore.

17 Generating HMI Installers

This chapter applies to projects whose target is **WhiskerHMI**. The IDE turns such a project into a Windows installer that you hand to the customer; running it puts a complete HMI application on a PC, connected to the Nexus.io Automation Controller the project points at. Deploying to a controller is covered in *Connecting, Deploying and Emulating*; what the installed application does is covered in *The WhiskerHMI PC Application*.

17.1 What the installer contains

The generated installer bundles two kinds of content:

- **The WhiskerHMI runtime** — the same application for every project: the HMI window program, its I/O scanner service (which talks to the controller), the local history database used by Trends and CSV export, and the Microsoft Visual C++ runtime the program needs on a fresh PC.
- **Your project's compiled configuration** — produced by Build (F5) for a WhiskerHMI project:

File	Contents
<code>screens.hmi</code>	The screens and their elements and bindings
<code>symbols.json</code>	The tag table
<code>app_config.json</code>	Project name and version, display size, window layout, login settings and role table
<code>io_scanner_config.json</code>	The devices to read — normally one ArenaTCP device carrying the controller's address and port — and the history settings
<code>alerts_config.json</code>	The alert definitions (only when the project has alerts)

File	Contents
security\	Only when an ArenaTCP device has <i>Secure connection</i> ticked. For a managed controller (cloud account or site CA): the authority's CA chain and revocation list and a <code>PANEL_IDENTITY_REQUIRED.txt</code> marker — never a certificate. For a bench controller provisioned from the local keystore: this PC's station certificate and key and the CA chain

The runtime is identical from one project to the next; the configuration is what makes the installer yours.

The installer does **not** contain a panel identity. A panel that connects securely to a managed controller needs one identity per PC, issued from the Security dialog's **Panels** sub-section and imported on that PC after installation from the panel's **Panel ▶ Import panel identity...** menu; the same installer serves every panel PC. An identity already imported survives an update, because the installer adds and replaces files without removing it. The procedure is in *The WhiskerHMI PC Application*, under *Panel identity — connecting to a managed controller*.

17.2 Before you generate

- **Inno Setup 6** must be installed on the PC running the IDE. It is a free installer builder the IDE drives in the background; the dialog tells you if it cannot find it.
- The WhiskerHMI project must build without errors. For a WhiskerHMI project **Build (F5)** compiles the configuration files on the PC; it needs no controller connection, and the **Run** button starts the runtime locally so you can check the screens before packaging.
- The ArenaTCP device in the project's IO Config must carry the controller address the customer's PC will use. If the device has *Secure connection* ticked and the project resolves to a cloud account or a site, the build stages the authority's chain and revocation list and the Output panel reports *Panel identity: this build trusts ...; import a panel identity package on each panel PC (Security dialog → Panels)*. Only a project that resolves to the local keystore (a bench controller) gets a station certificate minted at build time. See *The WhiskerHMI PC Application*.

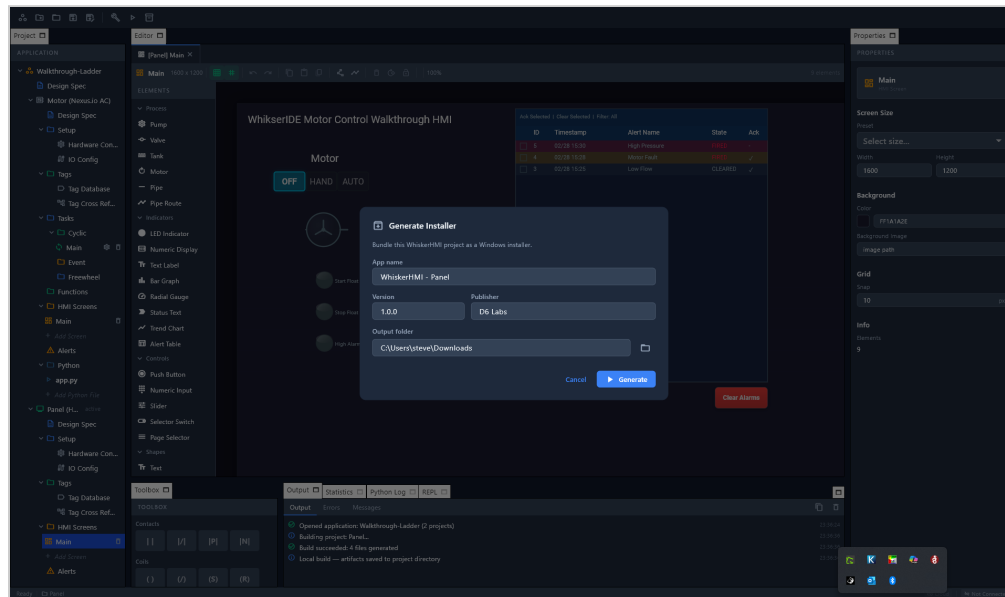
17.3 The Generate Installer dialog

With the WhiskerHMI project selected, click **Generate Installer** on the toolbar (it is only shown for WhiskerHMI projects). The dialog reads *Bundle this WhiskerHMI project as a Windows installer* and asks for:

Field	Default	Used for
App name	WhiskerHMI - <project name>	The Start Menu entry, desktop icon, the entry in Apps & features and the installer's title
Version	The project version, or 1.0.0	The installer file name and the version Windows records
Publisher	D6 Labs	The publisher shown by Windows; replace it with your company
Output folder	—	Where the installer is written; use Browse...

All four are required. Click **Generate**. The dialog switches to *Building installer...* and streams its log (also echoed to the Output panel): compiling the project, checking requirements, generating the installer script, running the Inno Setup compiler. When it finishes the title reads *Installer ready* with the path of the file, or *Installer failed* with the errors. **Open folder** shows the result in Explorer.

The output file is named `WhiskerHMI_Setup_<version>.exe` — from the Version field, not the app name — so give each project its own output folder.



Generate Installer dialog

Two warnings may appear in the Output panel without stopping the build: that the history database was not found (the installer then ships without Trends support) and that the Visual C++ redistributable was not found (the customer may need to install “Microsoft Visual C++ Redistributable (x64)” by hand on a fresh PC). Both components ship with the IDE; a warning means the IDE installation is incomplete.

17.3.1 If generation fails

The *Installer failed* page lists the reason. The ones you are likely to meet:

Message	What to do
<i>Inno Setup 6 not found</i>	Install Inno Setup 6 on this PC and generate again
<i>WhiskerHMI runtime not found or IO Scanner service not found</i>	The IDE's bundled runtime components are missing; reinstall the IDE
<i>Build directory does not exist</i>	The project did not build; fix the errors in the Output panel first
<i>Inno Setup failed (exit code N)</i>	The captured compiler output follows the message; the usual cause is an output folder that is read-only or still open in another program

The generator writes a temporary script into the output folder while it runs and removes it when the build succeeds; if it fails, the script is left there and can be sent to support.

17.3.2 Test the installer before hand-off

Run the new installer on your own PC first. The IDE's **Run** button shows the screens but not the installed form of the application, so this is the only way to see what the customer will see: the wizard pages, the SmartScreen prompt, the shortcuts, and the application starting with its status bar reading *IO Scanner connected* and the controller listed under *Devices*. Uninstall it from Apps & features afterwards if this PC is not meant to keep it.

17.4 What the customer sees

The installer is a standard Windows setup wizard. It needs administrator rights and installs:

- the application under `C:\Program Files\WhiskerHMI\<ProjectName>\` ;
- the configuration files under `C:\ProgramData\WhiskerHMI\<ProjectName>\config\` ;
- a Start Menu shortcut and, if the customer ticks *Create a desktop icon*, a desktop shortcut;
- a *Launch* option on the last page.

Nothing is registered as a Windows service. The application starts its I/O scanner and the history database itself each time it is launched and stops them when it closes, so the controller is polled and history is collected only while the application is running on a logged-in desktop. An operator who wants the panel up after a reboot puts the shortcut in the Windows Startup folder, or sets the PC to log in automatically.

The installer is not code-signed. Windows SmartScreen therefore shows an “unknown publisher” warning the first time it runs; the customer chooses *More info* and *Run anyway*. If you distribute installers regularly, sign them with your own code-signing certificate after generation.

Deliver the `.exe` however suits the site — email, download link, USB stick, or the customer's software distribution system. It has no dependencies beyond a 64-bit Windows PC.

17.5 Changing the configuration on site

The installed application has no settings screen. Everything it knows comes from the configuration folder, and the controller address lives in the ArenaTCP device entry of `io_scanner_config.json`. There are two ways to change it:

1. **Regenerate.** Edit the device in the IDE (IO Config → select the device → Properties panel), Build, Generate Installer with a new Version, and run the new installer on the PC. This is the supported path and keeps the project as the record of what is installed.
2. **Edit the file.** For a quick fix on site, an administrator can edit the `ip` (and `port`) of the device in `C:\ProgramData\WhiskerHMI\<ProjectName>\config\io_scanner_config.json` and restart the application. The next installer run overwrites the file.

Per-user data the application creates on the PC — saved trend presets, the offline login cache and the audit log — lives under the Windows user's application-data folder and is not touched by either path.

17.6 Updating an installed application

1. Make the changes in the IDE and raise the **Version** in the Generate Installer dialog.
2. Generate and send the new installer.
3. The customer runs it over the existing installation; the wizard replaces the application and the configuration in place. Shortcuts stay, and per-user data (presets, audit log) is kept.

There is no in-place update mechanism and no need for one: every version is a complete installer.

18 Project Properties Reference

Every project-wide setting lives in the **Properties panel**: click the project's header node in the Project Tree and the panel shows **Project Settings**, split into sections. Which sections appear depends on the project's target. This chapter lists every section and every field, in the order the panel shows them, for the three targets:

Section	Nexus.io controller	SmartController (Classic)	WhiskerHMI
Identity (Name, Version, Target)	yes	yes, plus Serial	yes
HMI Settings	yes	—	yes
Deployment	yes	yes	yes
Cloud (Cloud edition only)	yes	Broker note only	yes
Device Settings	—	yes	—
Modbus Server	yes	—	—
CIP tag server	yes	—	—
Display	—	—	yes
Authentication	—	—	yes
Security	—	—	yes
Window Layout	—	—	yes

Changes take effect as you type or when you leave the field; there is no Apply button. They are written to disk with the next **Save**.

PROPERTIES

Project Settings

Name
lift_station_plc

Version
1.0.0

Target
Nexus.io AC

HMI Settings

PIN
0000

4-6 digit PIN for HMI settings access

Cloud PIN Login
☒ Required at the panel

Your organization's PIN policy is off — the panel will show "PIN login is not enabled" until it is turned on in Whisker.io. Switch this off if that is not intended.

Session Timeout (s)
300

Re-lock after this much idle time; screen-off always re-locks.

Offline Fallback
☐ Locked while the cloud is unreachable

When on, the project PIN above logs in as "Local maintenance" if Whisker.io cannot be reached. Off is the secure default.

Site identity
Automatic

No site identity is imported on this PC (Security → Site → Import site identity). Automatic uses the local keystore.

Roles at the panel

Which role may do each action on this panel. "Any logged-in user" is anyone who unlocked it with a PIN. Roles come from your Whisker.io account.

36 account roles (account 10)

18.1 Identity

Field	What it does	Default
Name	The project's name as shown in the tree, in build messages and on the controller. Read-only here; rename the project from the tree	Set when the project is created
Version	Free-text version string stored with the project and included in the deployed bundle	1.0.0
Target	The controller type chosen when the project was created (Nexus.io AC , SmartController Classic or WhiskerHMI). Read-only; a project's target cannot be changed	Set at creation
Serial (<i>Classic only</i>)	The connected SmartController's hardware address, or (<i>not connected</i>)	—

18.2 HMI Settings

Shown for Nexus.io and WhiskerHMI projects. These settings travel with the project to the panel or the PC application.

Field	What it does	Default
PIN	Four to six digits. On the Nexus.io controller this PIN opens the panel's Settings screen (network settings, touch calibration, audit log). It is also the <i>maintenance</i> PIN used by Offline Fallback below	0000
Cloud PIN Login	Switch. When on, the panel starts locked; operators pick their name and enter their Whisker.io PIN before they can act. Shown only when your Whisker.io account has PIN login enabled (or the project already has it on). See <i>Setting Up Your Nexus.io Automation Controller</i>	Off
Session Timeout (s)	Seconds of no touches before the panel locks again. Screen sleep always locks. Range 30–86400. Shown only when Cloud PIN Login is on	300
Offline Fallback	Switch. When on, the project PIN above can log an operator in as <i>Local maintenance</i> if Whisker.io cannot be reached. Off is the secure choice. Shown only when Cloud PIN Login is on	Off

Field	What it does	Default
Site identity	Dropdown: Automatic or one of the site identities imported on this PC (Security dialog → Site section). Managed connects and signed deploys for this project use the selected site's certificate. <i>Automatic</i> uses the only imported site, or the local keystore if none is imported. A colleague's project that names a site you have not imported stays selectable and is marked (<i>not imported on this PC</i>)	Automatic
Roles at the panel	Table of panel actions with a required role for each. Roles come from your Whisker.io account; press the refresh button to reload them. Shown only when Cloud PIN Login is on. The action list is the same as the WhiskerHMI Security table below	Any logged-in user
History Retention (days)	How long the controller keeps historian samples for tags with Hist ticked in the Tag Database, plus any tag bound to a trend chart. Must be greater than 0. See <i>Historian and Trends</i>	60

Note. When you deploy a project with Cloud PIN Login on, the IDE first obtains a device credential for the connected controller's serial number from Whisker.io. If that fails the deploy stops and the Output panel says why. This happens on every deploy, so a project for a controller that runs without Whisker.io (an offline site, or the Offline edition) must have Cloud PIN Login switched **off**. The switch can always be turned off, even when you are not logged in.

18.3 Deployment

Field	What it does	Default
Store editable source on target	Switch. On: the deployed bundle includes the editable project, so anyone who connects later can pull it back with Download from target (see <i>Connecting, Deploying and Emulating</i>). Off: only the compiled program goes to the controller — choose this to keep your source off the device. The panel shows the current state as <i>On — the project can be pulled back from the target</i> or <i>Off — source kept off the device (IP protection)</i>	On

18.4 Cloud

Shown in the Cloud edition only.

Field	What it does	Default
Broker	Information only. The controller's connection to the Whisker.io message broker — address, TLS and per-device credentials — is installed on the unit at the factory and is not a project setting. A unit that was never factory-provisioned falls back to the plain broker until it is	—
Update Interval (s)	How often the controller publishes its process variables to the cloud. Not shown for Classic projects (they use Update Period under Device Settings)	60
ROC Interval (s)	Report-on-change rate limit: when a monitored process variable crosses its change threshold between full updates, the controller publishes it at most once per this many seconds	5

The account and location a project is published under are chosen in the **Cloud Explorer** when you use **Publish to Cloud**, not here. The Application's link to a cloud application record is on the Application node (see the end of this chapter).

18.5 Device Settings (Classic only)

These fields become the SmartController's `config.json` at Build.

Field	What it does	Default
Update Period (min)	Minutes between cloud updates	1
Timezone Offset	Hours from UTC applied to the controller's clock	-5

Field	What it does	Default
Logger Profile	How much the controller logs: <code>default</code> , <code>production</code> (less) or <code>debug</code> (everything — use it to diagnose start-up problems from the console). Any other value means <code>default</code>	<code>default</code>
Modbus Slave	Enabled/Disabled. When enabled the RS-485 port answers Modbus RTU requests	Disabled
Slave ID	The controller's Modbus RTU address. Shown when Modbus Slave is enabled	<code>1</code>
Baud Rate (slave)	RS-485 speed in slave mode	<code>9600</code>
Modbus Master	Enabled/Disabled. When enabled the controller polls Modbus RTU devices on RS-485	Disabled
Baud Rate (master)	RS-485 speed in master mode	<code>9600</code>
Timeout (ms)	How long to wait for a device reply before recording a failure	<code>1000</code>

18.6 Modbus Server (Nexus.io only)

The Nexus.io controller always runs a Modbus TCP server; these settings control how tags marked **Expose** in the Tag Database are numbered. See *Modbus Server and Map Viewer* for the whole picture.

Field	What it does	Default
Auto-numbering mode	Dropdown: Alphabetical , Group by data type , Creation order . How a newly exposed tag gets its Modbus address. An address typed by hand in a tag's properties always overrides the automatic one. Changing the mode asks for confirmation	Group by data type
Address ranges per data type	Shown only for <i>Group by data type</i> : a start address and a size for each of Coil bool , HR bool , HR int and HR real . Start 0–65535, size 1–65536	Coil bool 0 / 1000; HR bool 0 / 1000; HR int 1000 / 1000; HR real 2000 / 1000
Renumber all auto-assigned tags	Button. Opens a confirmation that reassigns every exposed tag's address under the current mode. Keep manually-assigned addresses (ticked by default) preserves hand-set addresses; untick it to reassign everything from scratch	—

Warning. Renumbering changes the addresses every SCADA system reads from this controller. Export a fresh Modbus map afterwards and update the other side. {.warning}

18.7 CIP tag server (Nexus.io only)

Serves the tags you publish by name over EtherNet/IP, for a Logix-style SCADA driver. See *CIP Tag Server*.

Field	What it does	Default
CIP tag server	Switch. Off, or On — tags are served by name on TCP 44818 . While on, the section counts how many tags are published; which ones is chosen per tag in the Tag Database or in tag Properties	Off

Warning. CIP has no authentication. Leave the server off on a controller reachable from an untrusted network. {.warning}

18.8 Display (WhiskerHMI only)

Field	What it does	Default
Width / Height	The design resolution of the PC application, in pixels. HMI screens are laid out at this size	1920 × 1080
Preset chips	One-click presets: 1920x1080, 1366x768, 2560x1440, 3840x2160, 1024x768	—

18.9 Authentication (WhiskerHMI only)

Field	What it does	Default
Cloud URL	The Whisker.io service the PC application contacts to authenticate operators. Leave the default unless D6 Labs tells you otherwise	The production Whisker.io service

Field	What it does	Default
Offline Mode	Dropdown: Allow Cached (operators who have logged in before can log in while the cloud is unreachable), Require Online (no cloud, no login), Disabled	Allow Cached
Session Timeout (min)	Minutes of inactivity before the operator is logged out	480
History Retention (days)	The same setting as under HMI Settings, shown here for WhiskerHMI projects	60

18.10 Security (WhiskerHMI only)

Field	What it does	Default
Security switch	On: operators log in with their Whisker.io account or PIN, and each action below needs the selected role; every action is audited. Off: no login and every action allowed; actions are still written to the local audit log	Off
Roles status	Shows how many roles were loaded from the account, with a Refresh roles from Whisker.io button	—

Field	What it does	Default
Action / Required role table	One row per action: View screens , Buttons and switches , Setpoints and sliders , Acknowledge alarms , Clear alarms , Save trend presets , Export history , Open windows , Audit log and settings . Each row picks a role; <i>View screens</i> and <i>Acknowledge alarms</i> may also be set to need no login	Any logged-in user
PIN login at the console	Switch. Lets operators log in with their PIN instead of a full account sign-in	Off
Offline grace (hours)	How long a cached login stays valid without the cloud	24

Individual widgets can be stricter than the table: select a button, switch, input, slider, faceplate or alarm table in the HMI designer and set its **Required role**.

18.11 Window Layout (WhiskerHMI only)

One row per HMI screen:

Column	What it does	Default
Screen	The screen's name (read-only)	—
Role	Main (the application's main window), Popup (opened from the Windows menu), Auto Open (opens with the application) or None	First screen <i>Main</i> , others <i>None</i>
Monitor	Default , or 1 (primary) to 4 — which display the window opens on	Default

Column	What it does	Default
Fill	Checkbox. Makes the window borderless and full-size on its monitor	Off

See *The WhiskerHMI PC Application* for how the layout behaves at run time.

18.12 The Application node

Click the Application header at the top of the tree to see **Application Properties**:

Field	What it does	Default
Name	The Application's name; also the archive's file name	Set at creation
Version	Free-text version for the whole Application	1.0
Customer, Site, Notes	Free-text metadata stored with the Application	empty
Cloud	<i>Linked to cloud</i> with the Application ID once registered; otherwise a Register with Cloud button (Cloud edition, signed in). Registration creates the cloud record that access control and signing refer to	Not linked
Devices	One row per project: its target, the last connection used (address and port, or serial port) and the controller's serial number once known	—

19 Ladder Instruction Reference

One entry per instruction available in the Ladder Logic Editor for a Nexus.io Automation Controller, grouped by purpose. How to place and wire them is in *Ladder Logic Editor*.

19.1 How to read the tables

- **Direction.** *Rung* is the network's own power flow — every block is driven by it and executes only while the network is true, so it is listed only where it carries a meaning beyond “enable”. *In* and *Out* are pins bound to tags in the Properties panel. *In/Out* is one tag that the block reads and writes back each scan. *Setting* is chosen in the Properties panel (a dropdown or checkbox), not bound to a tag.
- **Type.** BOOL, INT, DINT or REAL as in the Tag Database. *Number* means a DINT or REAL tag, or a numeric literal typed into the field; a literal compiles to a constant and creates no tag. *Word* means INT or DINT.
- **Unwired pins read 0.** A blank input is zero, a blank output is simply not written. Where zero has a special meaning — a timeout that disables supervision, a day mask that means every day — the entry says so.
- **Instance names.** Every block has an instance name that identifies its private state. It is not a tag; to observe a value, bind an output pin to a tag.
- **Integer or float.** Compute blocks work in integer or floating point according to the tags you bind; SCALE, FILTER and PID always compute in floating point and their outputs must be REAL tags.

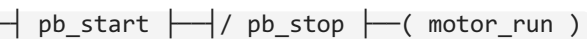
19.2 Basic: contacts and coils

19.2.1 NO Contact, NC Contact

A contact reads a BOOL and passes or blocks power. NO passes power while the tag is TRUE; NC while it is FALSE. A contact on a word tag can test a single bit by entering a **Bit Index**, shown as `status_word.3`.

Pin	Direction	Type	Meaning
Tag Name	In	BOOL (or word + bit index)	The value tested

NC inverts the test, not the field device: a normally-closed stop button wired to an input reads TRUE while it is not pressed, so test it with an NO contact.

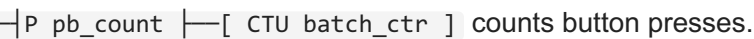
Example: 

19.2.2 P Contact, N Contact (edge)

P passes power for exactly one scan when the tag goes FALSE to TRUE; N for one scan when it goes TRUE to FALSE.

Pin	Direction	Type	Meaning
Tag Name	In	BOOL	The value watched for an edge

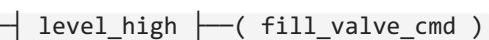
A one-scan pulse is shorter than the monitor refresh, so it is invisible in live view; feed it into a counter or a Set coil to see its effect.

Example:  counts button presses.

19.2.3 Coil, Neg Coil

A coil writes the network's power to a BOOL every scan: TRUE when the network is true, FALSE otherwise. Neg Coil writes the inverse.

Pin	Direction	Type	Meaning
Output Tag	Out	BOOL (or word + bit index)	Written every scan

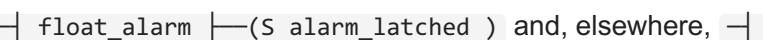
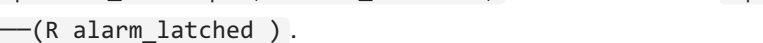
Example: 

19.2.4 Set Coil, Reset Coil

Set writes TRUE when the network is true and leaves the tag TRUE when the network goes false. Reset writes FALSE when the network is true and otherwise leaves the tag alone. Pair every Set with a Reset somewhere in the program.

Pin	Direction	Type	Meaning
Output Tag	Out	BOOL	Latched (Set) or cleared (Reset)

An alarm driven by a Set coil must use a latched alert rule; a non-latched rule clears the alarm the ladder just latched (see *Alerts Editor*).

Example:  and, elsewhere, .

19.2.5 Flash

While the network is true, the output goes TRUE at once and then toggles every RATE milliseconds (50 % duty). When the network goes false the output is FALSE and the phase resets.

Pin	Direction	Type	Meaning
RATE	In	Number (ms)	Half-period; 0 or unwired = 500 ms
Output Tag	Out	BOOL	The flashing output

Example:  with RATE 500 flashes a beacon at 1 Hz.

19.3 Timers

All three timers share the same pins. PT is interpreted in the chosen UNIT; ET is always reported in milliseconds and freezes at PT rather than climbing without limit.

19.3.1 TON — On-Delay Timer

When the network goes true, ET accumulates; Q turns on once ET reaches PT. When the network goes false, ET resets to 0 and Q turns off.

Pin	Direction	Type	Meaning
IN	Rung	BOOL	Start timing while true
PT	In	Number	Preset, in UNIT
UNIT	Setting	Seconds / Milliseconds	Unit of PT (default Seconds)
Q	Out	BOOL	TRUE once PT has elapsed
ET	Out	DINT (ms)	Elapsed time, frozen at PT

ET is not retained across a false network; for accumulated run time use TONR.

Example:  with Q bound to lead_start delays a pump start by five seconds.

19.3.2 TOF — Off-Delay Timer

Q turns on as soon as the network goes true. When the network goes false, Q stays on for PT and then drops.

Pin	Direction	Type	Meaning
IN	Rung	BOOL	Q follows this immediately when it rises
PT	In	Number	Off delay, in UNIT
UNIT	Setting	Seconds / Milliseconds	Unit of PT
Q	Out	BOOL	Held on for PT after IN falls
ET	Out	DINT (ms)	Time since IN fell, frozen at PT

Example: a cooling fan that runs for 60 s after the motor stops.

19.3.3 TP — Pulse Timer

A rising network starts a pulse: Q is on for exactly PT whatever the network does afterwards. It retriggers only after the pulse completes.

Pin	Direction	Type	Meaning
IN	Rung	BOOL	Rising edge starts the pulse
PT	In	Number	Pulse length, in UNIT
UNIT	Setting	Seconds / Milliseconds	Unit of PT
Q	Out	BOOL	TRUE for PT
ET	Out	DINT (ms)	Time into the pulse, frozen at PT

Example: a 2 s blowdown valve pulse on each rising edge of `blowdown_req`.

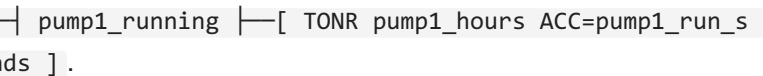
19.3.4 TONR — Retentive Timer

Like TON, but the accumulated time is kept when the network goes false, so it totals across many runs. R resets it. The accumulator lives in a tag you nominate, so it can be retained and written from the panel.

Pin	Direction	Type	Meaning
IN	Rung	BOOL	Accumulate while true
PT	In	Number	Preset, in UNIT

Pin	Direction	Type	Meaning
UNIT	Setting	Seconds / Milliseconds	Unit of PT, ACC and ET (default Seconds)
R	In	BOOL	Reset the accumulator to 0 (dominant)
Accumulator tag (ACC / ACC_OUT)	In/Out	DINT	The running total, read in and written back
Q	Out	BOOL	TRUE while ACC $\geq$ PT
ET	Out	DINT	Equals the accumulator, in UNIT

Choose Seconds for run-time meters: milliseconds overflow a 32-bit tag in about 25 days, seconds last 68 years. Make the accumulator tag VAR_RETAIN and it survives a power cycle; one tag per block.

Example:  .

19.4 Counters

PV is not a ceiling: counting continues past it; PV only decides Q.

19.4.1 CTU — Count Up

CV increments on each rising edge of the network. Q is on while CV $\geq$ PV. R resets CV to 0.

Pin	Direction	Type	Meaning
CU	Rung	BOOL	Count on each rising edge
R	In	BOOL	Reset CV to 0
PV	In	Number	Preset
Q	Out	BOOL	CV $\geq$ PV
CV	Out	DINT	Current count

Example: `—|P bottle_sensor |—[ CTU batch PV=24 ]` with Q bound to `batch_done`.

19.4.2 CTD — Count Down

CV decrements on each rising edge. Q is on while $CV \leq 0$. LD loads CV with PV. Counting below zero is normal behaviour, not a fault.

Pin	Direction	Type	Meaning
CD	Rung	BOOL	Count down on each rising edge
LD	In	BOOL	Load CV with PV
PV	In	Number	Value loaded
Q	Out	BOOL	$CV \leq 0$
CV	Out	DINT	Current count

Example: doses remaining in a batch, loaded from a recipe tag.

19.4.3 CTUD — Count Up/Down

Both directions on one instance. R (reset to 0) dominates LD (load PV).

Pin	Direction	Type	Meaning
CU	Rung	BOOL	Count up on each rising edge
CD	In	BOOL	Count down on each rising edge
R	In	BOOL	Reset CV to 0
LD	In	BOOL	Load CV with PV
PV	In	Number	Preset
QU	Out	BOOL	$CV \geq PV$
QD	Out	BOOL	$CV \leq 0$
CV	Out	DINT	Current count

Example: vehicles in a bay — entry sensor on CU, exit sensor on CD.

19.5 Edge detection and latches

19.5.1 R_TRIG, F_TRIG

Q is TRUE for one scan when the network rises (R_TRIG) or falls (F_TRIG). The block form of a P or N contact, for when the pulse is needed as a named tag.

Pin	Direction	Type	Meaning
CLK	Rung	BOOL	The signal watched
Q	Out	BOOL	One-scan pulse

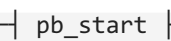
Example:  pulses `comm_lost` when a device drops.

19.5.2 SR — Set-Dominant Latch, RS — Reset-Dominant Latch

Q1 sets on the set input and resets on the reset input. They differ only when both are true at once: SR sets, RS resets.

Pin	Direction	Type	Meaning
S1 (SR) / S (RS)	Rung	BOOL	Set; the network drives it unless a set tag is given
R (SR) / R1 (RS)	In	BOOL	Reset
Q1	Out	BOOL	The latched state

Use SR where an alarm must be seen even if the reset is stuck on; RS where stop must always beat start.

Example:  with Q1 bound to `motor_cmd`.

19.6 Compare

19.6.1 EQ, NE, GT, GE, LT, LE — inline comparisons

An inline comparison sits in a segment like a contact and passes power when the test is true. The operator is chosen in the Properties panel.

Pin	Direction	Type	Meaning
IN1	In	Number	Left operand

Pin	Direction	Type	Meaning
IN2	In	Number	Right operand

Example: $\neg \text{level} > 80 \neg (\text{high_alarm})$. For start/stop levels prefer HYST, which adds the deadband.

19.6.2 CMP — Compare Block

Compares two values and reports all three relations at once.

Pin	Direction	Type	Meaning
IN1	In	Number	Left operand
IN2	In	Number	Right operand
GT	Out	BOOL	IN1 > IN2
EQ	Out	BOOL	IN1 = IN2
LT	Out	BOOL	IN1 < IN2

Example: IN1 = `aho_mode`, IN2 = 1 decodes an Off/Hand/Auto word into LT = off, EQ = hand, GT = auto in one network.

19.6.3 MEQ — Masked Equal

Q is TRUE when IN1 and IN2 match in every bit where MASK has a 1; bits where MASK is 0 are ignored. Integer only.

Pin	Direction	Type	Meaning
IN1	In	Word	Source
MASK	In	Word	Bits that matter
IN2	In	Word	Compare value
Q	Out	BOOL	Masked match

Example: test a drive status word for “ready and not faulted” while ignoring its speed bits.

19.7 Math

19.7.1 MATH operations

MOVE, ADD, SUB, MUL, DIV, MOD, MIN, MAX, ABS, SHL, SHR, ITOF, FTOI, the Bitwise group and the Advanced Math group are all one block with a different operation. Integer or float is chosen from the tags you bind. Unary operations use IN1 only.

Pin	Direction	Type	Meaning
IN1	In	Number	First operand
IN2	In	Number	Second operand (binary operations)
OUT	Out	Number	Result

Operation	Result
MOVE	OUT = IN1
ADD, SUB, MUL, DIV	OUT = IN1 + - × ÷ IN2. Division by zero returns 0 without a fault; guard the divisor.
MOD	Remainder of IN1 ÷ IN2 (integer only; 0 when IN2 is 0)
MIN, MAX	Smaller / larger of IN1 and IN2
ABS	Absolute value
SHL, SHR	Shift left / right by IN2 bits (0–31), zero-filled (integer only)
ITOF	Integer to float
FTOI	Float to integer, truncated toward zero (same as TRUNC)
AND, OR, XOR, NOT	Bitwise word operations (integer only) — not the network's contact logic
ROL, ROR	Rotate left / right by IN2 bits, wrapping (integer only)
SWPB	Swap the two bytes inside each 16-bit half: 0x1234 → 0x3412
SWPW	Swap the 16-bit halves of a 32-bit value: 0x12345678 → 0x56781234
SQRT	Square root; negative input returns 0

Operation	Result
ROUND	REAL to integer, nearest
TRUNC	REAL to integer, toward zero
EXPT	IN1 raised to IN2
LN, LOG, EXP	Natural log (≤ 0 returns 0), base-10 log, e to the power IN1
SIN, COS, TAN, ASIN, ACOS, ATAN	Trigonometry in radians; ASIN/ACOS outside ± 1 return 0

SWPB and SWPW repair Modbus values from devices that deliver bytes or words reversed; apply them to the raw tag before scaling. They work on the raw 32 bits, so they also fix a reversed REAL.

Example: `[ MATH SQRT IN1=dp_pa OUT=flow_root ]` for flow from a differential-pressure transmitter.

19.7.2 LIMIT — Clamp to Range

OUT = IN clamped between MN and MX.

Pin	Direction	Type	Meaning
MN	In	Number	Lower limit
IN_VAL	In	Number	Value
MX	In	Number	Upper limit
OUT	Out	Number	Clamped value

Example: keep a computed drive speed between 0 and 100.

19.7.3 SEL — Select by Boolean

OUT = IN1 when G is TRUE, otherwise IN0. A bit copy, so it works for any type.

Pin	Direction	Type	Meaning
G	In	BOOL	Selector
IN0	In	Number	Chosen when G is FALSE
IN1	In	Number	Chosen when G is TRUE
OUT	Out	Number	Selected value

Example: day/night setpoint selection.

19.7.4 MUX — Select 1 of 8

OUT = the input selected by K (0–7, clamped). Unwired inputs read 0.

Pin	Direction	Type	Meaning
K	In	Number	Selector
IN0 ... IN7	In	Number	Candidates
OUT	Out	Number	Selected value

Example: pick one of several recipe setpoints by a mode number.

19.7.5 PACK, UNPACK

PACK assembles up to 16 BOOL inputs into a word (B0 is bit 0). UNPACK splits a word into 16 BOOLs. Only wired bits need binding.

Pin	Direction	Type	Meaning
B0 ... B15	In (PACK) / Out (UNPACK)	BOOL	Individual bits
OUT (PACK) / PACKED (UNPACK)	Out / In	Word	The packed word

Example: build a Modbus status word from alarm bits; decode a drive's status word; drive coils from a DRUM's pattern.

19.8 Process

19.8.1 SCALE — Linear Scaling

Maps an input span onto an output span: $OUT = OUT_LO + (IN - IN_LO) \times (OUT_HI - OUT_LO) / (IN_HI - IN_LO)$. Always floating point; OUT must be a REAL tag.

Pin	Direction	Type	Meaning
IN_VAL	In	Number	Raw value
IN_LO, IN_HI	In	Number	Input span

Pin	Direction	Type	Meaning
OUT_LO, OUT_HI	In	Number	Output span (may be reversed)
Clamp output to range	Setting	checkbox	Limit OUT to the output span
OUT	Out	REAL	Engineering value

Tick **Clamp output to range** for any transmitter scaling; without it a failed sensor extrapolates to absurd values that propagate into alarms and control.

Example: 4–20 mA counts 800–4000 to 0–100 % level.

19.8.2 HYST — Deadband Comparator

Q turns on at ON_LEVEL, off at OFF_LEVEL, and holds between them. Direction is inferred: ON above OFF is a rising trip (start on high level), ON below OFF a falling trip (start on low level). Equal levels behave as a plain comparison.

Pin	Direction	Type	Meaning
IN_VAL	In	Number	Measured value
ON_LEVEL	In	Number	Level at which Q turns on
OFF_LEVEL	In	Number	Level at which Q turns off
Q	Out	BOOL	The result

The start/stop block for tanks and wet wells; the deadband is what stops a pump short-cycling.

Example: IN_VAL `wet_well_level`, ON_LEVEL `sp_lead_start`, OFF_LEVEL `sp_lead_stop`, Q `call_lead`.

19.8.3 TOTAL — Totalizer

Integrates a rate into a running total. A fractional carry is kept internally so a small rate still accumulates rather than rounding to nothing each scan.

Pin	Direction	Type	Meaning
RATE	In	Number	The rate being integrated

Pin	Direction	Type	Meaning
TIMEBASE	Setting	per second / per minute / per hour	What RATE is per (default per minute)
Accumulator tag (ACC / ACC_OUT)	In/Out	DINT	The running total
RESET	In	BOOL	Zero the total (dominant)
TOTAL	Out	DINT	Equals the accumulator

Same retained-tag rule as TONR. Pair with SQRT for differential-pressure flow.

Example: RATE `flow_gpm`, TIMEBASE per minute, accumulator `flow_total_gal` (VAR_RETAIN).

19.8.4 FILTER — First-Order Lag

Smooths a noisy signal. TC is the time constant in seconds: the output reaches about 63 % of a step in one TC. $TC \leq 0$ passes the input through. OUT must be a REAL tag.

Pin	Direction	Type	Meaning
IN_VAL	In	Number	Raw signal
TC	In	Number (s)	Time constant
OUT	Out	REAL	Filtered signal

Example: filter a 4–20 mA level before it feeds HYST so jitter cannot chatter a pump.

19.8.5 ROC — Rate of Change

Samples the input every PERIOD milliseconds and reports the change per TIMEBASE unit, holding the output between samples. The first sample is suppressed so a startup transient is not reported as a huge rate. OUT must be REAL.

Pin	Direction	Type	Meaning
IN_VAL	In	Number	Signal

Pin	Direction	Type	Meaning
PERIOD	In	Number (ms)	Sample period; 0 = 1000
TIMEBASE	Setting	per second / per minute / per hour	Units of OUT
OUT	Out	REAL	Rate of change

Example: leak and burst detection, rapid-drawdown alarms.

19.8.6 PID — Closed-Loop Controller

A full PID controller with anti-windup and derivative on the process value. Always floating point: every setpoint, value, gain and output must be a REAL tag. The block computes on its own SAMPLE_MS period, so tuning means the same thing whatever the scan rate.

Pin	Direction	Type	Meaning
SP	In	REAL	Setpoint
PV	In	REAL	Process value
KP	In	REAL	Proportional gain
KI_TI	In	REAL	Ki (per second) in independent form; Ti (seconds per repeat, 0 disables) in ISA form
KD_TD	In	REAL	Kd in independent form; Td (seconds) in ISA form
Gain form	Setting	Independent gains (Kp, Ki, Kd) / ISA standard (Kp, Ti, Td)	How the gains are read
MODE_AUTO	In	BOOL	1 = auto, 0 = manual; unwired = always auto
CV_MAN	In	REAL	Output while in manual
CV_MIN, CV_MAX	In	REAL	Output limits

Pin	Direction	Type	Meaning
Direct acting	Setting	checkbox	Off = reverse acting (output rises when PV is below SP), the usual fill or pressure case; on = direct (cooling, relief)
SAMPLE_MS	In	Number (ms)	Control period (default 100)
CV	Out	REAL	Controller output
ERR	Out	REAL	SP – PV
SAT	Out	BOOL	Output at a limit

The same number means Ki in one form and Ti in the other; getting the form wrong tunes the loop backwards. Manual-to-auto is bumpless. For on/off equipment use HYST; PID is for modulating control.

Example: SP `sp_level` , PV `level_filtered` , CV `vfd_speed_cmd` with CV_MIN 0 and CV_MAX 100.

19.8.7 ALT — Lead/Lag Pump Alternator

Decides which pumps run; CALL says how many the process wants (0 = none, 1 = lead, 2 = lead and lag, 3 = all). Up to three pumps; leave the third unwired for a duplex station. An unavailable pump is skipped, not counted, so a call for two still starts two.

Pin	Direction	Type	Meaning
CALL	In	Number	Pumps wanted, 0–3
Mode	Setting	Fixed order (1, 2, 3) / Rotate each start / Balance runtime (least-run leads)	How the lead is chosen (default Rotate)

Pin	Direction	Type	Meaning
ROTATE	In	BOOL	Rotate mode only: rising edge advances the lead; unwired = advance on every new call
AVAIL1 ... AVAIL3	In	BOOL	Pump can run; a pump with a RUN output but no AVAIL tag is always available
RT1 ... RT3	In	DINT	Balance mode only: runtime, normally a TONR accumulator
RUN1 ... RUN3	Out	BOOL	Start command per pump
LEAD	Out	DINT	1-based lead pump, 0 when none is called

Use it instead of hand-rolled alternation from counters and latches, which double-starts on a failure or strands a failed pump as lead.

Example: CALL from two staged HYST blocks (`call_lead` + `call_lag` via ADD), AVAIL from each pump's fault and HOA logic, RUN outputs to the MOTOR blocks' CALL contacts.

19.9 Industrial

19.9.1 DEBOUNCE — Input Filter

Q follows the network only after it has held its new value continuously for PT. Symmetric: it filters both on and off transitions, replacing the two-timer idiom.

Pin	Direction	Type	Meaning
IN	Rung	BOOL	The chattering signal
PT	In	Number	Stable time

Pin	Direction	Type	Meaning
UNIT	Setting	Milliseconds / Seconds	Unit of PT (default Milliseconds)
Q	Out	BOOL	Filtered signal
ET	Out	DINT (ms)	Time toward the trip, frozen at PT

Example: a float switch with PT 100 ms.

19.9.2 MOTOR — Motor Device Block

The networks every motor needs in one block: Off/Hand/Auto selection, an overload input, “did the starter pull in”, a start counter and a runtime meter — and a single instance for a faceplate to bind to.

Pin	Direction	Type	Meaning
CALL	Rung	BOOL	Auto-mode run request
RUN_FB	In	BOOL	Run feedback from the starter (required)
MODE	In	Number	0 = Off, 1 = Hand, 2 = Auto; unwired = Auto
OL	In	BOOL	Overload or external fault
FTS_T	In	Number	Fail-to-start timeout; 0 or unwired disables supervision
UNIT	Setting	Seconds / Milliseconds	Unit of FTS_T
SP	In	Number	Speed setpoint, passed through to SPEED
STARTS	In/Out	DINT	Start counter tag (make it VAR_RETAIN)

Pin	Direction	Type	Meaning
RT	In/Out	DINT	Runtime seconds tag (make it VAR_RETAIN)
RUN	Out	BOOL	Contactor output (required)
SPEED	Out	Number	Mirrors SP when both are bound
FTS	Out	BOOL	Failed to start
FLT	Out	BOOL	OL or FTS

Hand means running, like the middle position of a three-way switch wired to the starter; Auto follows CALL; anything else is Off, so a corrupt mode word can never start a motor. The block controls and annunciates; it does not latch. Faults clear when their cause goes away, and RUN is not withdrawn by a fault — the overload contact breaks the starter circuit, and the motor resumes when someone resets it. Latch a fault with a latched alert on FLT, or with one network on CALL if the output must drop. Starts count rising edges of RUN; runtime accumulates only while RUN_FB is true. Hours = RT ÷ 3600.

Example: CALL from `run_p1` (an ALT RUN output), RUN_FB `p1_aux`, MODE `p1_hoa`, OL `p1_o1`, FTS_T 5 seconds, RUN `p1_start`, FLT `p1_fault`.

19.9.3 VALVE — Valve Device Block

The sibling of MOTOR: “did the actuator actually travel”. Which valve style you have is declared by which pins you bind.

Pin	Direction	Type	Meaning
CALL_OPEN	Rung	BOOL	Open request (required)
CALL_CLOSE	In	BOOL	Close request, dual-acting only
OPEN_FB, CLOSE_FB	In	BOOL	Limit switches
MODE	In	Number	0 = Off, 1 = Open, 2 = Close, 3 = Auto; unwired = Auto

Pin	Direction	Type	Meaning
PT	In	Number	Travel timeout; 0 or unwired disables supervision
UNIT	Setting	Seconds / Milliseconds	Unit of PT
OPEN	Out	BOOL	Open output (required)
CLOSE	Out	BOOL	Close output, dual-acting only
FTO, FTC	Out	BOOL	Failed to open / close

Spring return: bind only OPEN; dropping the request closes the valve. Dual-acting: bind CLOSE too; if CALL_CLOSE is also bound, Auto closes only on that request and an actuator with neither request holds position, so partial travel is a normal state. Each direction is supervised only if its own limit switch is bound, so a valve with one switch or none never faults on what it cannot observe. FTO and FTC self-clear like MOTOR's faults.

Example: CALL_OPEN from `drum_step_bws` (an UNPACK bit), OPEN_FB `bws_ols` , PT 30 s, OPEN `bws_valve_cmd` , FTO `bws_fto` .

19.9.4 FIRSTOUT — First-Out Interlock Annunciator

When an interlock string drops everything stops at once; which permissive went first is the fault. FIRST captures the 1-based index of the first permissive to drop and holds it while any stays down.

Pin	Direction	Type	Meaning
IN1 ... IN16	In	BOOL	Permissives, TRUE = healthy
OK	Out	BOOL	Every bound permissive healthy
FIRST	Out	DINT	Index of the first to drop; 0 when healthy

Fill the slots contiguously from IN1: the block counts permissives up to the highest bound slot, so a gap reads as a dropped permissive and trips at the gap. Ties within one scan go to the lowest index. It self-clears; keep a record with an alert on FIRST.

Example: IN1 `e_stop_ok` , IN2 `guard_closed` , IN3 `air_ok` , OK
`line_permissive` , FIRST `first_out_idx` bound to a numeric display.

19.10 Sequencing and time

19.10.1 DRUM — Step Sequencer

A table of up to 64 steps, each with a 16-bit output pattern and a time. STEP is the current row (1-based, 0 = idle); PAT is that row's pattern — wire it into an UNPACK to drive coils. The step table is edited in the block's Properties panel.

Pin	Direction	Type	Meaning
IN	Rung	BOOL	Run; false pauses the timer, holds STEP and forces PAT to 0
Mode	Setting	time / event / both	What advances a step
EV	In	BOOL	Event advance (rising edge)
RST	In	BOOL	Back to idle
JOG	In	BOOL	Manual advance (rising edge)
Loop	Setting	checkbox	Wrap to step 1 after the last
Step tag (STEP_IN / STEP)	In/Out	DINT	The current step, in a tag you nominate
PAT	Out	DINT	Output pattern of the current step
DONE	Out	BOOL	Last step finished
ET	Out	DINT (ms)	Time in the current step
PT	Out	DINT (ms)	The current step's time

Each scan while running, in priority order: RST goes idle; a rising JOG advances; in event or both mode a rising EV advances; in time or both mode $ET \geq PT$ advances. A step time of 0 means no time limit — the step waits for EV or JOG even in time mode. After the last step, Loop wraps to step 1 with DONE pulsed for one scan; otherwise STEP holds with DONE true until RST or IN drops. Make the step tag VAR_RETAIN and the sequence resumes after a power cycle.

Example: a filter backwash — drain (event: low float), air scour 180 s, backwash 600 s, settle 60 s, filter-to-waste 300 s, return — with PAT feeding an UNPACK whose bits drive five VALVE blocks.

19.10.2 RTC — Wall Clock

Reads the controller's local time once per scan, so every block in a scan sees the same instant. Outputs hold while the network is false.

Pin	Direction	Type	Meaning
EN	Rung	BOOL	Update the outputs
VALID	Out	BOOL	The clock is trustworthy
YEAR, MONTH, DAY	Out	DINT	Date (month 1–12)
DOW	Out	DINT	Day of week, 0 = Sunday
HOUR, MIN, SEC	Out	DINT	Time of day
TOD	Out	DINT	Minutes since midnight, 0–1439
DOY	Out	DINT	Day of year, 1–366

VALID is true only when the year is plausible and the clock is synchronised — from the network or a battery-backed clock the controller has been told to trust. Alarm on VALID: a schedule must never run on an unset clock.

Example: TOD bound to `clock_tod` for HMI display and day-of-week gating in plain ladder.

19.10.3 SCHED — Time-of-Day Window

Q is true while the local time is inside the ON_TOD to OFF_TOD window on an enabled day. It is derived, not edge-triggered: a power-up at 03:00 inside a 22:00–06:00 window gives Q at once.

Pin	Direction	Type	Meaning
EN	Rung	BOOL	Enable

Pin	Direction	Type	Meaning
ON_TOD, OFF_TOD	In	Number	Window start and end, minutes since midnight
DOW_MASK	In	Number	Bit 0 = Sunday ... bit 6 = Saturday; 0 or unwired = every day
HOLD	In	BOOL	Force Q off (operator suppress)
Q	Out	BOOL	Inside the window now
ACTIVE_DAY	Out	BOOL	Today is in DOW_MASK
NEXT_MIN	Out	DINT	Minutes until the next transition; -1 when unknown

ON later than OFF is an overnight window (22:00 to 06:00 is 1320 to 360) and the day test applies to the start day. SCHED needs a trusted clock: while RTC's VALID is false, Q is 0 and NEXT_MIN is -1. Bind ON_TOD, OFF_TOD and DOW_MASK to VAR_RETAIN tags and operators edit the schedule with numeric inputs on the panel; several windows a day are several SCHED blocks in parallel.

Example: Q `backwash_window` as a permissive into a DRUM's IN; an off-peak fill window; a chemical-delivery lockout.

19.11 Program flow

19.11.1 PYFB — Python Function

Calls a `@pyfb` function in the project's `app.py` on each rising edge of the network and delivers the return value to OUT. The Python runtime does the work outside the scan, so a slow or missing function can never stall the program.

Pin	Direction	Type	Meaning
IN	Rung	BOOL	Rising edge sends one request
Python function	Setting	text	The <code>@pyfb</code> name in <code>app.py</code>
A1 ... A4	In	Number	Arguments; only bound pins are passed, in order
TO	In	Number (ms)	Reply timeout; blank = 1000
OUT	Out	Number	Result, typed by the tag bound here
DONE	Out	BOOL	True from the reply until the next request
ERR	Out	BOOL	Timeout, exception or missing function
ECODE	Out	DINT	1 timeout, 2 the function raised, 3 no function of that name, 5 program/runtime mismatch

On any error OUT keeps its last good value. A round trip is about 20 ms, so use ladder math for per-scan calculations. The Python side is in *Python on the Controller*.

Example: function `calc_dose`, A1 `flow_gpm`, A2 `sp_target_ppm`, OUT `dose_rate` (REAL), DONE `dose_valid`.

19.11.2 CALL, RET

CALL runs the named function when its network is true and the caller continues with its next network afterwards. RET, when its network is true, returns from a function to its caller, or ends the current scan of a task. Neither has pins or outputs; the function shares the project's tags with its caller. A function cannot call itself; see *Function Blocks, Faceplates and Tasks*.

19.12 Common traps

- **An instance name is not a tag.** Bind an output pin to a tag to observe a value; an output with no tag shows `--` and looks like a dead block.
- **Unwired inputs read 0.** For MOTOR's FTS_T, VALVE's PT and DEBOUNCE's PT that means "supervision disabled", never "trip immediately".
- **Accumulators live in tags.** TONR, TOTAL, MOTOR's counters and DRUM's step keep their value in the tag you nominate; make it VAR_RETAIN, and use one tag per block.
- **Float outputs need REAL tags.** SCALE, FILTER, ROC and PID write floating-point bits; an INT or DINT tag there holds garbage.
- **One-scan pulses are invisible in monitoring.** Feed P/N contacts and R_TRIG/F_TRIG into a counter or latch to see them.
- **Set-coil alarms need a latched alert rule.**
- **Divide by zero returns 0** rather than faulting.
- **Tick Clamp on SCALE** for every transmitter.

20 Historian and Trends

A Nexus.io Automation Controller keeps its own history of the tags you choose, on its internal storage, for a configurable number of days. The **Trend Chart** widget on the panel draws that history. A WhiskerHMI PC application keeps a second history on the PC for the tags it reads, and offers a **Trends** dialog and CSV export. This chapter explains what is recorded, where, how to turn it on, and how to look at it.

20.1 Two historians

	On the controller	On the WhiskerHMI PC
Records	tags with Hist ticked, plus any tag on a Trend Chart	every tag the PC project reads
Sample period	1 s	1 s
Filtering	deadband and heartbeat (below)	none
Kept for	History Retention (days) of the controller project	History Retention (days) of the WhiskerHMI project
Viewed with	Trend Chart widget on the panel	Trends dialog, Trend Chart elements on PC screens, CSV export

The two are independent: the PC does not read the controller's history, and the panel cannot export CSV. If you need a file, the PC application is where to export it.

20.2 Recording history on the controller

20.2.1 Choosing the tags

Open the **Tag Database**. The **Hist** column shows a timeline icon per tag; click it to toggle. The tooltip reads *Kept in the controller history (tap to stop)* or *Not historized (tap to keep 60-day history)*. Tags used by a Trend Chart element on any screen are recorded whether or not Hist is ticked, so a trend never comes up empty.

In the Cloud edition a selected tag's Properties panel also shows a **History (on-controller, 60-day)** switch and, when it is on, **History Deadband (0 = every change)**. Leave the deadband at 0 for setpoints and BOOL tags; set it to the noise band of an analog value (for example 0.5 for a level in percent) to keep the stored history small.

20.2.2 Retention

In Project Properties, under **HMI Settings**, set **History Retention (days)**. The default is 60. Press Enter to commit the value. The field explains itself: tags with History checked in the tag table, plus any tag on a Trend Chart, are kept on the controller this long.

20.2.3 What is stored

The historian reads the selected tags once a second and keeps a sample when the value has moved by more than the deadband since the last stored sample, or when 60 seconds have passed without one (a heartbeat, so a flat line still has points). BOOL tags are stored as 0 and 1. Alongside the raw samples it keeps a per-minute minimum, maximum and average for every tag.

Raw samples are kept for 7 days; the per-minute summaries are kept for the full retention period. Old data is purged every hour, and a size guard trims the oldest raw days early if the database grows past its limit. Because summaries are only one row per tag per minute, a plant with a few dozen historized tags uses well under a gigabyte for 60 days. Retained history survives a program update and a power cycle; at most the last second of samples is lost on power failure.

History is configured on the controller by the same deploy as the program: connect and press **Build (F5)**; the IDE compiles and downloads the program together with the historian configuration. Changing the tag set or retention takes effect on the next deploy; tags that stay selected keep their existing history.

20.3 The Trend Chart widget

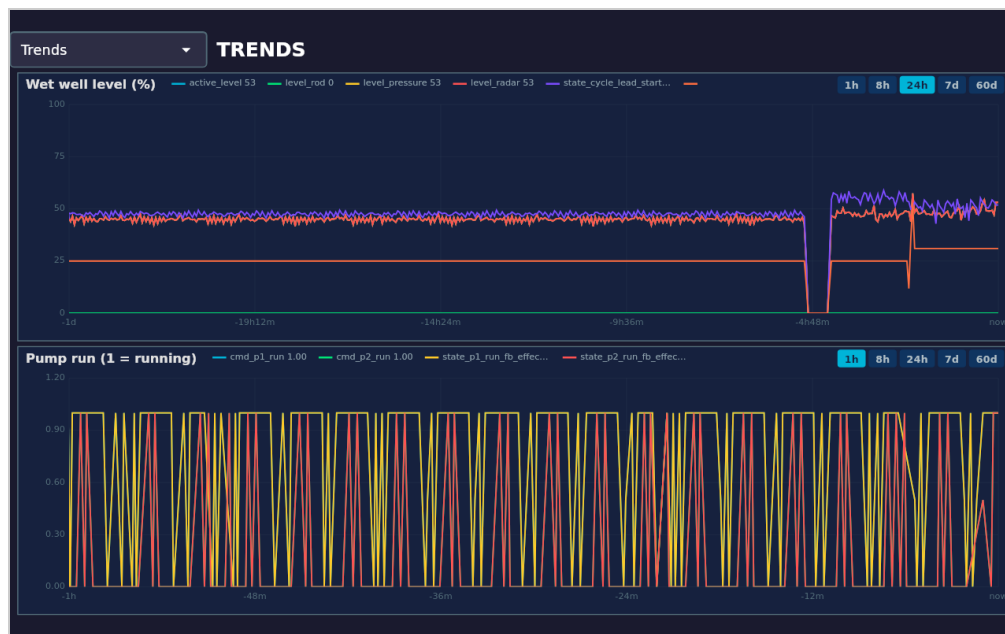
Place a **Trend Chart** (Indicators group) on a screen and set its properties:

Property	Meaning
Title	Caption above the plot (default Trend)
Tags	The tags to plot; add up to six (search and click to add chips)

Property	Meaning
Time Range (s)	The window shown when the screen opens, 10 s to 7 days (default one day)
Refresh (ms)	How often the widget re-queries the historian (default 2000)
Y Min, Y Max	Fixed vertical scale; leave both empty for auto-scale

On the panel the chart shows a legend with each tag's latest value and a set of range chips — **1h**, **8h**, **24h**, **7d**, **60d** — that the operator taps to change the window; the widget's Time Range is only the starting point. Long windows are drawn from the per-minute summaries, short ones from raw samples. If the chart reads *No history yet*, the tag has just been added and the first samples have not arrived; *No tags configured* means the Tags list is empty.

In the designer and in HMI Preview the chart draws sample curves rather than data.



20.4 History on the WhiskerHMI PC

The PC application's I/O scanner writes every tag it reads, once a second, into a local time-series database that the installer ships with the application. Retention comes from the WhiskerHMI project's **History Retention (days)** (Project Properties, under the Authentication group; default 60) and is applied each time

the scanner starts. There is no per-tag selection or deadband on the PC side. See *The WhiskerHMI PC Application* for installation and login; the rest of this section describes the dialogs.

20.4.1 The Trends dialog

Click **Trends** in the status bar. The dialog has a searchable tag list on the left (*Search tags...*) and the plot on the right:

- Tick up to **six** tags; the counter reads **N / 6 selected** . Every tag the project reads is offered.
- Range chips **1 h**, **8 h**, **24 h**, **7 d**, **30 d**, **60 d** set the window; the plot refreshes every 2 s (every 10 s for windows over a day). Long windows are averaged into about 300 points.
- **Saved trends...** lists presets; **Save as preset** stores the current tags and range under a name you enter (*Preset name*), and **Delete preset** removes one. Presets are kept per Windows user and per project on that PC.
- **Export CSV** opens the export dialog pre-filled with the plotted tags and range.

The chart has no zoom or cursor; choose a shorter range to see detail.



20.4.2 Exporting history to CSV

Click **Export** in the status bar, or **Export CSV** in the Trends dialog. The **Export History to CSV** dialog asks for:

1. The tags — any number, with **Select all** and **None**.
2. **Time range** — chips **Last 1 h** to **Last 60 d**, or explicit **Start** and **End** date and time pickers.
3. **Resolution** — **Raw samples**, **1 second**, **10 seconds**, **1 minute** (default), **15 minutes** or **1 hour**. Coarser resolutions average the samples in each interval.

The dialog estimates the row count (*About N rows $\times$ M tags*). Click **Export...** and choose a file; the suggested name is `whiskerhmi_export_<date>_<time>.csv`.

The file has a `timestamp` column in local time (`YYYY-MM-DD HH:MM:SS`) followed by one column per tag, headed with the tag name and its units when known. An empty range still writes the header row and tells you so.

With operator login enabled, opening Trends needs the *View screens* permission, saving presets needs *Save trend presets*, and exporting needs *Export history*; every export is written to the audit log.

20.5 Alert history

Alarm events are not part of the tag historian. The controller keeps its own list of the last 1000 FIRED and CLEARED events, shown by the **Alert Table** widget on the panel and in the PC application; it has no time-based retention and no CSV export. See *Alerts Editor*.

21 The WhiskerHMI PC Application

WhiskerHMI is the Windows application the IDE generates from a WhiskerHMI project: the same screens you design for a Nexus.io Automation Controller, running in one or more windows on a PC and reading the controller live. This chapter describes it as an installer and an operator meet it — starting it, connecting, windows, the status bar, login, trends and the audit log. Building the project and producing the installer is covered in *Generating HMI Installers*; designing the screens in *HMI Designer*.

21.1 Starting the application

Launch it from the Start Menu or desktop shortcut created by the installer. The application starts its two helpers if they are not already running — the **I/O scanner**, which holds the connection to the controller, and the local history database — then opens the main window, plus any windows whose role is *Auto Open*. Closing the main window closes every other window and stops the helpers.

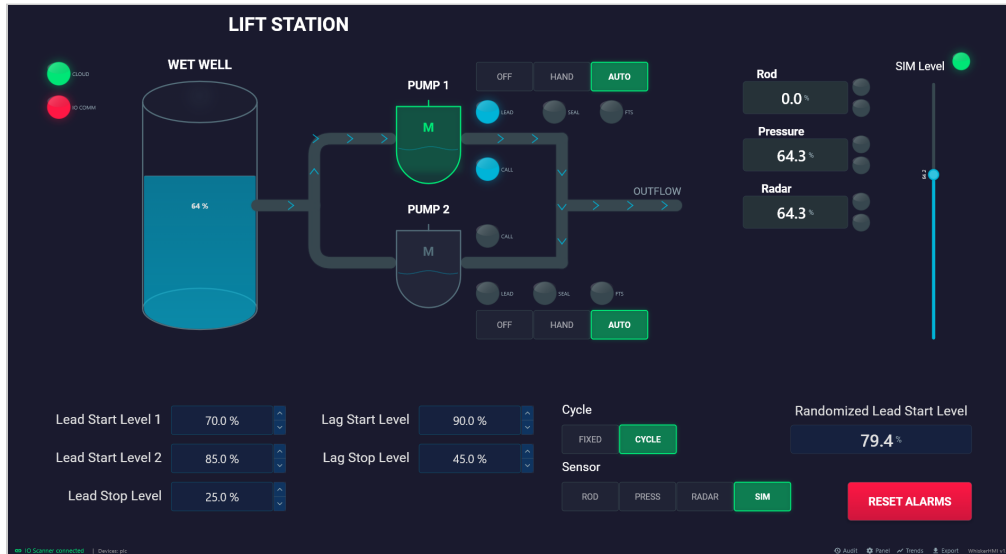
The application has no settings screen; its configuration is the compiled project (see *Generating HMI Installers* for where the files live and how to change the controller address).

21.2 Connecting to the controller

The I/O scanner connects to the controller named by the project's ArenaTCP device and keeps its tags in step. The status bar shows the state at its left end:

Status	Meaning
IO Scanner connected (green)	The application is talking to its scanner; the device list follows (Devices: controller)
IO Scanner disconnected — reconnecting...	The scanner is not answering; the application retries with a back-off of up to 10 s
No IO Scanner — offline mode	The configuration has no tag table; screens show defaults only

When the scanner is up but a controller is not, a dark red banner appears above the status bar: `<device> (<address>:<port>) – not connected`, or a count when several devices are down. It disappears when the connection returns; the check runs every 3 seconds. While a controller is unreachable, its tags read as zero and writes are dropped.



WhiskerHMI on a control-room PC showing the lift station, connected to its controller

21.3 Panel identity — connecting to a managed controller

If the project's ArenaTCP device has **Secure connection (NexusSec — mutual TLS)** ticked, the scanner opens a mutually authenticated TLS connection: it presents this PC's own certificate — its **panel identity**, issued to the station name `HMI-<PC name>` — and checks the controller's certificate against the CA chain shipped in the configuration. What the panel PC needs depends on the controller's posture (see *Security, Provisioning and Enrolment*):

Controller	What the panel needs
Standalone (how a new controller arrives)	Nothing. Leave <i>Secure connection</i> unticked; the panel connects plainly and the rest of this section does not apply

Controller	What the panel needs
Managed — enrolled into a Whisker.io account, or provisioned into a site CA	A panel identity issued by that authority. Its data port accepts only secure connections from a certificate the authority issued and has not revoked. The identity is issued once per panel PC from the IDE and imported on the panel PC itself; it is not part of the installer
A bench unit provisioned from the IDE's local keystore	Nothing to import: the IDE issues the station certificate from that keystore during the build and ships it in the configuration

21.3.1 What the build carries

When a WhiskerHMI project is built for a managed controller, the IDE decides the authority the same way it does for a deploy: the Whisker.io account you are logged in to, otherwise the project's **Site identity** (Project Properties). For either, the `security` folder of the compiled configuration holds only that authority's CA chain and revocation list, plus a marker file `PANEL_IDENTITY_REQUIRED.txt` naming the authority. Nothing is issued from the local keystore. The Output panel says so:

Panel identity: this build trusts cloud account N; import a panel identity package on each panel PC (Security dialog → Panels)

(or *site* for a site-managed project). A build without a revocation list on this PC adds a warning to refresh it in the Security dialog. Only a project that resolves to the local keystore still gets a station certificate minted at build time.

21.3.2 Issuing a panel identity

The engineer issues one identity per panel PC, in the IDE:

1. Open **Security** (the toolbar shield). For a cloud project, scroll to the Account section's **Panels (WhiskerHMI PC identities issued by this account)**; for a site-managed project, to the Site section's **Panels (WhiskerHMI PC identities issued by this site)**.
2. Click **Issue panel identity....** The dialog is titled *Issue panel identity — account N* or — and asks for:
 - **Panel PC name** — the Windows computer name of the panel PC (the hint reads *e.g. CONTROL-ROOM-PC-1*). Below it the dialog shows the resulting **Station name:** `HMI-` followed by the name in

upper case, with anything other than letters, digits and hyphens replaced by a hyphen. The name is a label for your records; the importer does not check it against the PC.

- **Validity (days)** — 365 by default, 1 to 3650. A cloud account issues in whole years and rounds the days up.
- **Panel passphrase (for the package)** and **Confirm panel passphrase** — at least eight characters. It protects the private key inside the package; the operator types it once, at import.
- **CA passphrase (this PC)** — site path only; the site's CA keys are encrypted with it.

3. Click **Issue...** and choose where to save the package. The suggested name is `HMI-<NAME>.<account or site id>.whisker-panel`.

The Output panel confirms *Issued panel identity HMI- ()*, *valid to*, the package path, and the instruction to copy the file to the panel PC and tell the operator the passphrase separately. The package holds the panel's certificate and private key (encrypted with the panel passphrase), the authority's CA chain and its current revocation list. Send the file and the passphrase by different routes, as you would an engineer identity.

PIN login credential. When the project has **Cloud PIN Login** switched on (Project Properties) and the identity is issued from a cloud account, the same action also mints the PC's PIN-login credential for `HMI-<NAME>` and adds it to the package; the confirmation then ends *with a PIN-login credential for this PC*. This is the PC's counterpart of the device credential a controller receives on every deploy, and it is what lets the panel register PIN logins and upload its audit log. It is never part of the installer, which is one file for every PC. The project must have been set up on the account you are logged in to; otherwise the IDE stops with *Wrong account for PIN login*. A site-managed project has no account, so a site-issued package carries no credential and its panel offers email-and-password login only.

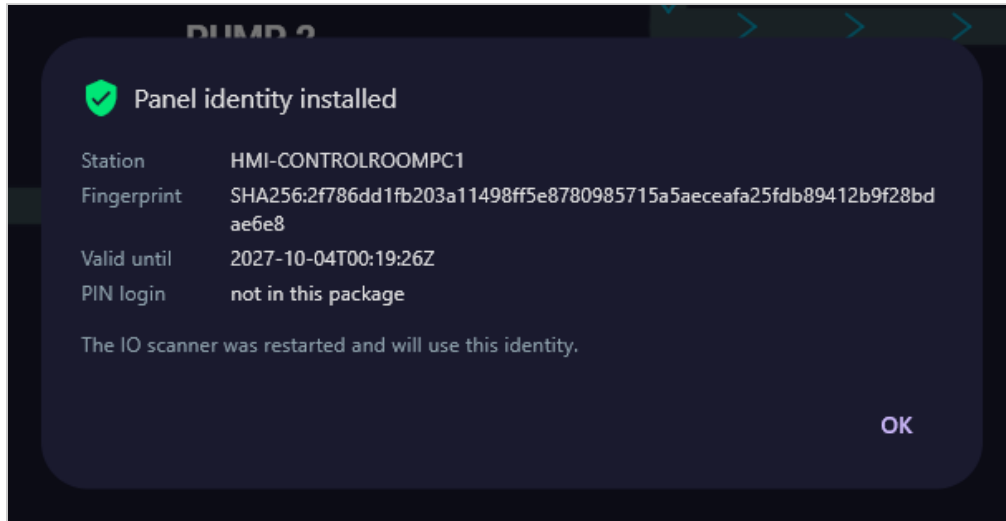
The **Panels** table records every identity issued: **Station**, fingerprint, expiry and status, with a **Revoke** icon per row; a station whose package carried a PIN-login credential is marked `[PIN]`. On the cloud path the table lists identities this IDE issued; on the site path it is the site ledger's panel rows.

21.3.3 Importing the identity on the panel PC

1. Install the WhiskerHMI application (see *Generating HMI Installers*) and start it. Until the identity is imported the dark red banner reports the controller as *not connected*.

2. Copy the `.whisker-panel` file to the PC (any folder; it is read once).
3. In the main window's status bar click **Panel** and choose **Import panel identity**.... The Panel menu is on the main window only, because that window owns the I/O scanner.
4. Pick the file. The **Import panel identity** dialog names the file and asks: *Enter the passphrase the engineer set when this package was issued.* Type it and click **Import**.
5. The I/O scanner checks the package before installing anything: the passphrase decrypts the key; the key matches the certificate; the certificate is within its validity dates and chains to the package's CA chain; and, when the configuration already carries a CA chain, that chain and the package's have the same root — the panel must be built for the authority that issued the identity. It then writes the certificate, the key (readable by the current Windows user only), the chain and the revocation list into the configuration's `security` folder and removes the `PANEL_IDENTITY_REQUIRED.txt` marker. A package that carries a PIN-login credential has it checked too — it must name this station — and installed as `hmi_auth.json` in the configuration folder, readable by the current Windows user only.
6. The result dialog reads **Panel identity installed**, with **Station**, **Fingerprint**, **Valid until** and **PIN login** (*credential installed or not in this package*), and the note *The IO scanner was restarted and will use this identity.* (or, if the restart failed, *Restart the IO scanner (or this panel) to use this identity.*). When a credential was installed the note adds *Restart this panel to enable PIN login with the new credential.* — the application reads the credential when it starts. On failure the dialog reads **Import failed** with the scanner's reason (below). Within a few seconds the banner clears and the controller is listed under *Devices*.

The import is written to the audit log as *Imported panel identity* with the station name and fingerprint as the target; the passphrase is never recorded. The identity stays on the PC across application updates and restarts; a newer installer adds and replaces configuration files but does not remove it.



21.3.4 Revoking a panel

A panel PC that is retired, re-imaged or no longer trusted has its identity revoked in the IDE: in the same **Panels** sub-section click the **Revoke** icon on its row and confirm (site path: enter the CA passphrase). The certificate goes on the authority's revocation list and the row's status shows *Revoked*. Controllers refuse the panel once they hold that list:

- **Cloud account** — enrolled controllers fetch the account's list every fifteen minutes, so the panel is refused within fifteen minutes.
- **Site CA** — the list is regenerated at once and travels with the engineers: the next time any engineer's IDE connects to a managed controller it pushes the newer list (or use **Push CRL to unit**). A controller nobody connects to keeps accepting the panel until then.

On the cloud path, revoking a panel marked [PIN] also revokes its PIN-login credential, so the PC can no longer register PIN logins or upload audit events; the confirmation says so. If that second step fails the message tells you to revoke the credential in the Whisker.io Control Panel.

A revoked panel cannot be reinstated. If the PC stays in service, issue a new identity and import it; the new one replaces the old on the PC.

21.3.5 Messages and what they mean

Message	Where	Meaning
<i>No panel identity installed on this PC — use Import panel identity... in the panel's menu (followed by the authority the build trusts, in brackets)</i>	The I/O scanner's connection error for the device	The configuration was built for a managed controller and no identity has been imported yet. Import one
<i>passphrase incorrect</i>	Import failed	The passphrase does not decrypt the package's key. Ask the engineer who issued it
<i>this panel was built for another authority</i>	Import failed	The package's root CA differs from the chain the configuration was built with — for example a site-issued identity on a panel built for a cloud account. Have the identity issued by the authority the controller trusts, or rebuild and reinstall the panel for the right one
<i>station certificate expired on / is not valid until</i>	Import failed	The identity is outside its validity; issue a new one (or check the PC's clock)
<i>station certificate does not chain to ca_chain.pem</i>	Import failed	The package is inconsistent; issue it again
<i>hmi_auth.json is for , not this panel</i>	Import failed	The package's PIN-login credential belongs to another station; the package was assembled wrongly — issue it again
<i>not a .whisker-panel package (zip unreadable) / package is missing ...</i>	Import failed	The file is damaged or is not a panel package
<i>IO scanner executable (WhiskerHmiService.exe) not found</i>	Import failed	The application's service folder is incomplete; reinstall

Message	Where	Meaning
The banner returns after a revocation	Runtime	The controller's revocation list now names this panel; import a newly issued identity

The scanner reports its connection error in its own output and the device is shown as *not connected* in the banner; the panel does not display the error text itself.

21.4 Windows and the layout editor

Each screen of a WhiskerHMI project can be a window. In the IDE, select the project node and open the **Window Layout** section of its properties (it appears only for WhiskerHMI projects). The table has one row per screen:

Column	Options
Role	Main (the primary window; exactly one), Popup (opened by the operator from the Windows menu), Auto Open (opens with the application), None (reached only by Nav Buttons and Page Selectors inside another window)
Monitor	Default, 1 (primary), 2, 3, 4 — which display the window opens on; a missing monitor falls back to the primary
Fill	Borderless and full-size on that monitor (the taskbar is covered), for kiosk-style panels

Windows without Fill open centred on their monitor at the project's display size and can be moved and resized like any Windows window; positions are not remembered between launches. The window title is the project name, or `<project> – <screen>` for secondary windows.

At runtime the **Windows** item in the status bar lists every Popup and Auto Open screen, with its monitor number when one is assigned; click one to open it. A window that is already open is not opened twice. Close a secondary window with its title-bar close button.

Each window runs as its own process, and an operator login belongs to the window it was made in: a popup opened from a logged-in main window asks for its own login when a gated control is used.

21.5 The status bar

Left to right:

- the connection status and device list (above);
- when logged in, the operator's name and **Log out**; otherwise **Log in** (when login is enabled);
- **Audit** — opens the audit log (requires the *Audit log and settings* permission);
- **Panel** — this PC's own settings; today its one item is **Import panel identity...** (main window only; see *Panel identity — connecting to a managed controller* above);
- **Windows** — the menu of openable windows;
- **Trends** and **Export** — the trend and CSV-export dialogs, present when the project ships history support;
- the version label.

There is no alarm banner: alerts are shown where the designer placed an **Alert Table** element (see *Alerts Editor*). On the PC the table is a live viewer of the controller's alert history; acknowledge, clear and reset made here do not reach the controller and are undone at the next refresh, so reset latched alerts at the panel or through a ladder-driven push button.

21.6 Trends and CSV export

Trends plots up to six tags over 1 hour to 60 days from the PC's local history and can save presets; **Export** writes any tags and time range to a CSV file at a chosen resolution. Both are described in *Historian and Trends*. History on the PC is recorded for every tag the project reads and kept for the project's **History Retention (days)**.

21.7 Operator login and roles

Login is optional and needs the **Cloud edition** of the IDE and a Whisker.io account: operators are the account's users, and roles are the account's roles. Configure it in the WhiskerHMI project's properties:

- **Security** — the switch that turns gating on, followed by a table of actions and the role each requires: View screens, Buttons and switches, Setpoints and sliders, Acknowledge alarms, Clear alarms, Save trend presets, Export history, Open windows, Audit log and settings. Each can be *No login needed* (viewing and acknowledging only), *Any logged-in user* or a named role. **Refresh roles from Whisker.io** reloads the role list.
- Individual buttons, switches, inputs, sliders, faceplates and Alert Tables can override their action's role with their own **Required role** (Properties panel, **Security** section).
- **PIN login at the console** — lets operators log in by picking their name and entering their Whisker.io PIN instead of typing an email and password.
- **Session Timeout (min)** — idle minutes before the operator is logged out (default 480). Any touch or click restarts the timer.
- **Offline grace (hours)** — how long a user who has logged in on this PC can keep logging in while Whisker.io is unreachable (default 24).

With the switch off, every control works for everyone; changes are still written to the audit log with no operator name.

21.7.1 What the operator sees

When *View screens* requires a login, the **Operator login** panel covers the window at start-up; otherwise the operator works unauthenticated and is asked to log in when they touch a gated control. The panel offers **Email & password** (with a verification **Code** step when the account uses two-factor authentication) and, when enabled, **PIN: a Who are you?** picker of the account's users and a keypad; the PIN is submitted automatically at its full length. A **Continue as viewer** button appears when viewing needs no login.

A wrong entry is answered with *Not recognised* and the number of tries left. After five wrong PINs that user is locked for five minutes, doubling on each further round up to 30 minutes; the message says so and other users can still log in. PIN login also requires the PC's own PIN-login credential, which arrives inside its panel identity package when the project has **Cloud PIN Login** on (see *Panel identity — connecting to a managed controller*); without it the panel says *PIN login not provisioned on this PC* and email-and-password login remains available.

Using a control the operator's role does not allow shows a short dark-red message — *Log in to setpoints and sliders or Requires Supervisor* — and nothing is written. **Log out** in the status bar, a **Lock Button** on a screen, or the session timeout ends the session.

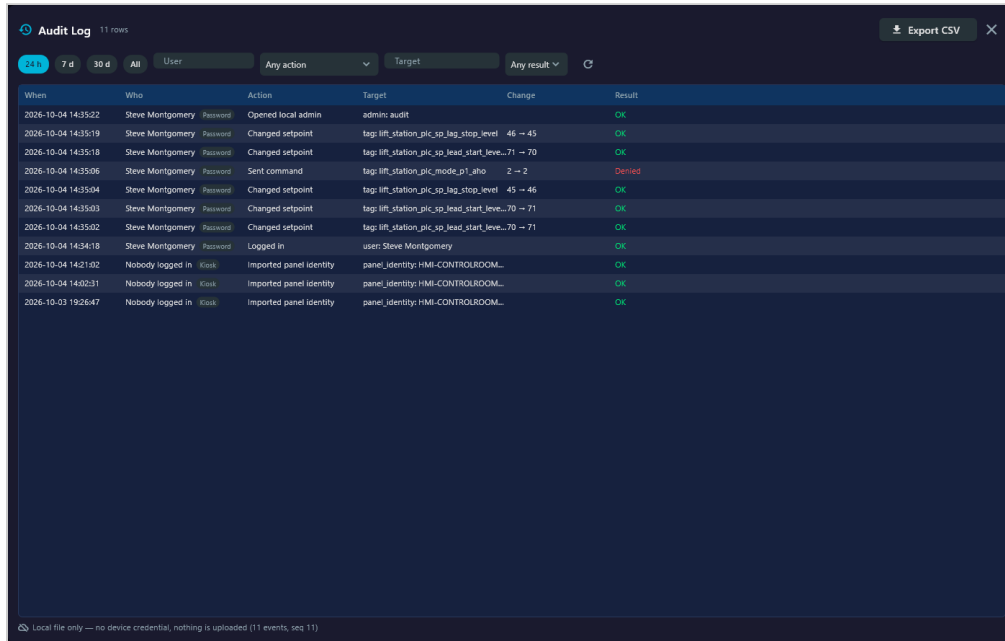
When Whisker.io cannot be reached, users who have logged in on this PC within the grace period can still log in from a local cache (the PIN picker is captioned *Offline — recent users only*). After the grace period expires the operator is told so and must wait for the connection to return.

21.8 The audit log

Every change of state on the PC is recorded locally: logins, failed logins, logouts, locks and timeouts; each button, switch, setpoint and slider write with the tag, the old and new value, the element and the screen; alarm acknowledgements and clears; preset saves; exports; and every denied action with its reason. Viewing screens and opening windows are not logged. Records are chained with a hash so that a removed or altered line is detectable, and are kept for 400 days in monthly files.

Click **Audit** in the status bar (requires *Audit log and settings*). The **Audit Log** dialog shows **When**, **Who** (with the login method), **Action**, **Target**, **Change** (old → new) and **Result** (OK, Denied or Failed), newest first. Filter by range (**24 h**, **7 d**, **30 d**, **All**), **User**, action group (*Sessions*, *HMI changes*, *Alarms* or a single action), **Target** and result. **Export CSV** writes the filtered rows to `whiskerhmi_audit_<date>_<time>.csv` with the full record including the chain hash.

The bottom line of the dialog reports whether events are also being uploaded to the Whisker.io account's audit trail (*Uploaded through seq N*) or kept *Local file only*. Upload requires the same PC registration as PIN login; an **Upload now** button retries immediately. The account-side audit trail is described in *Security, Provisioning and Enrolment*.



When	Who	Action	Target	Change	Result
2026-10-04 14:35:22	Steve Montgomery	Opened local admin	admin: audit		OK
2026-10-04 14:35:19	Steve Montgomery	Changed setpoint	tag: lift_station_plc_sp_tag_stop_level	46 → 45	OK
2026-10-04 14:35:18	Steve Montgomery	Changed setpoint	tag: lift_station_plc_sp_lead_start_level...	71 → 70	OK
2026-10-04 14:35:06	Steve Montgomery	Sent command	tag: lift_station_plc_mode_p1_aho	2 → 2	Denied
2026-10-04 14:35:04	Steve Montgomery	Changed setpoint	tag: lift_station_plc_sp_tag_stop_level	45 → 46	OK
2026-10-04 14:35:03	Steve Montgomery	Changed setpoint	tag: lift_station_plc_sp_lead_start_level...	70 → 71	OK
2026-10-04 14:35:02	Steve Montgomery	Changed setpoint	tag: lift_station_plc_sp_lead_start_level...	70 → 71	OK
2026-10-04 14:34:18	Steve Montgomery	Logged in	user: Steve Montgomery		OK
2026-10-04 14:21:02	Nobody logged in	Imported panel identity	panel_identity: HMI-CONTROLROOM...		OK
2026-10-04 14:02:31	Nobody logged in	Imported panel identity	panel_identity: HMI-CONTROLROOM...		OK
2026-10-03 19:26:47	Nobody logged in	Imported panel identity	panel_identity: HMI-CONTROLROOM...		OK

Local file only — no device credential, nothing is uploaded (11 events, seq 11)

21.9 Files on the PC

Location	Contents
C:\Program Files\WhiskerHMI\<ProjectName>\	The application and its helpers
C:\ProgramData\WhiskerHMI\<ProjectName>\config\	The compiled project configuration; its security folder holds the CA chain and revocation list from the build and, once imported, the panel identity
The Windows user's application-data folder, WhiskerHMI\<ProjectName>\	Trend presets, the offline login cache and the audit folder
The user's temporary folder, whisker_hmi.log	The application log from the current launch, the first place to look when a window will not open

22 The AI Designer and AI Fix

The AI Designer, which drafts an Application from a written description and diagnoses failed builds, will be available in a future release of Whisker IDE.

23 CIP Tag Server

A Nexus.io Automation Controller can serve its tags **by name** over EtherNet/IP, the protocol Allen-Bradley controllers use. A SCADA driver that talks to a CompactLogix — AVEVA's ABCIP driver, for example — connects to the controller, discovers the published tags for itself, and reads and writes them under the names you gave them in the Tag Database. There are no register numbers to assign and no map to hand over. This chapter covers turning the server on, choosing which tags it publishes and which a client may write, the rules it enforces, and how to set up a SCADA driver against it.

Note. This is the controller as an EtherNet/IP *server*. Reading tags *from* an Allen-Bradley PLC, where the controller is the client, is an **EtherNet/IP PLC** device in *I/O Configuration and Devices*. The two are independent and can run together. The Modbus TCP server (*Modbus Server and Map Viewer*) also runs alongside; a tag can be published on either, both or neither.

23.1 How the server works

The server listens on TCP port 44818, the standard EtherNet/IP port, and answers the services a Logix-style driver uses:

- **Session and identity.** A client registers a session, can ask the controller who it is (the Identity Object and *List Identity*), and can read the controller name. The controller identifies itself honestly, as a Nexus.io controller with its own model and serial number, not as an Allen-Bradley product.
- **Tag browsing.** A client lists every published tag with its name and data type, so a driver that uploads the tag database — as ABCIP does — gets the list from the controller itself.
- **Reads and writes by name,** singly or many in one request, using both the plain and the fragmented tag services. A single BOOL is set and cleared with the masked write Logix clients use for bits.
- **Connected and unconnected messaging.** A driver may open a connection for its cyclic reads or send each request on its own; both are served.

Published names are the tag names, unchanged, except that a space becomes an underscore. There is no program scope and no prefix: the tag `active_level` is `active_level` to the client. Names are matched without regard to case, so a

driver that sends `ACTIVE_LEVEL` gets the same tag. A Logix name is letters, digits and underscores, not starting with a digit, at most 40 characters; a published tag whose name breaks that rule is left out of the map and the build reports it as a warning.

Values come from the controller's live data, refreshed every 50 ms, so a client reading fifty tags costs the controller no more than one reading one. If the controller's data layer stops responding the server refuses reads rather than serve a stale value; a SCADA showing a frozen number as though it were live is worse than one showing a fault.

Tag type	On the wire	Bytes
BOOL	Logix BOOL	1
INT	Logix INT	2
DINT	Logix DINT	4
REAL	Logix REAL, IEEE 754 single precision	4

Only these four scalar types exist in the Tag Database, so there are no arrays or strings to configure on the client side.

A tag may be published under a **CIP Name** that differs from its Tag Database name (the CIP section of the tag's Properties). A CIP Name with a dot in it, such as `CSLS19PMP00108.oRun`, is presented the way a Logix controller presents a member of a user-defined structure: the client browses a structure `CSLS19PMP00108`, reads its template to learn the member `oRun` and its type, and reads or writes the member by that path. That is how a SCADA whose standard names are `Equipment.Signal` reads them without any renaming on its side. Every member is still one of the four scalar types above.

The map of published tags is compiled with the program: connect and press **Build (F5)**; the IDE compiles and downloads the program, and the server restarts with the new map. A driver that keeps its own copy of the tag list notices the change and fetches the list again.

23.2 Turning the server on

The server is **off in every new project**. Select the project node in the Project Tree and find the **CIP tag server** section of Project Settings. Its switch reads **Off**, or **On — tags are served by name on TCP 44818**. While it is on the section counts how many of the project's tags are published.

Warning. CIP carries no authentication and no encryption. Anyone who can reach port 44818 on the controller can read every published tag and write every writeable one. Leave the server off on a controller reachable from an untrusted network, and publish only the tags the SCADA needs. {.warning}

Turning the server on does nothing by itself: a project with the server on and no tags published serves nothing.

23.3 Publishing a tag

Tags are published one at a time. Open the **Tag Database**; when the server is on, a **CIP** column appears next to **Hist** with a network icon per tag — outlined when the tag is not published, filled when it is. Click the icon to publish or withdraw a tag. Hover it to see whether a published tag is read-only or writeable.

Select a tag and open the **CIP tag server** section in its Properties panel:

- The first switch publishes the tag. Beside it the panel reads *Published as "name" or Not published*.
- **CIP Name**, shown only for a published tag, is the name a client uses instead of the tag's own. Leave it blank to publish the tag under its name (the box shows that name greyed). Type a Logix name, or a member path such as `CSLS19PMP00108.oRun`, to follow a SCADA's naming scheme; the *Published as* text changes to match when you leave the box. Each part between dots is letters, digits and underscores, not starting with a digit, at most 40 characters; anything else is refused with a message saying so. Two tags with the same CIP Name, or a CIP Name equal to another published tag's name, are reported as a warning at build time and the second is left out.
- The second switch, shown only for a published tag, decides whether a client may write it: *A CIP client may write this tag or Read-only to CIP clients*. New tags are published read-only; make a tag writeable only when the SCADA has to change it.
- If the server is off for the project, the section says so and points you to Project Properties. The settings are kept and take effect when the server is turned on.



PROPERTIES

sp_high_alarm_level
REAL %MR5

Data Type
REAL (4 bytes) ▼

Variable Class
VAR_RETAIN ▼

Source
Source
Local (this controller) ▼
To read a tag from another controller or an Allen-Bradley PLC, add that device first in IO Config.

Initial Value
90

Description
High level alarm setpoint (%)

Modbus Server
☒ Expose via Modbus
HR 2012
2012 / 42013

CIP tag server
☒ Published as "CSLS19XW00120.iHiOn"
CIP Name CSLS19XW00120.iHiOn
☒ A CIP client may write this tag

Process Variable

23.4 What a client may and may not write

The server enforces three rules, whatever the client asks:

1. **A read-only tag is never written.** The client gets a privilege violation, the value does not move, and the controller's log names the tag.
2. **Physical I/O is never written**, even if you try to mark it writeable. The writeable switch is disabled for an input or output tag with the note *Read-only — physical I/O is never writeable over CIP*, and if a project file marks one writeable anyway the compiler publishes it read-only and reports it as a warning at build time. A SCADA that needs to command an output writes a memory tag and the ladder decides what reaches the terminal, exactly as an operator at the panel would.
3. **A tag that is not published cannot be read or written at all.** A client asking for an unpublished name gets an error, not a value.

A write to a writeable memory tag changes the value on the controller at once and the change survives the next refresh. A field input the ladder overwrites every scan will, of course, be overwritten again.

23.5 Setting up an AVEVA ABCIP driver

The server has been tested against AVEVA's ABCIP Operations Integration Server with InTouch. Other drivers that read a CompactLogix by tag name should work the same way but have not been tested. In the ABCIP configuration:

1. Add a **PORT_CIP** port, under it an **ENB_CLX** Ethernet node with the controller's address, under that a **BACKPLANE_CLX** and a **LOGIX5000_CLX** device at slot 0. The controller answers at slot 0.
2. On the Logix5000 device set **Optimization Mode** to **No optimization**. This is the one setting that matters. The default, *Optimize for read*, has the driver build read blocks inside a real Logix controller, which this server does not provide; with it selected every read fails with *Requested service not supported*. *No optimization* reads each tag by name and batches the requests, which is what the server is built for.
3. Leave **Auto Load Tags on Startup** and **Use Persisted Tags** on. The driver uploads the tag list from the controller on first connection and keeps a copy keyed to the controller's serial number; when you deploy a program with different published tags it notices and uploads again.

4. Add a device group (topic) with the update interval you want, and in InTouch an access name that points at it. Item names are the tag names.

An InTouch tag for a BOOL is an I/O Discrete, for an INT or DINT an I/O Integer, for a REAL an I/O Real. Set the tag's raw range equal to its engineering range unless you want InTouch to scale the value; a mismatch between the two shows up as a wildly wrong number, not as an error.

An InTouch tag *name* cannot contain a dot, so a CIP Name like `CSLS19PMP00108.oRun` cannot be the tag name in a standalone InTouch application. Name the InTouch tag `CSLS19PMP00108_oRun` and put the dotted name in its **Item**: the item is what ABCIP sends to the controller, so the controller still sees the exact name. A System Platform application, whose object attributes are written `Object.Attribute`, references the dotted name directly.

For more than a handful of tags, load them in bulk. Write a DBLoad file (a CSV with one row per tag: name, type, access name, item name, read-only) and load it from the **InTouch Application Manager**: select the application and choose **DBLoad**, with WindowMaker closed. Newer versions of WindowMaker have no DBLoad of their own. Mark only the tags you published writeable as read/write in InTouch; the controller refuses the others anyway (see *What a client may and may not write*).

23.6 What a refused write looks like

When a SCADA operator pokes a value the server refuses — a read-only tag or physical I/O — the poked value shows on the screen for one update and then snaps back to the controller's value. ABCIP records the refusal in the AVEVA logger as a write error with status `0F`, and the controller's own log names the tag and the reason. The driver's error item on the access name does not change. If an operator reports that a value “won't take”, that is what happened, and the tag's writeable switch is the place to look.

23.7 When the SCADA cannot connect

- The driver must reach the controller's address on TCP port 44818. The controller's own firewall allows it on the Ethernet port; check any firewall between the two.
- The server only runs when the project turns it on and the program has been downloaded since. Turning the switch on in the IDE does nothing until the next Build (F5).

- If the driver connects but every read fails, check **Optimization Mode** on the ABCIP device: it must be *No optimization*.
- If the driver reports the tag list is empty, no tags are published. The Project Settings section shows the count.
- After the controller restarts, or a new program is downloaded, a driver that had a connection open loses it and reconnects on its own within its reply timeout. A message or two logged as timed out at that moment is normal.
- An unpublished item name is rejected when the driver validates it, before it ever reaches the controller: ABCIP logs *not defined in processor*.
- If every item suddenly fails and the AVEVA log says *Unable to add item ... , license not valid*, the OI Server is running in demo mode and its two-hour demo period has run out. Restart the ABCIP service, then WindowViewer. A licensed server does not do this.
- Deploying a new program needs nothing on the AVEVA side: the driver sees the new tag list within seconds, uploads it again and carries on.

See *Troubleshooting* in the appendices for more.

24 The Walkthrough scenario

The next three chapters walk you through three end-to-end projects that share one physical setup. Building the same control system three different ways — once in ladder logic, once in Python, once as a separate HMI panel on a PC — is the fastest way to see what each editor in the Whisker IDE actually does and how the pieces fit together. The shared scenario also gives you something concrete to keep on your bench: at the end of the three walkthroughs, you'll have one running controller and one running HMI panel that you can exercise by toggling float switches and watching the system respond.

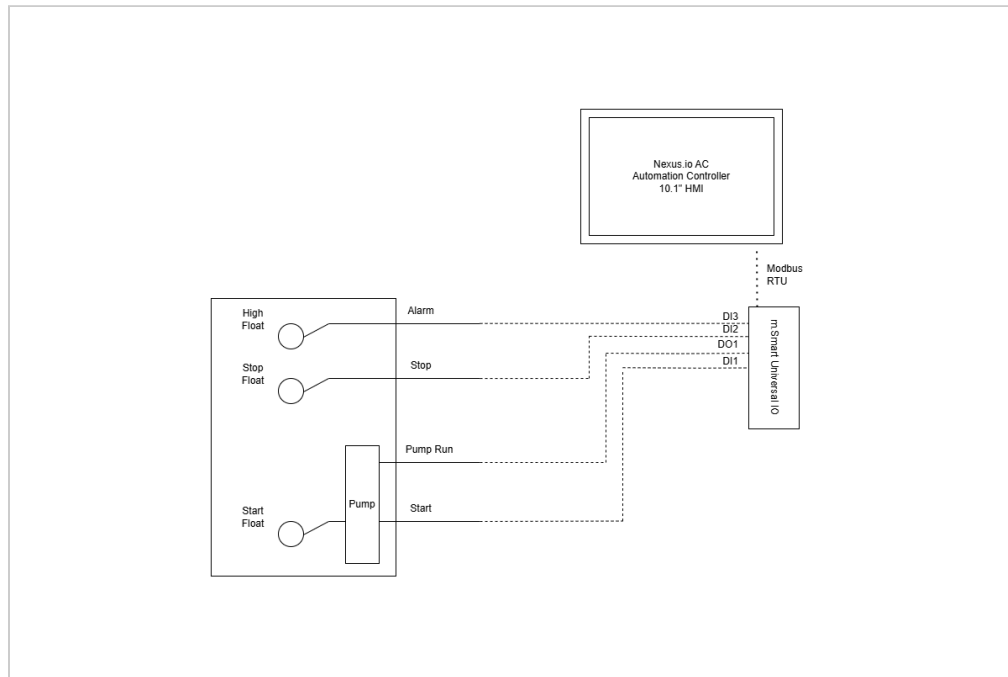
This chapter sets up everything the walkthroughs assume — the hardware, the field wiring, the tag names, the control intent, and the HMI layout. Read it once. Each walkthrough then refers back here for setup details rather than repeating them.

24.1 What you're building

A single motor controlled by three float switches and an operator selector. The operator chooses one of three modes — **Off**, **Hand**, or **Auto** — using an AHO selector switch on the HMI. In Auto mode, the motor runs when the **start** float closes and stops when the **stop** float opens, like a classic seal-in. In Hand mode, the motor runs unconditionally; the operator owns it. In Off mode, the motor is off.

A separate **high-alarm float** acts as a safety. When it closes, the controller latches an alarm and refuses to run the motor in Auto until the operator presses a **Clear Alarms** button on the HMI. In Hand mode the operator can still run the motor through the alarm — that lets them pump down or service the system without fighting the alarm latch. Off still means off, alarm or no alarm.

That's the whole scenario. Four inputs (three floats plus an operator mode selection), one output, one latched alarm, one operator panel.



System diagram — tank, three floats, pump, mSmart Universal IO, Nexus.io controller

The diagram above shows the physical setup. The three float switches (Start, Stop, High) sit at three heights inside the tank. The pump drives the tank's discharge. All four field signals wire into the **mSmart Universal IO** module on inputs DI1, DI2, DI3 and output DO1 — those module channels become the tags `float_start`, `float_stop`, `float_alarm`, and `motor_run` after you rename them in Step 3 of each walkthrough. The Nexus.io Automation Controller talks to the mSmart module over **Modbus RTU** on its RS-485 port; the wiring map below spells out which DI/DO maps to which tag.

24.2 Choose your path

A new Nexus.io controller arrives ready to use: plug it into your LAN and the IDE finds it as a **Standalone** controller that accepts a plain connection and takes whatever program you build. How you work with it from there depends on one question — is this controller going to be connected to the Whisker.io cloud? Every walkthrough step that depends on the answer is marked **Path A** or **Path B**. Everything else — tags, logic, screens, testing — is the same.

Path A — Cloud. You use the Cloud edition of the IDE and have a Whisker.io account. In the Cloud Explorer you provision the controller as a device in your account; the IDE hands it its enrolment bootstrap and the controller fetches its own certificate from your account's certificate authority, then restarts in **managed** mode. From then on the IDE connects to it securely with the certificate the cloud

issued to you at login, only signed programs from your account are accepted, and its data flows to your account. A cloud-connected controller is always managed — enrolment does that for you. See *Security, Provisioning and Enrolment* for the full story.

Path B — No cloud. You run the IDE without an account. Connect to the controller by discovery or by address and deploy; nothing else is required. Whether the controller should also be *managed* — every engineer with their own identity, only signed programs accepted, revocation when someone leaves — is your call. If you want that without a cloud account, one PC at your site becomes the site's certificate authority (Security dialog → Site section → **Make this keystore a site...**) and provisions the controller with **Provision unit (site)**; engineers import an identity package from that PC. You own that certificate authority; D6 keeps no copy of it. See *Security, Provisioning and Enrolment*. None of the walkthroughs needs it.

Tip. If you have an account, do Path A's provisioning before Step 8 of the first walkthrough — it is one dialog. On Path B there is nothing to do in advance. {.tip}

Steps 1–7 (creating the project, tags, logic and screen) need no controller at all — and if you have no controller yet, **Build & Emulate** runs the finished project on your PC: the IO Stimulus tab lets you flip the float switches by hand and the HMI Preview tab shows the screen (see *Connecting, Deploying and Emulating*).

24.3 The hardware

You need three things:

- **A Nexus.io Automation Controller** on your LAN, powered on (provisioned into your account first if you are on Path A).
- **An mSmart Universal IO module** wired to one of the controller's RS-485 ports as a Modbus RTU slave.
- **Field wiring:** three float switches on three digital inputs, one motor contactor (or a stand-in like an indicator lamp) on one digital output.

If you don't have a real motor handy, a 24 V relay or a single LED on the DO will do — you just need to see something react when the logic turns the output on.

24.3.1 Wiring map

Field signal	Module channel	Modbus register	Tag name (after rename)
Start float	DI1	Holding reg 0	float_start
Stop float	DI2	Holding reg 1	float_stop
High-alarm float	DI3	Holding reg 2	float_alarm
Motor contactor	DO1	Holding reg 99	motor_run

The mSmart Universal IO module exposes a lot more than this — six DIs, five AIs, an AO, plus counters and event-rate readbacks — but the walkthroughs only use four channels. Anything you don't reference in the project stays unmapped and ignored.

24.3.2 RS-485 settings

The demo mSmart module talks Modbus RTU at the following settings. Both ends of the RS-485 link must agree, or no data flows.

Setting	Value
Baud rate	9600
Data bits	8
Parity	None
Stop bits	1
Slave ID	1
Controller serial port	rs485 (UART 1)

Set the baud rate to 9600 on the controller. The demo module runs at **9600**, not the 19200 some controllers ship with — if the rate is wrong you'll add the device fine but read all zeros. The controller's RS-485 settings are described in *Setting Up Your Nexus.io Automation Controller*. The IDE-side device picks up its rate from the same `rs485` port, so the two stay in sync once the controller is set.

24.4 Tags — where they come from

The walkthroughs use the twelve tags below — **four** I/O tags and **eight** memory tags. (The Python walkthrough skips `const_hand`, so it uses eleven.) They come from **two different places** in the IDE, and getting the order right matters:

- **I/O tags** are created automatically by the IO Configuration editor when you add a hardware module. You don't (and can't) type a `%I` or `%Q` address into the Tag Database — those addresses belong to physical I/O channels and the IDE assigns them based on which device they belong to.
- **Memory tags** are created in the Tag Database. These are the internal bits and words your logic uses to remember state — modes, latches, button pulses — that don't correspond to a physical wire.

So the setup sequence in each walkthrough is:

1. Add the mSmart Universal IO module in the IO Configuration editor.
2. The IDE auto-creates one tag per channel on the module (`uio_DI1`, `uio_DI2`, ..., `uio_D01`, ...). Rename the four you care about to friendlier names.
3. Open the Tag Database and add the eight memory tags that have no physical counterpart.

Name	Type	Address	Origin	What it is
<code>float_start</code>	BOOL	<code>%I0</code>	auto from <code>uio</code> device, renamed from <code>uio_DI1</code>	Start float input
<code>float_stop</code>	BOOL	<code>%I1</code>	auto from <code>uio</code> device, renamed from <code>uio_DI2</code>	Stop float input
<code>float_alarm</code>	BOOL	<code>%I2</code>	auto from <code>uio</code> device, renamed from <code>uio_DI3</code>	High-alarm float input
<code>motor_run</code>	BOOL	<code>%Q0</code>	auto from <code>uio</code> device, renamed from <code>uio_D01</code>	Motor output

Name	Type	Address	Origin	What it is
aho_mode	INT	%MW0	added in Tag Database	Operator mode: 0=Off, 1=Hand, 2=Auto
const_hand	INT	%MW1	added in Tag Database (initial value 1)	Constant 1, used as IN2 of the CMP block. The CMP inputs require tag references — this is how you spell the constant “Hand”
motor_off	BOOL	%MX0	added in Tag Database	True when aho_mode = 0 ; set by the CMP block
motor_hand	BOOL	%MX1	added in Tag Database	True when aho_mode = 1 ; set by the CMP block
motor_auto	BOOL	%MX2	added in Tag Database	True when aho_mode = 2 ; set by the CMP block
call_motor	BOOL	%MX3	added in Tag Database	“Tank needs the motor” — set by start float, reset by stop float
alarm_latched	BOOL	%MX4	added in Tag Database	High-alarm latch (set by start-of-alarm float, cleared by button)

Name	Type	Address	Origin	What it is
clear_alarms	BOOL	%MX5	added in Tag Database	Clear-alarms button (one-shot, set by HMI, cleared by logic)

A few conventions to know:

- **%I and %Q addresses** map directly to physical I/O channels — %I0 is the first input bit, %Q0 is the first output bit. The I/O scanner fills %I from the module each scan and writes %Q back out. The exact address numbering depends on how many devices and channels precede this one; the IDE handles the bookkeeping.
- **%MX addresses** are internal memory bits. They have no physical counterpart; they're storage that lives in arena memory and survives between scans.
- **%MW0** is a 16-bit internal word, used here to hold the AHO mode as a small integer (0, 1, or 2).

The Whisker IDE uses these IEC-61131 address conventions for all PLC-style projects. The full address map is in the *Tag Database* chapter of the full manual.

The other auto-created tags. When you add the mSmart Universal IO module the IDE creates one tag for every channel on the device — roughly 30 tags. You only rename four. The rest stay in your project as untouched `uio_*` tags. They don't hurt anything; the I/O scanner reads them every scan whether you use them or not. If you want a tidier Tag Database, the IO Configuration editor lets you remove individual unused points.

24.5 The control logic — what it does

All three walkthroughs implement the same behavior. Reading this specification before diving into a walkthrough makes the editors a lot easier to follow.

The logic factors into five small steps. Each step is one ladder rung in the Ladder Walkthrough and a matching few lines of `app.py` in the Python Walkthrough:

```

(1) Mode decode – one CMP block compares aho_mode against const_hand
(=1):
    motor_off := aho_mode < const_hand    # LT output
(aho_mode = 0 → Off)
    motor_hand := aho_mode == const_hand  # EQ output
(aho_mode = 1 → Hand)
    motor_auto := aho_mode > const_hand    # GT output
(aho_mode = 2 → Auto)

(2) Alarm latch:
    if float_alarm: alarm_latched := TRUE

(3) Clear alarms (one-shot):
    if clear_alarms:
        alarm_latched := FALSE
        clear_alarms := FALSE    # auto-release the button

(4) call_motor latch (Auto-mode seal-in):
    if float_start: call_motor := TRUE    # tank wants the
motor
    if float_stop:  call_motor := FALSE    # tank doesn't
anymore

(5) Motor output:
    motor_run := motor_hand
                OR (motor_auto AND call_motor AND NOT
alarm_latched)
  
```

A few points worth flagging because they catch people out:

- **Why a CMP block instead of three separate compares.** The Whisker IDE has single comparison contacts (EQ, NE, GT and so on) and also a CMP block that emits LT / EQ / GT outputs from one comparison. The block fits this problem perfectly: one CMP network produces all three mode flags at once, and the rest of the logic reads them like ordinary BOOLs. The same approach scales nicely if you ever add a fourth mode (just add another CMP).
- **Hand overrides the alarm.** Step (5) is just `motor_hand` ORed with the Auto branch — no alarm check on the Hand side. This is a deliberate safety choice (operator on Hand needs to be able to pump down a flooded tank). If your site needs the opposite, drop a `NOT alarm_latched` contact in series with `motor_hand` and you're done.
- **The seal-in lives on its own.** Pulling the start/stop latch out of the motor rung into dedicated `call_motor` networks makes both cleaner. The motor rung becomes a one-line summary of intent ("Hand, or Auto + call + no

alarm”), and the seal-in is two small networks (set on start float, reset on stop float) you can debug in isolation. The split into two networks is because the Whisker IDE allows only one coil per network — the Set and the Reset have to live on separate networks.

- `clear_alarms` is a **one-shot**, not a level. HMI button writes TRUE; logic clears the latch and immediately writes FALSE back to `clear_alarms`. If it weren’t one-shot, holding the button down while the alarm condition was still active would re-latch every scan and you’d never see it clear.

24.6 The alarm

A single alert definition in each project:

Field	Value
Name	HighLevelAlarm
Condition	<code>float_alarm = TRUE</code>
Latched	No (unchecked)
Output tag	<i>(blank — leave it empty)</i>

Name, condition, latch flag, and output tag are the only configurable fields — there is no separate severity or message.

The alert is deliberately **non-latched**. Each time `float_alarm` goes TRUE it records a `FIRE` entry in the alert history, and each time it goes FALSE it records a `CLEARED` — so every occurrence shows up, not just the first. (A latched alert fires once and stays active until an explicit reset, which would give you a single stuck entry instead of an event log.)

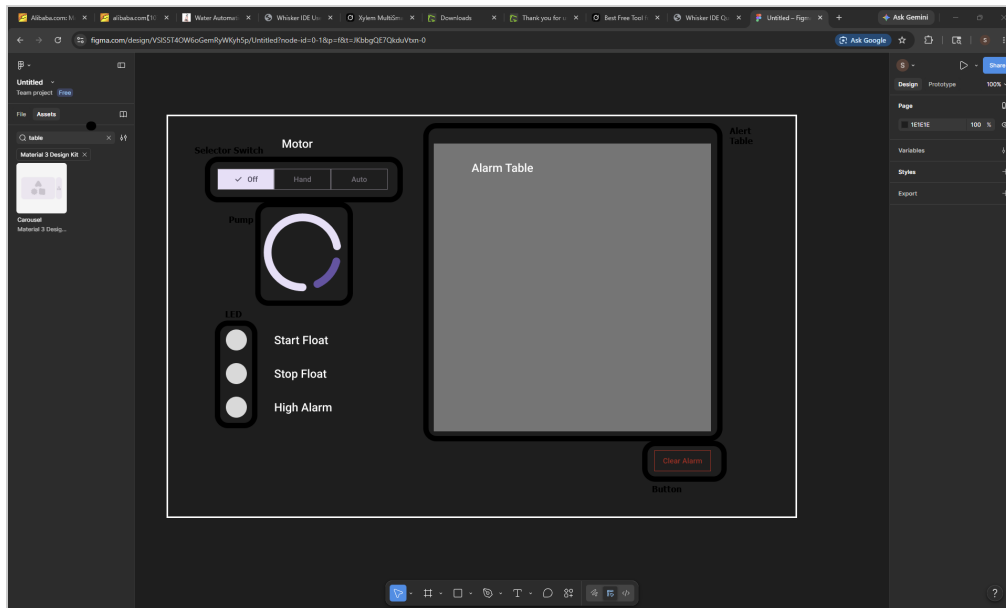
Latching lives in the ladder, not the alert. The motor *interlock* is held by the Set/Reset coils that drive `alarm_latched` (the control logic above), and the **Clear Alarms** button clears that latch. The alert sits alongside it purely to record occurrences for the history and the HMI alert table.

24.7 The HMI screen

One screen, identical across the three walkthroughs. Walkthrough 3’s WhiskerHMI screen is literally a copy of the Walkthrough 1 embedded HMI screen with the controller IP filled in.

Five widget types do all the work:

- **selectorSwitch** — the AHO mode selector. Configured with three positions (OFF=0, HAND=1, AUTO=2) and bound to `aho_mode`. Writable — the operator's choice goes back to the controller every time they tap.
- **led** — three of them for the float inputs. Read-only indicators. Green when TRUE.
- **pump** - for `motor_run`. Turns green and animates when motor is running.
- **pushButton** — the Clear Alarms button. Configured as momentary, bound to `clear_alarms`. Writes TRUE while held; the controller's one-shot logic immediately writes it back to FALSE.
- **alertTable** — a pre-built widget that shows current and historical alerts. No tag binding; it reads from the alert system's internal state. You don't have to wire anything up — drop it on the screen and it Just Works.



24.8 Watching the live data — Tag Monitor

Before you start a walkthrough, it helps to know where you'll watch the controller think. The Whisker IDE has three live-data views, and you'll lean on all of them during testing:

- **Tag Monitor** — bottom-panel tab, table view of every addressed tag and its current value, refreshed every poll cycle. This is the fastest way to confirm an input is wired correctly: wiggle a float switch, watch the `float_*` row flip `0 → 1` in well under a second. Filter box narrows the list

(`float` , `motor` , etc.); the eye icon toggles system tags (Comm OK, Cell health) in or out of view; the copy icon dumps the whole visible table to the clipboard as TSV. You'll use the Tag Monitor in Step 9 of each walkthrough.

- **Ladder editor's Debug Mode** (*Walkthrough 1 only*) — colors every contact and coil green/grey based on its live state. Best for “is the rung firing?” questions.
- **Python Log** — the controller's `ctx.log.info(...)` output streamed live, every entry selectable and copyable for pasting into chat or a bug report.

If you only remember one: the Tag Monitor is the one you'll open first when something doesn't behave the way you expect.

Note. Live values appear only after the IDE knows the controller is running the project you have open — in practice, after you press Build in Step 8 of a walkthrough. Until then the Tag Monitor stays empty on purpose, so you never read a stale program's memory under your project's tag names.

24.9 What's next

You're ready to build. Pick a walkthrough:

- **Walkthrough 1 — Ladder** if you want to start with the most visual / most PLC-traditional approach. Recommended if you're new to the Whisker IDE.
- **Walkthrough 2 — Python** if you prefer code, or want to see how the same logic shapes up when you can use variables, conditionals, and the full Python standard library.
- **Walkthrough 3 — WhiskerHMI** if you've completed Walkthrough 1 or 2 and now want to put an operator panel in front of it.

The walkthroughs are independent. You don't have to do all three, and you can do them in any order — though Walkthrough 3 assumes the controller from Walkthrough 1 or 2 is already running so it has something to connect to.

25 Your First PLC Project — Ladder

This walkthrough builds the scenario from [The Walkthrough scenario](#) as a ladder-logic PLC project running on a Nexus.io Automation Controller. By the end you'll have a working motor controller with a latched alarm, an Auto/Hand/Off selector, and a touchscreen HMI on the controller's display.

Allow about 45 minutes the first time through. Each subsequent walkthrough will go faster because you'll already know the IDE's mechanics.

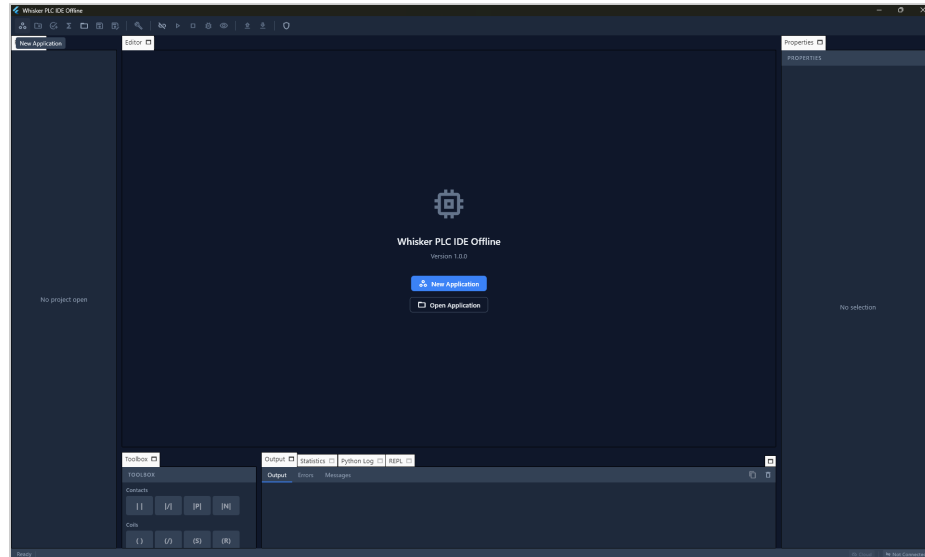
25.1 Before you start

You need:

- The Whisker IDE installed (see *Getting Started*).
- A Nexus.io controller powered on, on the same LAN as your PC (provisioned into your account first if you are on Path A — see *Choose your path* in the scenario chapter), with the mSmart Universal IO module wired per [the wiring map](#). Steps 1–7 need no controller; Step 8 does.
- The Walkthrough scenario chapter read (or at least skimmed) — this walkthrough doesn't re-explain the tags, wiring, or control intent.

25.2 Step 1 — Create the Application and Project

1. Launch the Whisker IDE.
2. From the toolbar, click the **New Application** icon (the leftmost icon, a small workspace symbol).



3. The New Application dialog appears. Fill it in:

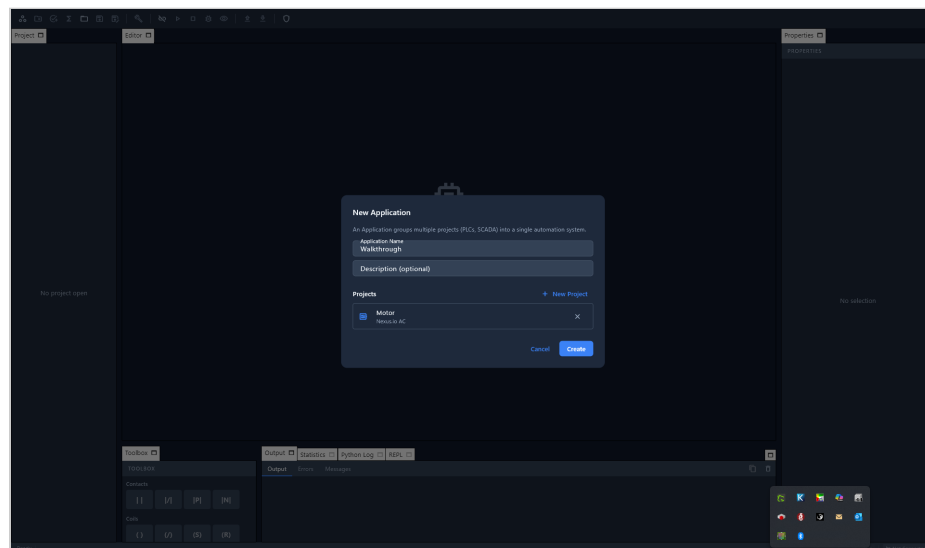
- **Application name:** Walkthrough-Ladder
- **Description:** leave blank (or type something brief like “Motor control demo”)

4. Click **New Project**. A New Project sub-dialog opens. Fill it in:

- **Project name:** Motor
- **Target hardware:** Nexus.io AC

Click **Add** to attach the Project to the new Application — you’ll return to the outer Application dialog with **Motor** listed.

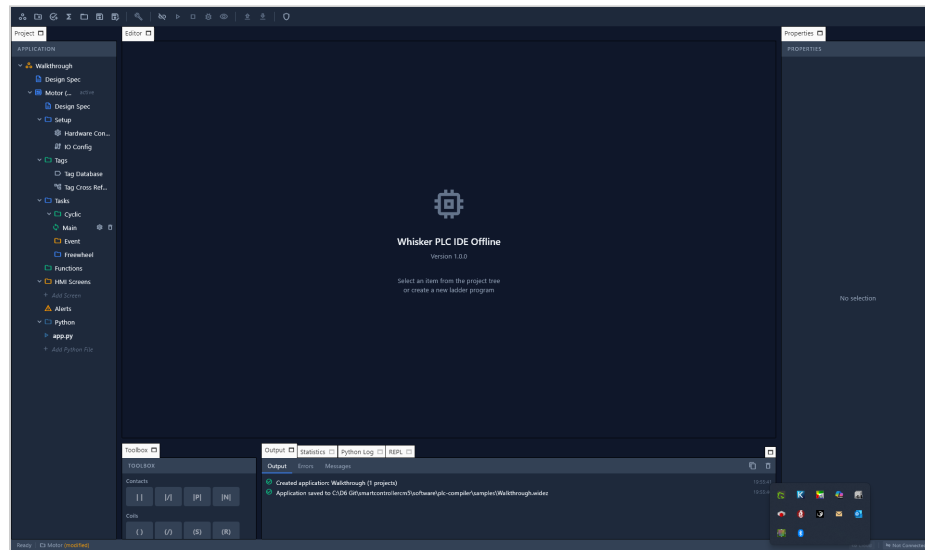
5. Back in the outer Application dialog, click **Create**.



- The IDE automatically opens a Save dialog for the new Application. Pick somewhere convenient (your Documents folder is fine) and accept the filename `walkthrough-Ladder.widez`. No need to press **Ctrl+S** — the Create button saves for you.

When the dialog closes you should see:

- A new Application named **Walkthrough-Ladder** in the title bar.
- A single project named **Motor** in the Project Tree on the left.
- An empty editor area in the middle.

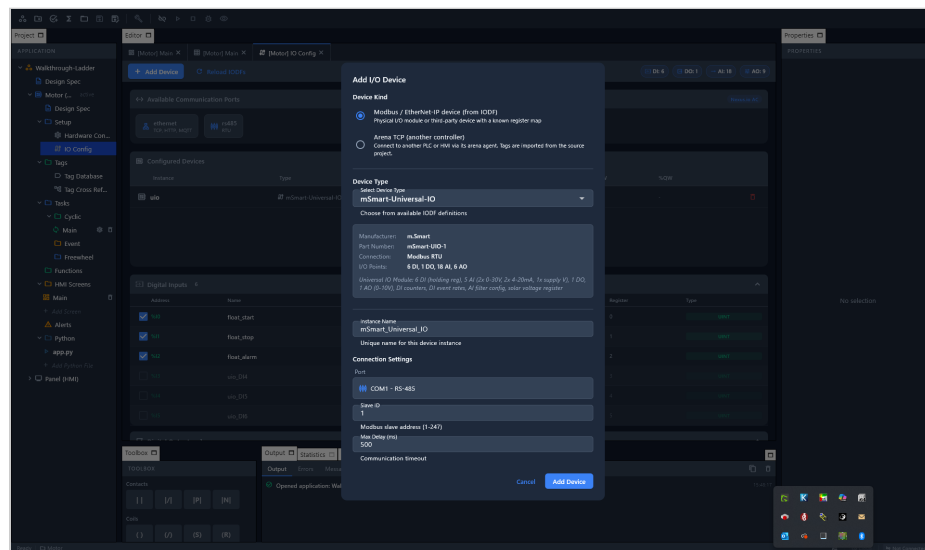


The Motor project comes pre-populated with a **Main** task and a Main ladder program — that's the file you'll write the logic in. We'll get there in a few steps.

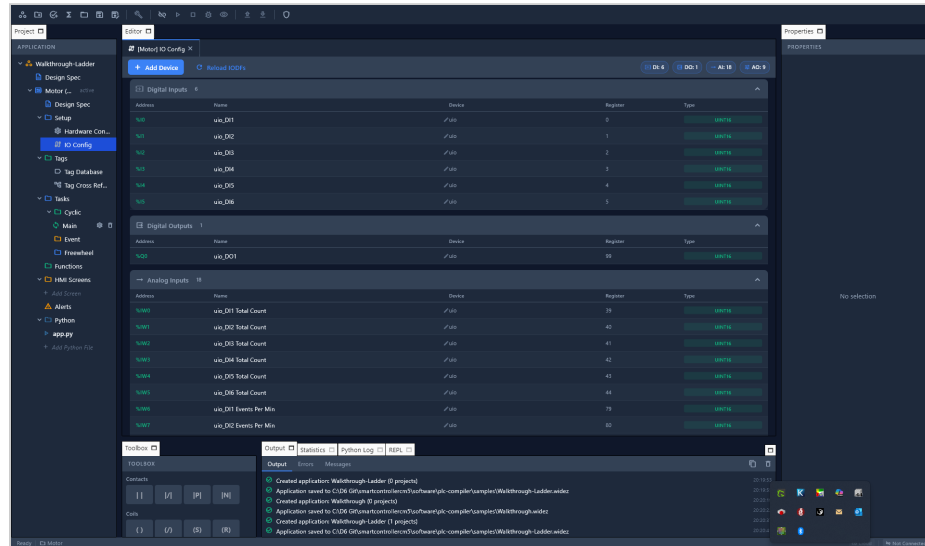
25.3 Step 2 — Add the mSmart Universal IO module

- In the Project Tree, click **Motor** → **Setup** → **IO Config**. The IO Configuration editor opens.
- Click **Add Device** in the toolbar. The Add Device dialog opens.
- Configure the device:
 - Device Kind:** select **Modbus / EtherNet-IP device (from IODF)**. This is the first choice in the dialog and it determines which fields appear below.

- **Device Type:** pick `mSmart-Universal-IO` from the dropdown. (If it's not there, the IDE hasn't seeded its default IODFs yet — re-open this dialog and it should appear.)
- **Instance name:** `uio` (short for “universal IO” — you'll reference this name later when mapping points).
- **Connection settings** auto-fill from the device type:
 - **Port:** `rs485` (the RS-485 port on the Nexus.io controller)
 - **Slave ID:** `1`



4. Click **Add Device**. The new device appears in the **Configured Devices** panel with a cable-style RTU icon to its left.
5. Look at the **Digital Inputs**, **Digital Outputs**, **Analog Inputs**, and **Analog Outputs** panels below the Configured Devices panel. The IDE has auto-created roughly 30 tags — one for every channel on the module — and distributed them into the right panel by type. Each tag is named `uio_<channel>` (so `uio_DI1`, `uio_DI2`, ..., `uio_D01`, ...) and has an auto-assigned IEC address.



6. **Disable the IO points you won't use.** Each row in the Digital Inputs / Digital Outputs / Analog Inputs / Analog Outputs panels has a checkbox at its left edge. When enabled (the default) the IO scanner polls that channel every scan cycle. The walkthrough uses only **four** channels — **DI1** , **DI2** , **DI3** , and **DO1** . Enable those and disable everything else:

- **Digital Inputs:** keep **DI1** , **DI2** , **DI3** ; uncheck **DI4** , **DI5** , **DI6** .
- **Digital Outputs:** keep **DO1** (it's the only one).
- **Analog Inputs:** uncheck *all* of them — **AI1 – AI5** , every ... **Filter Window** , every ... **Total Count** , every ... **Events Per Min** , and **Solar Voltage** . (The per-channel counters and filter windows are word values, so the IDE files them under Analog Inputs, not Digital Inputs.)
- **Analog Outputs:** uncheck **A01** (**uio_A01**).

Disabled rows dim out and stay in your project — flip the checkbox back on if you want them later.

Why this matters. The IO scanner does one Modbus transaction per enabled point per scan cycle. With all ~30 points enabled you're spending most of the bus's time reading values you never look at, and any one of those transactions can time out and slow the next scan. Trimming to the four you use takes wire traffic from ~30 transactions/scan down to ~2 (one coalesced read of DI1-3 plus one write to DO1) and makes the end-to-end response feel instant.

7. Press **Ctrl+S**.

Why this is the right place to create I/O tags. Tags that map to physical I/O don't make sense without a module behind them — the address `%I0` only exists because there's an input channel on a device somewhere. The IDE enforces this: you can't type a `%I` or `%Q` address into the Tag Database directly. You add the device, the IDE creates the tags, you rename them.

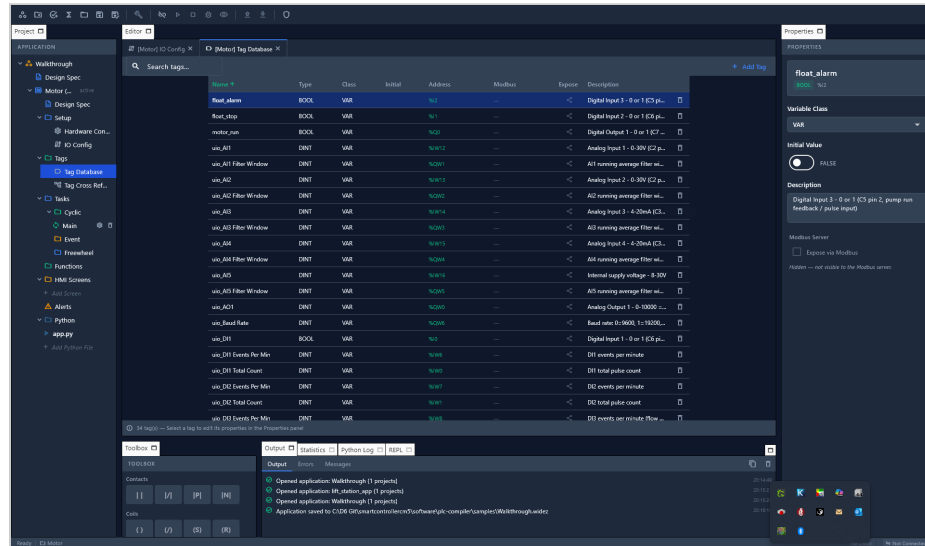
25.4 Step 3 — Rename the I/O tags you'll use

The 30 auto-created tags include analog readings, pulse counters, event rates, and config registers. The walkthrough only uses four of them. Renaming those four to friendlier names makes the ladder and HMI easier to read.

1. In the Project Tree, expand **Motor** → **Tags** and click **Tag Database**. The Tag Database editor opens. You'll see the `uio_*` tags the IO module created.
2. Double-click the `uio_DI1` row's **name** cell to edit it in place. Type `float_start` and press **Enter**. There is no "Rename" button or menu — name editing is just an in-place edit on the name cell.
3. Repeat for the other three:

From	To	Why
<code>uio_DI1</code>	<code>float_start</code>	Start float input
<code>uio_DI2</code>	<code>float_stop</code>	Stop float input
<code>uio_DI3</code>	<code>float_alarm</code>	High-alarm float input
<code>uio_D01</code>	<code>motor_run</code>	Motor contactor output

The addresses (`%I0` , `%I1` , `%I2` , `%Q0`) don't change — only the names do. Anywhere else in the project that referenced the old names is updated automatically.



4. Press **Ctrl+S**.

25.5 Step 4 — Add the memory tags

Eight more tags don't correspond to any physical wire — they're internal state the logic uses to remember things between scans, plus the mode flags the CMP block in Step 5 will produce. All of them go in the Tag Database directly.

1. Still in the Tag Database editor, click **Add Tag**.
2. First tag — the operator mode:
 - **Name:** `aho_mode`
 - **Type:** `INT`
 - **Class:** `VAR`
 - **Memory Area:** `Memory Word (16-bit)` → address auto-fills `%MW0`
 - **Description:** `Operator mode (0=Off, 1=Hand, 2=Auto)`

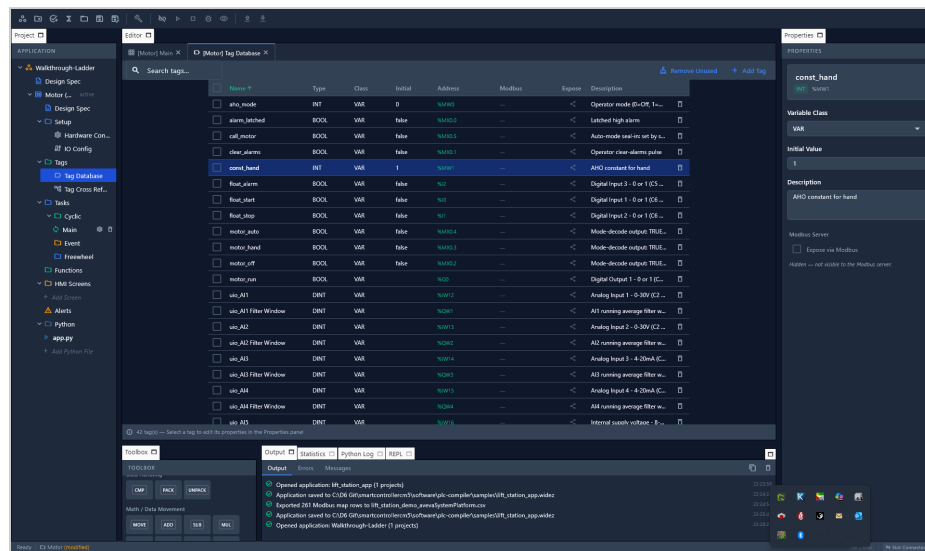
Click **Add**.

3. Repeat for the remaining seven:

Name	Type	Class	Memory Area	Address (auto)	Initial value	Description
const_hand	INT	CONST	Memory Word (16-bit)	%MW1	1	Constant used as IN2 of the CMP block — the AHO “Hand” value
motor_off	BOOL	VAR	Memory Bit	%MX0		Mode-decode output: TRUE when aho_mode = 0
motor_hand	BOOL	VAR	Memory Bit	%MX1		Mode-decode output: TRUE when aho_mode = 1
motor_auto	BOOL	VAR	Memory Bit	%MX2		Mode-decode output: TRUE when aho_mode = 2
call_motor	BOOL	VAR	Memory Bit	%MX3		Auto-mode seal-in: set by start float, reset by stop float
alarm_latched	BOOL	VAR	Memory Bit	%MX4		Latched high-alarm
clear_alarms	BOOL	VAR	Memory Bit	%MX5		Operator clear-alarms pulse

Set the Class column carefully. Every tag here is **VAR** (the default, like `aho_mode`) *except* `const_hand`, which must be **CONST**. The **Class** field is in the Tag properties panel.

Why a tag for a constant? The CMP block's IN1 and IN2 inputs both expect tag references — the IDE doesn't let you type a literal number into IN2. So we make one tag, `const_hand`, with Class **CONST** and an initial value of **1**, and use it as the comparison operand. As a **CONST** it never changes at runtime; it's just how you spell “the integer 1” to the CMP block. (Leaving it **VAR** lets the logic overwrite it, which breaks the comparison.)



4. Press **Ctrl+S**.

About address prefixes. **%I** and **%Q** were assigned to your I/O tags by the module. **%MX** is the prefix for internal memory *bits*, **%MW** for 16-bit *words*, **%MD** for 32-bit double-words, and **%MF** for 32-bit floats. The Memory Area dropdown lets you pick the right one; the IDE fills the address number automatically based on which slots are already taken.

25.6 Step 5 — Write the ladder logic

In the Project Tree, expand **Motor** → **Tasks** and click **Main**. (Main is a task, not a folder, so it doesn't expand — a single click on the task opens its ladder program directly; a double-click works too.) The Ladder Editor opens with five empty **networks** — what other PLC platforms call rungs. The sub-steps below add more networks as you need them using the **+** button at the top of the editor.

One coil per network. The Whisker IDE enforces a single terminal (coil, timer, counter, FB, etc.) per network. If two outputs need to be driven from the same condition — for example, “when `clear_alarms` is pressed, reset `alarm_latched` AND reset `clear_alarms`” — you spell that as two networks that share the same contact pattern, not as two parallel branches inside one network. That's why the walkthrough uses seven networks rather than five.

Ladder editor shortcuts you'll use in this step. The IDE leans on keyboard shortcuts more than drag-and-drop for branch wiring. Worth learning early:

Shortcut	What it does
drag from Toolbox	drop an instruction onto the selected segment
Ctrl + Down	create a parallel branch <i>below</i> the current branch
Ctrl + Up	reconnect the current branch back <i>up</i> to its parent
Ctrl + Right	extend a wire to the right on the current branch
Del	delete the selected instruction
Ctrl + Del	delete the entire current network
+ (toolbar)	add a new empty network at the bottom

Two conventions for the networks below. (1) You don't need to click a network to “select” it first — just drag the instruction onto the target network's segment. (2) In the element dialogs a **contact** asks for a **Tag Name** and a **coil** asks for an **Output Tag**; the steps use those labels.

25.6.1 Network 1 — Decode the operator mode (CMP)

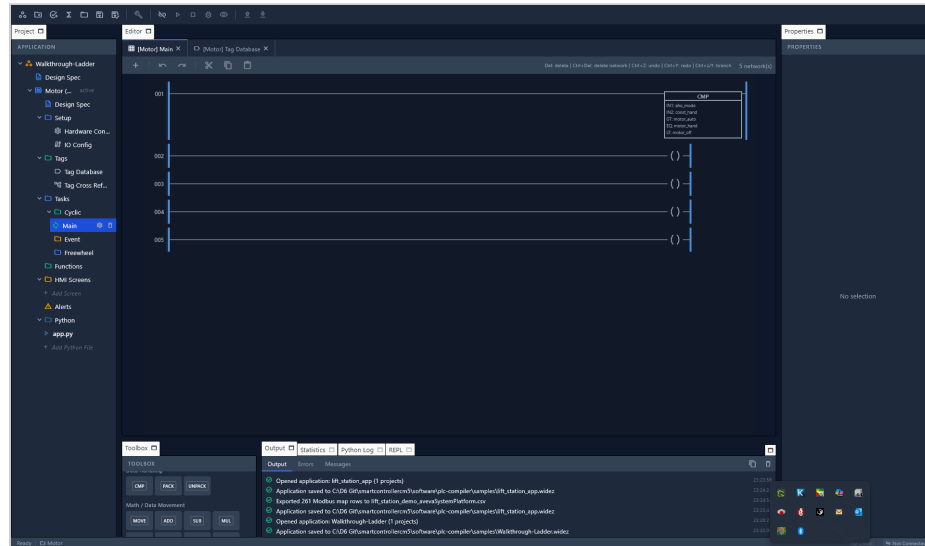
The IDE has single comparison contacts (EQ, NE, GT and so on — see the *Ladder Instruction Reference*), but it also ships a **CMP** function block that performs one comparison and writes three BOOL output tags directly — one each for “less than”, “equal”, and “greater than”. That’s exactly what we want for AHO — one network gives us `motor_off`, `motor_hand`, `motor_auto` all at once.

CMP is a **sink instruction** (like a coil). It has no right-hand output pin; instead its three outputs are set as properties in the property dialog and written to tag values directly. You drop one CMP in the terminal slot of a network and the network is complete.

1. From the **Toolbox**, drag a **CMP** block (under Compare) onto the terminal slot at the right end of network 1. A dialog appears. Give the block an **Instance name** (e.g. `cmp_mode`), then set the properties **in the order the dialog lists them** — IN1, IN2, then GT, EQ, LT:

- **IN1:** `aho_mode`
- **IN2:**
`const_hand` (the constant tag you added in Step 4 — the CMP block expects a tag reference here, not a literal number)
- **GT output:** `motor_auto` (Rationale: `aho_mode > 1` means `aho_mode = 2`, which is Auto.)
- **EQ output:** `motor_hand` (Rationale: `aho_mode = 1` is Hand.)
- **LT output:** `motor_off` (Rationale: `aho_mode < 1` means `aho_mode = 0`, which is Off.)

Click **OK**. The CMP block sits in the terminal slot with the four bound tag names labelled on it. There is no coil to wire — the CMP block writes the three output tags itself every scan.

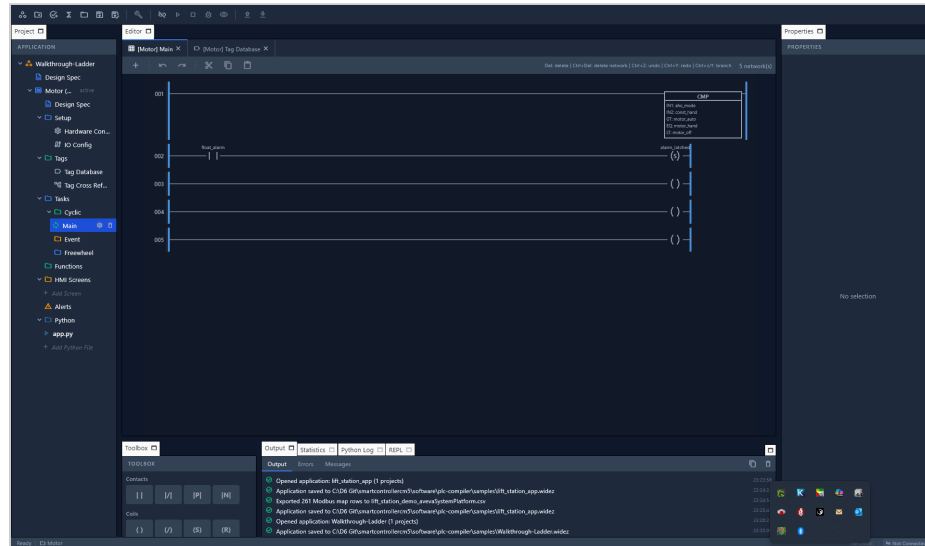


What's nice about this pattern. Three BOOL flags fall out of one CMP network. The rest of the program reads them like any other contact; nobody downstream needs to know `aho_mode` is an integer or what its values mean. That separation makes it easy to add a fourth mode later (e.g. a Maintenance mode at `aho_mode = 3`) — add a second CMP to compare `aho_mode` to the value 2 and you have four mode flags from two networks.

25.6.2 Network 2 — Latch the alarm

When the high-alarm float closes, latch `alarm_latched` with a **Set** coil so the bit stays on after the float opens.

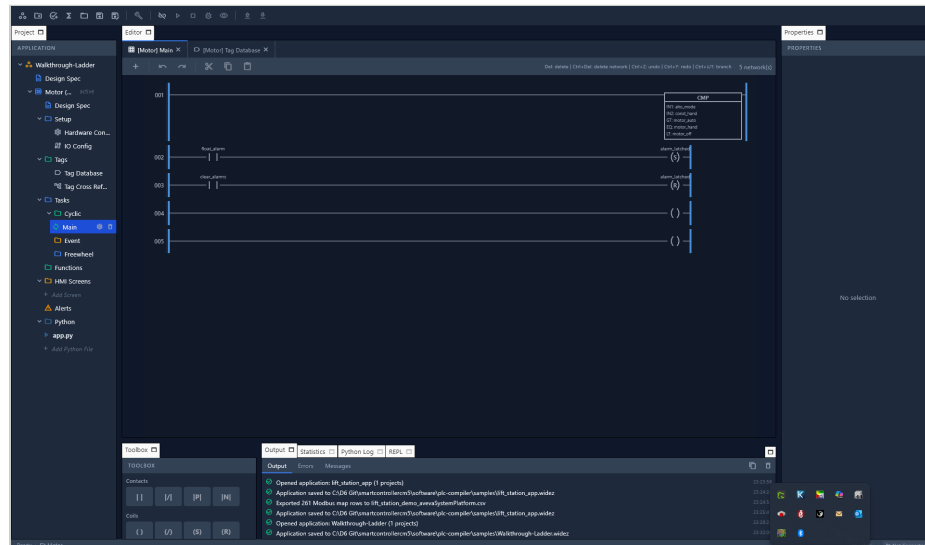
1. Click into network 2.
2. Drag a **Normally Open Contact** onto the first segment. Tag: `float_alarm`.
3. Drag a **Set Coil** (`(S)`) — the coil marked with an `s` in the toolbox — to the terminal slot. There is no “mode” to choose: the popup asks only for the output tag. Enter `alarm_latched`.



25.6.3 Network 3 — Clear the alarm latch

When the operator presses Clear Alarms, reset `alarm_latched`.

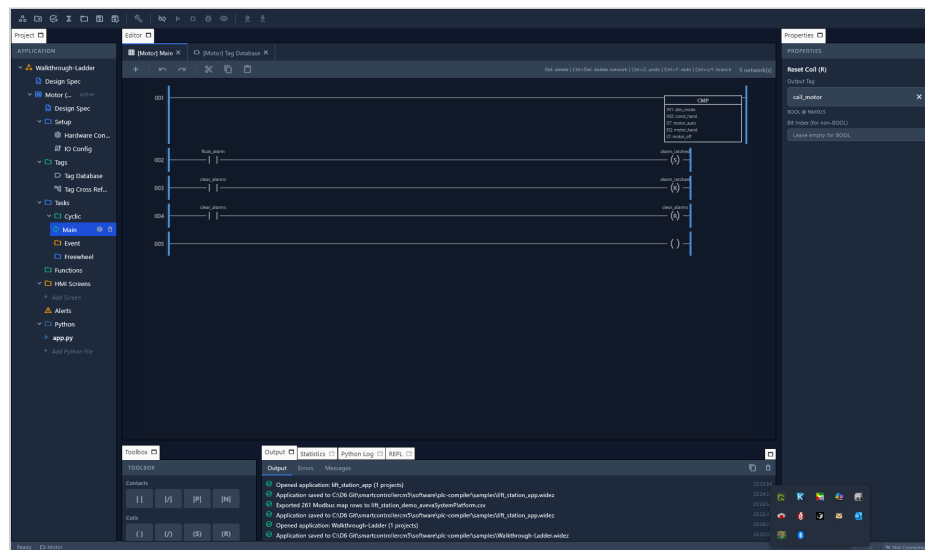
1. Click into network 3.
2. Drag a **Normally Open Contact** onto the first segment. Tag: `clear_alarms`.
3. Drag a **Reset Coil (R)** to the terminal slot. Tag: `alarm_latched`.



25.6.4 Network 4 — Auto-release the Clear Alarms button (one-shot)

`clear_alarms` is driven by the HMI button. The HMI writes TRUE while the button is pressed; we want the *controller* to immediately write it back to FALSE so the next scan sees a fresh edge if the operator presses again. This network resets `clear_alarms` using its own value as the trigger — that’s what makes the button behave as a one-shot pulse even though the HMI widget writes a level.

1. Click into network 4.
2. Drag a **Normally Open Contact** onto the first segment. Tag: `clear_alarms`.
3. Drag a **Reset Coil (R)** to the terminal slot. Tag: `clear_alarms`.



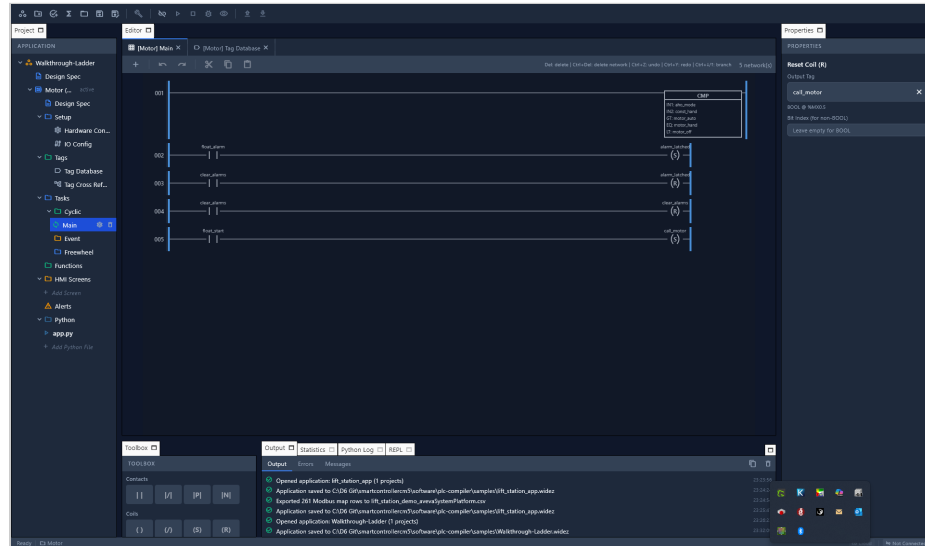
Why two networks instead of one? Networks 3 and 4 fire from the same condition (`clear_alarms = TRUE`) but drive different coils, and the IDE allows only one coil per network. So we spell the “reset both” intent as two networks with identical contact patterns. They execute in the same scan, so the behavior is equivalent to “reset both at once”.

25.6.5 Network 5 — `call_motor` SET (start float closes the tank’s request)

Pulling the Auto-mode seal-in out into its own pair of networks keeps the motor output readable. `call_motor` represents “the tank is asking for the motor to run” — it goes TRUE when the start float closes and stays TRUE (latched) until the stop float clears it in the next network.

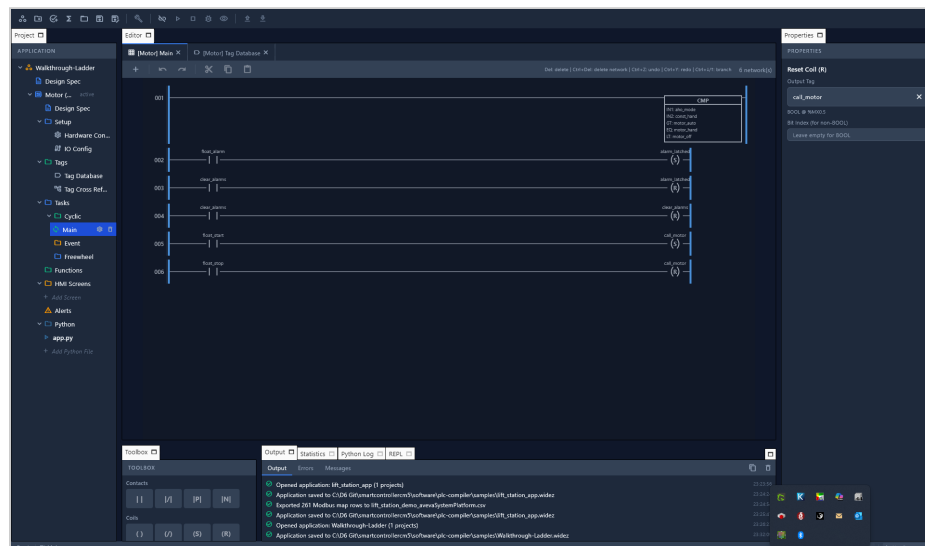
1. Click into network 5.

2. Drag a **Normally Open Contact** onto the first segment. Tag: `float_start`.
3. Drag a **Set Coil (S)** to the terminal slot. Tag: `call_motor`.



25.6.6 Network 6 — `call_motor` RESET (stop float clears the request)

1. Click the '+' button to add a new network - creates network 6.
2. Drag a **Normally Open Contact** onto the first segment. Tag: `float_stop`.
3. Drag a **Reset Coil (R)** to the terminal slot. Tag: `call_motor`.



25.6.7 Network 7 — Drive `motor_run`

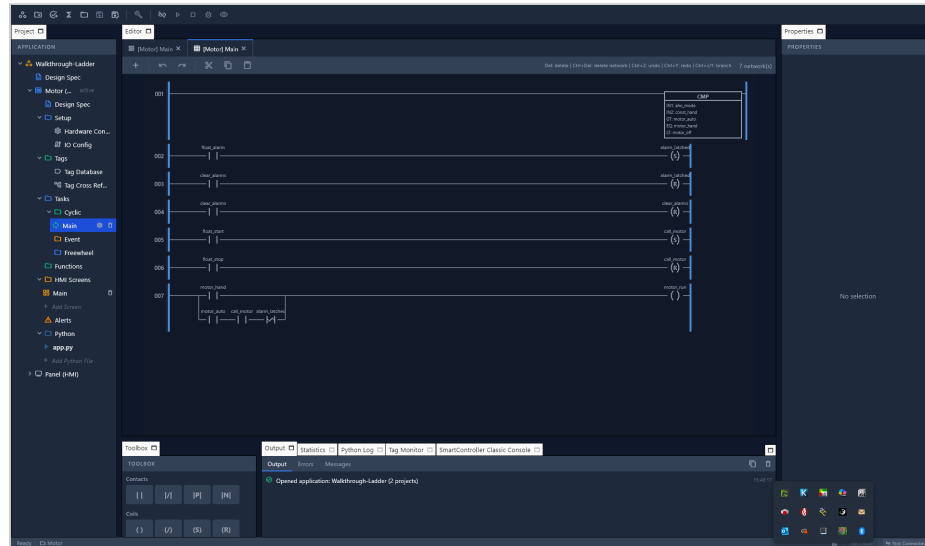
The payoff network. With all the state already decoded above, this one reads like a one-line sentence:

“Motor runs in Hand, or in Auto when the tank is calling and the alarm isn’t latched.”

Two parallel branches feed one Simple coil:

- **Top branch:** NO contact `motor_hand` . That’s it.
- **Bottom branch:** NO `motor_auto` → NO `call_motor` → NC `alarm_latched` , all in series.

1. Click the **+** button to add network 7.
 2. Drag a **Normally Open Contact** onto the first segment. **Tag Name:** `motor_hand` .
 3. Drag a **Simple Output** coil to the terminal slot (that’s the label in the toolbox tooltip). **Output Tag:** `motor_run` .
 4. With the `motor_hand` contact selected, press **Ctrl + Down** to add a parallel branch below.
 5. On the new branch, drag in series:
 - **Normally Open Contact**, tag `motor_auto` .
 - **Normally Open Contact**, tag `call_motor` .
 - **Normally Closed Contact**, tag `alarm_latched` .
- With the last contact (`alarm_latched`) selected, press **Ctrl + Up** to reconnect the sub-branch back to the parent rung so the two branches feed the same `motor_run` coil. (The branch does not auto-reconnect; you have to ask for it explicitly.)



6. Press **Ctrl+S**.

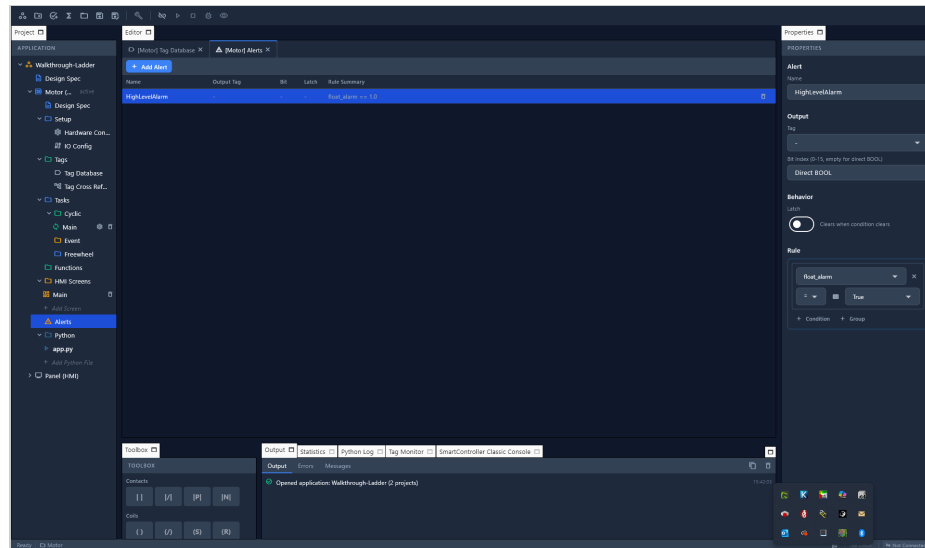
Tip: read the network out loud. “Motor runs when (*Hand*) or (*Auto AND call_motor AND no alarm*).” If that sentence matches the behavior you want, the network is correct. If you ever need to change which conditions inhibit the motor, this is the only network you’ll touch — the seal-in (networks 5–6) and mode decode (network 1) stay as-is.

25.7 Step 6 — Define the HighLevelAlarm alert

1. In the Project Tree, click **Motor** → **Alerts**. The Alerts editor opens.
2. Click **Add Alert**. The dialog asks only for a **Name** — enter `HighLevelAlarm` and confirm. The new alert appears in the Alerts list; **select it and configure the rest in the Properties panel** on the right:
 - **Condition:** build `float_alarm == True`. Pick `float_alarm` from the left dropdown; because it’s a **BOOL** tag the value field becomes a **True / False** picker — choose `==` and **True**. (The condition editor adapts to the tag’s type: **BOOL** gives a True/False picker, **INT/DINT** a whole-number field, and **REAL** a decimal field.)
 - **Latched:** **leave unchecked**. A non-latched alert *tracks* the condition — it logs a `FIRE`d each time `float_alarm` goes true and a `CLEAR`ed when it goes false, so every occurrence shows up in the alarm history. (A *latched* alert fires once and stays active until an explicit reset — not what you want for an event log.)

- **Output tag:** leave blank. The motor *interlock* is latched separately by the ladder Set/Reset coils in Networks 2–3; this alert just reports occurrences for the history and alert table.

The Alert system does not currently have severity or per-alert message fields; the only configurable pieces are name, condition, latch flag, and output tag.



3. Press **Ctrl+S**.

Latching lives in the ladder, not the alert. The *motor interlock* latches via the Set/Reset coils in Networks 2–3 (`alarm_latched`), and the **Clear Alarms** button clears it. The alert is deliberately **non-latched** so the alarm *history* logs every high/low of the float instead of one stuck entry. Two separate jobs: the ladder holds the interlock; the alert records events.

25.8 Step 7 — Build the HMI screen

1. In the Project Tree, click **Motor** → **HMI Screens**. Click **Add Screen** in the toolbar and name it `Main`. The HMI Designer opens with an empty canvas.
2. Place the widgets on the canvas. Widgets aren't dragged in from the Toolbox — **single-click a widget in the Toolbox** and the IDE autoplaces it on the canvas; then drag it from the canvas into the position you want. Place these widgets:
 - **selectorSwitch** at the top — for AHO mode.

- **pump** below it — for motor run status.
- **text** above the pump — label for the motor.
- **led × 3** in a row below the pump — for the three float inputs.
- **pushButton** at the bottom right — for Clear Alarms.
- **alertTable** along the bottom — for the alarm display.

3. Configure each widget by clicking it and editing the Properties panel.

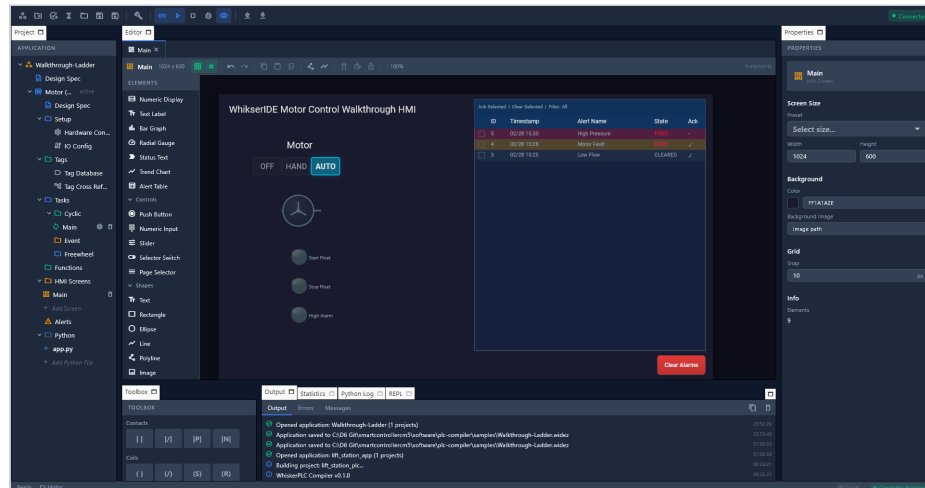
Tag bindings live at the bottom of the Properties panel. For each widget below, click it on the canvas and scroll the Properties panel to its **Bindings** section at the bottom. That's where every “Bind X → tag” instruction below is set; the upper parts of the Properties panel cover label, colors, and other look-and-feel options.

- **selectorSwitch** (the properties appear in this top-to-bottom order in the panel):
 - Positions: 3
 - Labels: OFF, HAND, AUTO
 - Bind `position` → `aho_mode` (in the **Bindings** section at the bottom of the panel)
- **pump** (motor run):
 - Bind **Running** → `motor_run`. The pump has no single “state” binding — it exposes separate **Running** and **Faulted** bindings.
 - **Faulted**: leave unbound — this walkthrough has no motor-fault tag.
 - **Active color**: green (shown while Running is true).
 - **Fault color**: red (shown when Faulted is true). It never shows here since Faulted is unbound, but in a real project a motor overload bit would bind to Faulted and operators rely on the red/green distinction for quick “is something wrong?” reads.
 - **Body color**: dark gray (the static housing of the pump icon).
- **text** (motor label):
 - Text: `Motor` (the text widget's field is **Text**, not Label)
- **led × 3** (floats): bind each one to `float_start`, `float_stop`, `float_alarm` respectively. Labels: `Start Float`, `Stop Float`, `High Alarm`.

◦ **pushButton:**

- Bind output → `clear_alarms`
- Mode: `momentary`
- Label: `Clear Alarms`

◦ **alertTable:** no binding needed; just drop it on the canvas. It reads alarm state from the alert system on its own.



4. Press **Ctrl+S**.

25.9 Step 8 — Connect and Build

1. Click the **Connect** icon in the toolbar (the plug icon to the right of Build). The Connect dialog opens and starts scanning the LAN. Each controller it finds is listed with its posture: a new controller shows as **Standalone**; one you have provisioned into your account shows as **Managed**.

Awaiting enrolment? That posture only appears on a controller ordered with management required; it accepts nothing until it is provisioned (Path A, or the site CA). See *Security, Provisioning and Enrolment*.

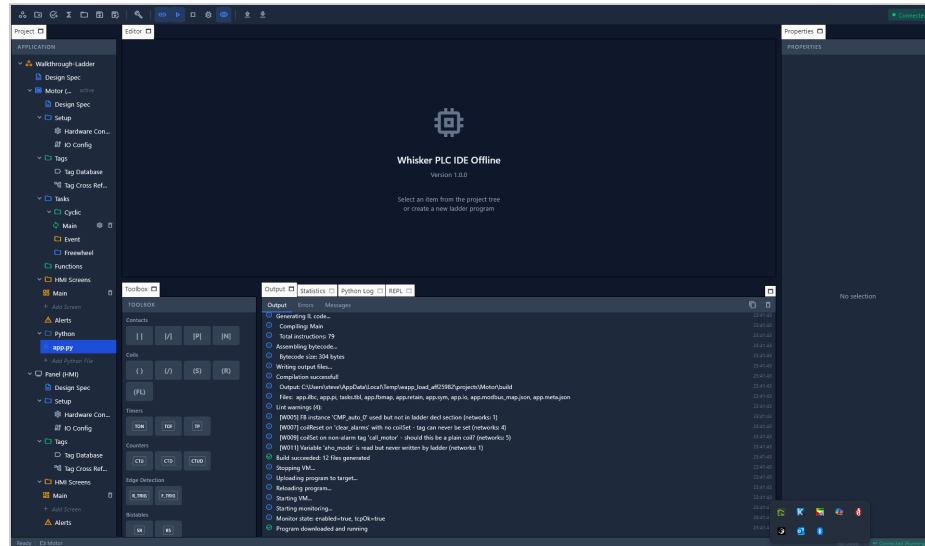
2. When your controller appears in the list, click it and click **Connect**. If it does not appear — mDNS discovery is blocked on many corporate networks and never crosses a VPN — click **Enter address manually** and type the controller's IP address.
 - **Path A:** the controller is managed, so the IDE connects securely using the certificate your account issued to you when you logged in. Nothing to configure.
 - **Path B:** the IDE connects plainly. (If you have made the controller managed through a site CA, it connects with your site identity instead; with more than one site imported, pick the right one under **Project Properties** → **Security** → **Site identity** first.)
3. A confirmation notification appears at the bottom of the IDE and the status bar shows the connection. The first time you connect with this project the IDE also compares it with whatever the controller already holds. A brand-new controller holds nothing, so nothing happens; if you later connect to a controller that runs a *different* project, a dialog offers to **Download from target** — choose it only if you want to edit that project instead of yours.
4. Press **F5** (or click **Build**). Build is only enabled while you are connected, because for a Nexus.io project one press does both jobs: the IDE compiles, then downloads the program to the connected controller (signing it first when the controller is managed). The Output panel shows the sequence:

```
Building project: Motor...  
Build succeeded: 12 files generated  
Uploading program to target...  
Reloading program...  
Starting VM...
```

On a managed controller (Path A) you will also see `Signed by ...` and `Uploading signed bundle to target...`.

If the build stops with a list of **undefined tags**, a name you typed somewhere (a contact, a coil, a widget binding) does not exist in the Tag Database. The dialog offers to create them; check the types it proposes and try again.

5. The controller loads the new program and starts it; the touchscreen switches to your Main screen within a few seconds.



Note. From this point on the IDE knows the controller is running exactly this project, and the Tag Monitor and Debug Mode come alive. If you edit the project afterwards, they switch off again until your next Build — that is deliberate, so you never watch live values under tag names the running program does not have.

25.10 Step 9 — Test it

If your hardware is wired and your float switches are accessible, this is the fun part.

Open the Tag Monitor first. Click the **Tag Monitor** tab in the bottom panel. With the controller connected you'll see live values for every addressed tag. Type `float` in the filter box to keep `float_start`, `float_stop`, `float_alarm` (and `motor_run`) on screen. As you exercise each switch in the steps below, watch these values flip in real time — if a switch doesn't move its tag, the issue is wiring or slave config, not your ladder logic.

1. **Verify the HMI** — Walk over to the controller's touchscreen. The Main screen should be showing the layout you designed. The mode selector starts at OFF, all LEDs are dark.
2. **Hand mode** — Tap **HAND** on the selector. The Motor LED turns green and DO1 energizes on the I/O module. Confirm by looking at the module's DO1 LED.
3. **Off mode** — Tap **OFF**. Motor LED goes dark, DO1 de-energizes.

4. **Auto mode without floats** — Tap **AUTO**. With both floats open, motor stays off (no start signal yet).
5. **Auto seal-in** — Briefly close the start float. The Motor LED should turn on and stay on after the float opens again.
6. **Auto stop** — Close the stop float. Motor LED goes off.
7. **High alarm** — Close the high-alarm float. The Alarm LED lights, the alert table shows `HighLevelAlarm`, and the motor turns off (if it was on in Auto). Now open the float again: the LED and alert table clear (they track the live float), but **the motor stays off** — the ladder's Set-coil latch (Network 2) holds the interlock until you clear it. Each high/low of the float adds a `FIRED / CLEARED` pair to the alarm history.
8. **Hand override** — While alarmed, tap **HAND**. Motor turns back on (the safety override). Tap back to **AUTO** — motor turns off again.
9. **Clear the interlock** — Tap **Clear Alarms**. This resets the ladder's `alarm_latched` (Network 3), releasing the motor interlock so the motor can run again in Auto. (The alert table already cleared when you opened the float — it tracks the live condition.)

If all nine steps behave as described, you're done with the Ladder walkthrough. The controller is now running a complete motor-control program with an alarm system and an operator panel.

25.11 What's next

- Continue to [Walkthrough 2 — Python](#) to see the same project written in Python. The contrast between the two approaches is the most educational part of doing both.
- Continue to [Walkthrough 3 — WhiskerHMI](#) to add a Windows HMI panel that mirrors the controller's embedded screen.
- Or stop here and explore. Open the Tag Monitor (see *Monitoring and Debugging* in the full manual) to watch tag values change in real time. Experiment with adding a runtime timer to the alarm latch so it auto-clears after 5 minutes. The ladder logic is yours now.

26 Your First PLC Project — Python

This walkthrough builds the same scenario as the Ladder walkthrough, on the same Nexus.io controller, with the same I/O module and the same HMI screen — but the control logic lives in `app.py` instead of in ladder networks (rungs). If you've already finished the Ladder walkthrough, most of the setup steps will be familiar.

Reading both walkthroughs side by side is the fastest way to understand when ladder is the right tool and when Python is. Short version: ladder is unmatched for boolean interlock logic the way a maintenance electrician will read it; Python is unmatched when you need arithmetic, data structures, conditionals more than two levels deep, or a library call. The motor scenario fits both, which is exactly why it's a useful comparison.

Allow about 30 minutes if you've already done the Ladder walkthrough, or 45 minutes if this is your first.

26.1 Before you start

Same prerequisites as the Ladder walkthrough — a Nexus.io controller on your LAN (provisioned into your account first on Path A), the mSmart Universal IO wired up, and the [Walkthrough scenario](#) chapter read.

26.2 Step 1 — Create the Application and Project

Same procedure as Walkthrough 1, Step 1. The only difference is the naming: call the Application `Walkthrough-Py` and the Project `Motor` (target `Nexus.io AC`). Save as `Walkthrough-Py.widez`. We use a different filename so it can coexist with the Ladder walkthrough project on your disk.

(If you skipped Walkthrough 1, see its [Step 1](#) for screenshots of the dialogs.)

26.3 Step 2 — Add the mSmart Universal IO module

Same as Walkthrough 1 Step 2 — open **Motor** → **Setup** → **IO Config**, click **Add Device**, set **Device Kind** to **Modbus / EtherNet-IP device (from IODF)**, pick `mSmart-Universal-IO` as the **Device Type**, instance name `uio`, port `rs485`, slave ID `1`, defaults for the rest. The IDE auto-creates ~30 `uio_*` tags, one per channel on the module.

Don't forget the **Disable the IO points you won't use** sub-step from the Ladder Walkthrough Step 2 — uncheck everything except `DI1`, `DI2`, `DI3`, and `D01`. Same reasoning as the ladder walkthrough: the IO scanner does one Modbus transaction per enabled point per scan, so trimming to the four channels you actually read makes the controller's response to a float change feel instant.

26.4 Step 3 — Rename the I/O tags

Same as Walkthrough 1 Step 3 — in the Tag Database, rename the four tags the walkthrough uses:

From	To
<code>uio_DI1</code>	<code>float_start</code>
<code>uio_DI2</code>	<code>float_stop</code>
<code>uio_DI3</code>	<code>float_alarm</code>
<code>uio_D01</code>	<code>motor_run</code>

The tag *names* are what the Python code references — that's why they have to match. The addresses (`%I0` / `%I1` / `%I2` / `%Q0`) are unaffected by the rename.

26.5 Step 4 — Add the memory tags

Add the seven memory tags below — the same set as Walkthrough 1 **minus** `const_hand`. The ladder version needed `const_hand` only because the CMP block can't take a literal operand; in Python you compare `aho_mode == 1` directly, so there's no constant tag to create. The Python code in Step 5 still writes the four intermediate tags (`motor_off` / `motor_hand` / `motor_auto` / `call_motor`) just like the ladder version does, so you can see the same intent expressed two ways.

Name	Type	Memory Area	Address
<code>aho_mode</code>	INT	Memory Word (16-bit)	<code>%MW0</code>
<code>motor_off</code>	BOOL	Memory Bit	<code>%MX0</code>
<code>motor_hand</code>	BOOL	Memory Bit	<code>%MX1</code>
<code>motor_auto</code>	BOOL	Memory Bit	<code>%MX2</code>

Name	Type	Memory Area	Address
call_motor	BOOL	Memory Bit	%MX3
alarm_latched	BOOL	Memory Bit	%MX4
clear_alarms	BOOL	Memory Bit	%MX5

26.6 Step 5 — Write the Python application

This is the part that differs. The Motor project comes with a default `app.py` containing a stub `run(ctx)` function and a brief comment explaining the available API. You'll replace the stub with the motor control logic.

1. In the Project Tree, expand **Motor** → **Python** and click `app.py`. The Python Editor opens (a single click is enough; there's no separate "open" step).

Editor tips. Tab and Shift-Tab indent/outdent the currently-selected lines (no selection = insert 4 spaces / outdent the current line). Pasting code from another editor, chat, or the web normalizes tab characters to 4 spaces automatically — no need to reformat first.

2. Replace the entire contents of the file with the script below.

Don't retype it — and don't copy it out of this PDF. Copying Python from a PDF mangles the indentation (and PDFs may split the block across a page), which breaks the code because indentation is significant. The complete script already ships with the IDE: open the **Walkthrough-Py** sample (File → Open → Documents\Whisker Projects) and copy the `run(ctx)` body from its Python editor, where the indentation is intact. The listing here is for reading along.

```

"""Motor controller – Walkthrough 2

Implements the AHO state machine, alarm latching, and one-shot
Clear Alarms behavior described in the Walkthrough scenario.
Equivalent in behavior to the ladder version in Walkthrough 1
–
the five sections below mirror Networks 1-5 of the ladder
program.
"""

SCAN_PERIOD_S = 0.05      # 50 ms – matches the I/O scanner
                           cadence

def run(ctx):
    ctx.log.info("Motor controller started")

    while not ctx.shutdown_requested:
        # --- Read inputs
        -----
        float_start = ctx.read_tag("float_start")
        float_stop  = ctx.read_tag("float_stop")
        float_alarm  = ctx.read_tag("float_alarm")
        clear_alarms = ctx.read_tag("clear_alarms")
        aho_mode     = ctx.read_tag("aho_mode")
        alarm_latched = ctx.read_tag("alarm_latched")
        call_motor   = ctx.read_tag("call_motor")      #
        previous scan

        # (1) Mode decode – CMP equivalent
        -----
        motor_off = aho_mode < 1      # LT (aho_mode = 0 →
        Off)
        motor_hand = aho_mode == 1    # EQ (aho_mode = 1 →
        Hand)
        motor_auto = aho_mode > 1     # GT (aho_mode = 2 →
        Auto)

        # (2) Latch the alarm
        -----
        if float_alarm:
            alarm_latched = True

        # (3) One-shot clear
        -----
        if clear_alarms:
            alarm_latched = False
            ctx.write_tag("clear_alarms", False)      # auto-
            release

```

```
# (4) call_motor seal-in
-----

if float_start:
    call_motor = True
if float_stop:
    call_motor = False

# (5) Motor output
-----

motor_run = motor_hand or (
    motor_auto and call_motor and not alarm_latched)

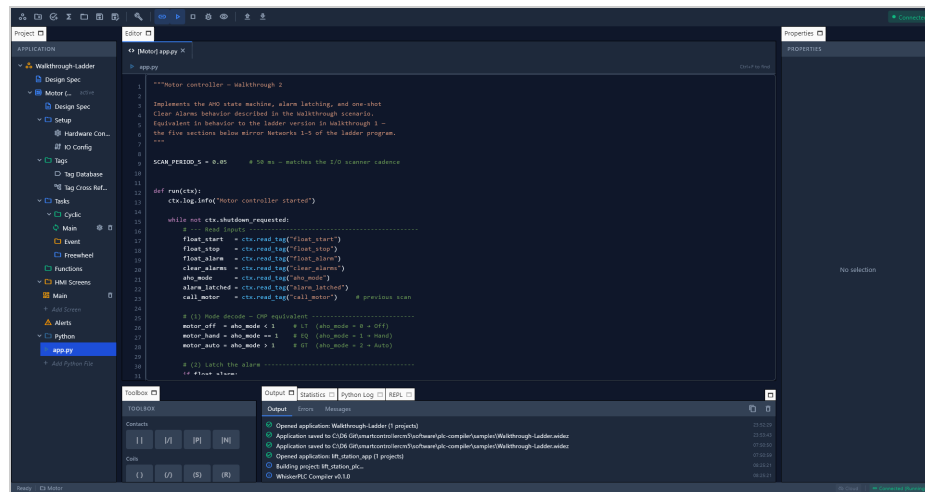
# --- Write outputs
-----

ctx.write_tag("motor_off",      motor_off)
ctx.write_tag("motor_hand",     motor_hand)
ctx.write_tag("motor_auto",     motor_auto)
ctx.write_tag("call_motor",     call_motor)
ctx.write_tag("alarm_latched",  alarm_latched)
ctx.write_tag("motor_run",     motor_run)

ctx.wait(SCAN_PERIOD_S)

ctx.log.info("Motor controller stopped")
```

3. Press **Ctrl+S**.



A few things worth understanding before you move on:

- **ctx is the controller's API surface.** The `ctx` object is passed into `run(ctx)` by the Python runtime on the controller. It gives you `read_tag` / `write_tag` for named tags, lower-level `di_read` / `dq_write` / `mw_read` /

`mw_write` for direct arena access, and `wait(seconds)` for sleeping in a shutdown-aware way. Don't use `time.sleep()` — it won't notice when the controller's service is being shut down for an update.

- **while not `ctx.shutdown_requested`** is the standard outer loop. When the controller's smartcontroller service receives a stop or an update, it sets `shutdown_requested = True` and `ctx.wait()` returns immediately so your loop can finish its scan and exit cleanly. The `ctx.log.info("...stopped")` line confirms you exited normally.
- **The scan period is yours to choose.** Ladder is fixed at the task's configured rate (typically 10 ms on the Nexus.io controller). Python runs in its own thread; you decide how often it loops. 50 ms is a reasonable default for HMI-driven logic — fast enough that the operator doesn't see latency, slow enough that the controller has headroom for everything else. Drop it to 10 ms if you need ladder-comparable response; raise it to 500 ms if you're doing heavy calculations between scans.
- **"Previous scan" semantics for `call_motor`.** The line `call_motor = ctx.read_tag("call_motor")` reads the latch's *previous* state into a local variable. If neither `float_start` nor `float_stop` fires this scan, the local variable still holds the previous value and gets written back unchanged at the bottom. That's the same pattern as a ladder network where a coil tag is referenced as a contact in the same network — the value being read is whatever was written by the previous scan. In Python you have to do it explicitly; in ladder the editor does it implicitly.
- **No need to update `clear_alarms` on the HMI.** When you call `ctx.write_tag("clear_alarms", False)`, the change goes into arena memory immediately. The HMI runtime polls arena and sees the change on its next refresh, so the button visually de-activates within a few hundred milliseconds.
- **Why write `motor_off` / `motor_hand` / `motor_auto` at all?** Strictly, the Python code could just use the local booleans in the `motor_run` expression and never write them to tags. But writing them gives the Tag Monitor (and the HMI, if you add indicator LEDs) visibility into the controller's mode decode, matching exactly what the ladder version exposes. It's a couple of microseconds per scan and it makes the two implementations directly comparable.

26.7 Step 6 — Define the HighLevelAlarm alert

Same as the Ladder Walkthrough Step 6. Name `HighLevelAlarm`, condition `float_alarm == True` — `float_alarm` is a BOOL, so the value field is a **True / False** picker (choose `==` and **True**) — **Latched unchecked** (the alert tracks the condition so every occurrence is logged; your Python code handles the motor-interlock latch), output tag blank. The Alert system has no severity or message field — name, condition, latch, and output tag are the only knobs.

26.8 Step 7 — Build the HMI screen

Same as Walkthrough 1 Step 7 — same five widgets, same bindings, same layout. If you've already built it in your Ladder project, you can copy the screen across rather than rebuilding it. The clipboard **isn't** shared between two IDE windows, so do it within a single window by switching projects:

1. Save this Python project (**Ctrl+S**), then open the Ladder walkthrough's `Walkthrough-Ladder.widez` (**File** → **Open**) in the same window.
2. In its Project Tree, right-click **Motor** → **HMI Screens** → **Main** and choose **Copy**.
3. Re-open your Python project (**File** → **Open**), right-click its **HMI Screens** node, and choose **Paste**.

Otherwise build it from scratch following Walkthrough 1's HMI step.

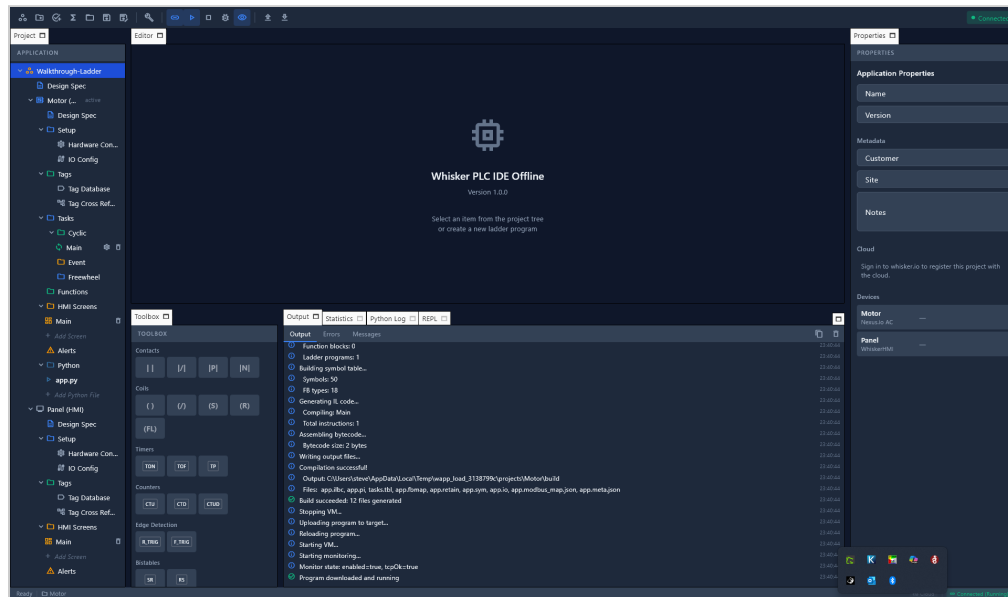
26.9 Step 8 — Connect and Build

Same as Walkthrough 1 Step 8: connect to the controller (Path A or Path B), then press **F5** — one press compiles and downloads (signing first on a managed controller).

This time the build output mentions the Python bundle:

```
Building project: Motor...
Build succeeded: 11 files generated
Uploading program to target...
Reloading program...
Starting VM...
```

The controller's smartcontroller service detects the new `python_app.json` bundle within a couple of seconds, stops the previous `run(ctx)` thread (if any), unpacks the new bundle, and starts the new one. You don't have to restart anything manually.



26.10 Step 9 — Test it

Same nine test steps as Walkthrough 1 Step 9 — Hand, Off, Auto seal-in, Auto stop, alarm latches, Hand overrides alarm, Clear Alarms releases. The behavior should be indistinguishable from the ladder version.

The Python walkthrough doesn't have a Debug Mode (Python isn't a graph that lights up), so the **Tag Monitor** is your primary window into what the controller is doing — open it in the bottom panel and filter on `float`, `motor`, or `alarm` to keep the relevant tags on screen while you exercise the floats. If a tag doesn't move when you expect it to, you can check the **Python Log** to see what your `run()` loop saw.

If you have both walkthrough projects on hand, try this: deploy the Ladder project, run through the tests, then deploy the Python project and run through them again. Because the two are different projects, the IDE asks you to confirm replacing the one already on the controller — that's expected; confirm it. You won't be able to tell which is running just by watching the HMI — which is the whole point.

26.11 What Python gives you that ladder doesn't

The motor scenario is intentionally simple, so this walkthrough doesn't show off what Python is genuinely good for. Some things you could try next, that would be painful or impossible in ladder:

- **Runtime configuration.** Read a YAML file shipped with the project, change setpoints without recompiling.
- **HTTP calls.** Fetch a setpoint from a remote service every minute. Push status to a webhook. Pull a weather forecast and adjust based on rain probability.
- **State machines beyond two levels.** AHO is shallow; some processes have nested states (sequencing through a startup procedure, for instance) that get unreadable in ladder fast but stay clean in Python.
- **Logging and debugging.** `ctx.log.info(...)` writes to the controller's Python log. The IDE's **Python Log** panel (bottom of the workspace, next to the Output and Tag Monitor tabs) streams that log live — each entry selectable and copyable for pasting into chat or a bug report. The Python Log panel is all you need to watch your code run.
- **Standard library.** `datetime`, `statistics`, `json`, `re`, `collections` — all available, all the same Python you know from your laptop.

26.12 What's next

- Continue to [Walkthrough 3 — WhiskerHMI](#) to put a Windows HMI panel in front of the controller you just programmed (whether by ladder or Python).
- Open the Tag Monitor (see *Monitoring and Debugging* in the full manual) and watch the values change as you exercise the floats and selector.
- Try extending the Python code: add a runtime counter that records motor starts and reports the total with `ctx.log.info(...)`. It streams live to the IDE's **Python Log** panel, so you can read and copy it without pulling a file off the controller. See *Python on the Controller* in the full manual for the full `ctx` API — and for Python function blocks, which let a ladder network call your Python.

27 Your First WhiskerHMI Project

This walkthrough adds a Windows HMI panel to the Application you built in Walkthrough 1. Instead of creating a new Application from scratch, you'll open `Walkthrough-Ladder.widez` and add a second Project to it — a WhiskerHMI project — that runs on a Windows PC and shows the same operator screen the controller's built-in touchscreen shows.

WhiskerHMI is a viewer + interactor. It has no I/O, no ladder, no Python, no alerts of its own. It reads tags directly from the controller's arena agent over TCP and writes operator input back. All control intent still lives in the Motor PLC project from Walkthrough 1. WhiskerHMI just shows it.

The result is a self-contained Windows installer (.exe) you can run on a desktop PC, a panel PC on the plant floor, or several PCs at once — every panel sees the same live state from the same controller.

Allow about 15 minutes. You need to have completed Walkthrough 1 first (or Walkthrough 2; the WhiskerHMI side is identical either way).

Path A or Path B? On **Path B** (a controller that is not managed, which is how a new controller arrives) the panel connects plainly and this walkthrough is complete as written. On **Path A** (a managed controller — cloud-enrolled, or provisioned through a site CA) the controller accepts only secure connections on its data port, so the panel PC needs a **panel identity**: in Step 3 tick **Secure connection**, and after installing the panel in Step 8 issue an identity for that PC from the Security dialog and import it from the panel's **Panel ▶ Import panel identity...** menu. The procedure is in *The WhiskerHMI PC Application*.

27.1 Before you start

- Walkthrough 1 (or 2) is complete and the Motor PLC project is running on a Nexus.io controller.
- You know the controller's IP address (the one you connected to in the Ladder Walkthrough Step 8).
- The PC where the WhiskerHMI panel will run is on the same LAN as the controller. (You can build and test on your dev PC; the installer copies to any other Windows PC.)

27.2 Step 1 — Reopen the Walkthrough-Ladder Application

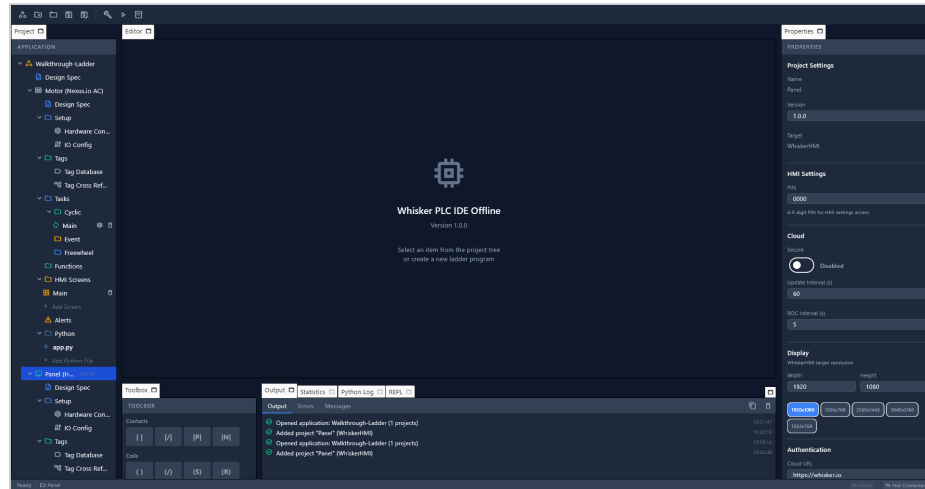
1. **File** → **Open Application**, pick `Walkthrough-Ladder.widez`. The IDE loads the Application; the Project Tree shows the **Motor** project from Walkthrough 1.

Why no Modbus setup this time. Earlier versions of WhiskerHMI required you to expose tags via the controller's Modbus server and manually map them on the panel side. With ArenaTCP the panel reads the controller's arena agent directly — every tag is automatically readable, no expose step, no Modbus address bookkeeping. The Modbus server is still there if you need it for third-party SCADA, but WhiskerHMI doesn't use it.

27.3 Step 2 — Add a WhiskerHMI project to the Application

You already have an Application — Walkthrough. You're going to add a second project to it.

1. In the Project Tree, click the Application root node (**Walkthrough-Ladder**) to select it.
2. From the toolbar, click **New Project** (the same button you used in the Ladder Walkthrough to add the Motor project, but now you're adding a sibling).
3. In the New Project dialog:
 - **Project name:** `Panel`
 - **Target hardware:** `WhiskerHMI`
4. Click **Create**. The Project Tree now shows two siblings under **Walkthrough-Ladder**: **Motor** (the PLC project from the Ladder Walkthrough) and **Panel** (the new WhiskerHMI project, currently empty).



5. Press **Ctrl+S** to save the Application.

What WhiskerHMI looks like in the tree. A WhiskerHMI project has no Tasks branch (it runs no ladder or Python) and no Functions — it has Tags, IO Configuration, Alerts, and HMI. The IDE knows the target type runs no control logic and hides the branches that don't apply.

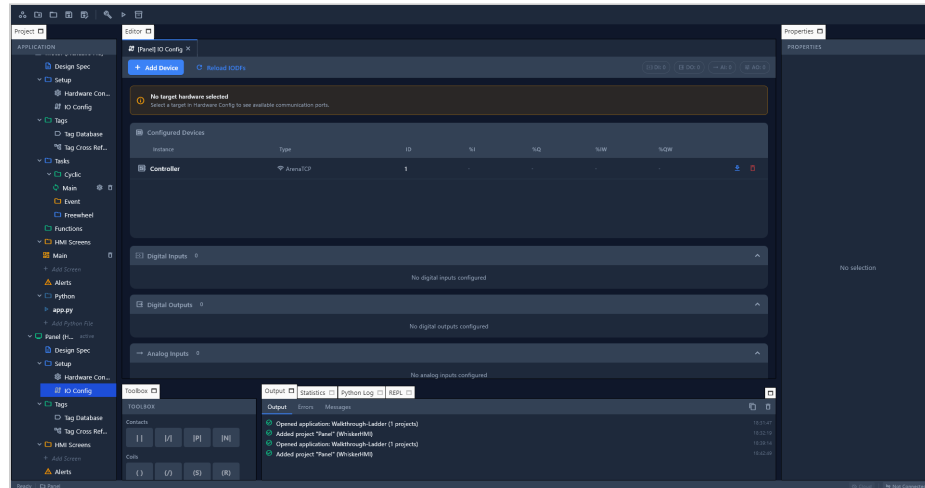
27.4 Step 3 — Add an ArenaTCP device pointing at the controller

The Panel needs to know which controller to read tags from. ArenaTCP is a direct connection to the controller's arena agent on TCP port 5000 — no Modbus translation in between.

1. **Panel** → **Setup** → **IO Config** → click **Add Device**.
2. At the top of the dialog, choose **ArenaTCP** for the device kind (the alternative is IODF-backed Modbus, which is what you used in the Ladder Walkthrough if you have any RTU devices wired in).
3. Fill in the ArenaTCP form:
 - **Instance name:** `controller`
 - **IP address:** the Nexus.io controller's address from Walkthrough
 1. (If you started from a shipped sample Panel project rather than building it here, its `controller` device may carry a placeholder address — set it to your controller's actual one.)

- **TCP port:** 5000 (the controller's data port, *not* the 9000 port the IDE uses to program it).
- **Secure connection:** unticked on Path B; ticked on Path A (see the box at the top of the chapter).

4. Click **Add Device**. The Configured Devices table shows the new **controller** row with a download icon (Import Tags) next to the delete icon.



Why ArenaTCP and not Modbus. Modbus is the right answer when the remote endpoint is a third-party device that doesn't speak anything else. Talking from one Whisker controller (or HMI) to another, the arena agent is the native protocol — it carries tag names, data types, and qualities, so the panel doesn't need a hand-maintained Modbus address table to know what each register means.

27.5 Step 4 — Import tags from the Motor project

Now bring the six HMI-facing tags from the Motor project into the Panel. Doing this through the import flow keeps the names, types, and controller addresses in sync — no retyping, no transcription errors.

1. On the **controller** row in the Configured Devices table, click the **download** icon (Import Tags).
2. The Import Tags dialog opens. The source list at the top shows sibling projects in the current Application. Pick **Motor**.

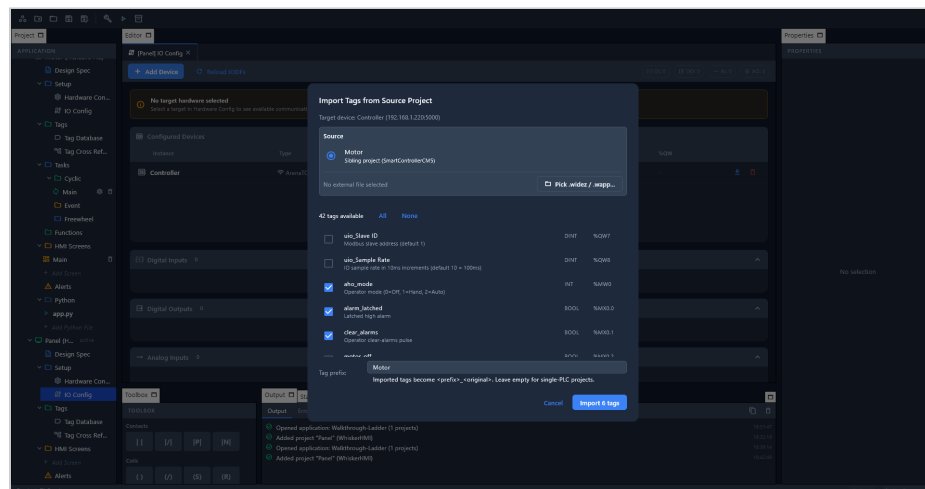
(If your source PLC lives in a separate `.widez` file — say, you’re building a panel for a controller whose project you don’t have open — click the **Pick .widez/.wapp** button instead and browse to the file.)

- The dialog lists every tag from the Motor project with its type and address. Tick the six HMI-facing tags:

Tag	Type
float_start	BOOL
float_stop	BOOL
float_alarm	BOOL
motor_run	BOOL
aho_mode	INT
clear_alarms	BOOL

Leave the five tags the HMI doesn’t display unticked: the four internal scratch tags (`motor_off` , `motor_hand` , `motor_auto` , `call_motor`) plus `alarm_latched` . None of them is bound to a widget — `alarm_latched` is the ladder interlock latch, `motor_run` is the actual outcome, and `aho_mode` already reflects the operator’s mode selection.

- The **Prefix** field defaults to `Motor` (the source project name). For this walkthrough there’s only one controller so collisions aren’t an issue — clear the prefix field to keep the original tag names. Step 5 imports the screen from `Motor` and the bindings reference the bare names (`motor_run` , not `Motor_motor_run`).



- Click **Import**. The dialog reports `6 tags imported` and closes.

6. Open **Panel** → **Tags** → **Tag Database** to verify. All six tags appear with the addresses they had in Motor (`%I0` , `%Q0` , `%MW0` , etc.). The Panel project records the `controller` device as each tag's source internally; at build time the IO scanner uses that link to read each tag from the controller's arena rather than from the panel's own (empty) local arena.

Why the prefix matters. When a single panel reads from multiple controllers — say three pump stations all running the same Motor app — every controller has its own `motor_run` , `aho_mode` , etc. Without a prefix the second import would collide with the first by name. The convention is to prefix each import with the controller's identity (`PS1_` , `PS2_` , `PS3_` or `Station1_` , etc.) so all 21 tags coexist in one panel.

Behind the scenes the IDE allocates each imported tag a unique slot in the panel's local arena. When you import a tag from PS2, the IDE finds the next free local address in that area — so `PS1_motor_run` and `PS2_motor_run` end up at *different* local slots even though both map to `%Q0` on their source controllers. At build time the compile-service emits a per-tag mapping (remote `%Q0` → local `%Q<whatever>`); the panel's IO scanner reads each remote arena and copies each tag's value into its mapped local slot. For a single-controller panel like this one, no prefix is fine — the imported tags get the same local addresses they had on the source.

27.6 Step 5 — Copy the HMI screen from Motor

The HMI screen you built in the Ladder Walkthrough Step 7 already exists in the Motor project; you don't need to rebuild it.

1. In the Project Tree, **right-click Motor** → **HMI Screens** → **Main** and choose **Copy**. A notification confirms the screen is on the clipboard.
2. **Right-click Panel** → **HMI Screens** and choose **Paste**. The Main screen appears under the Panel project with all five widgets and their bindings intact.

The paste also auto-remaps bindings to whatever prefix you chose in Step 4: if you imported with prefix `Motor_` , the binding that read `aho_mode` on Motor's screen is rewritten to `Motor_aho_mode` on the Panel's copy. A notification at the bottom confirms how many bindings were remapped. If multiple ArenaTCP devices in the Panel import from the same source project, the paste asks which controller the screen is for.

If you haven't imported the source tags yet, the paste is refused with a dialog listing what's missing. Import first, then paste again.

3. Under **Panel** → **HMI** → **Windows**, confirm that `Main` is the primary window — pasted screens inherit their source's window role, so this should be the case automatically.

27.7 Step 6 — Build the Panel runtime

1. Click anywhere inside the Panel project (e.g., select **Panel** → **HMI** → **Screens** → **Main** in the tree) so the IDE knows which project you want to build.
2. Press **F5**. The Output panel shows:

```
[INFO] Building project: Panel...  
[INFO] Generating screens.hmi (1 screen)  
[INFO] Generating symbols.json (6 tags)  
[INFO] Generating io_scanner_config.json (1 ArenaTCP source)  
[INFO] Generating app_config.json  
[OK] Build succeeded: 4 files generated
```

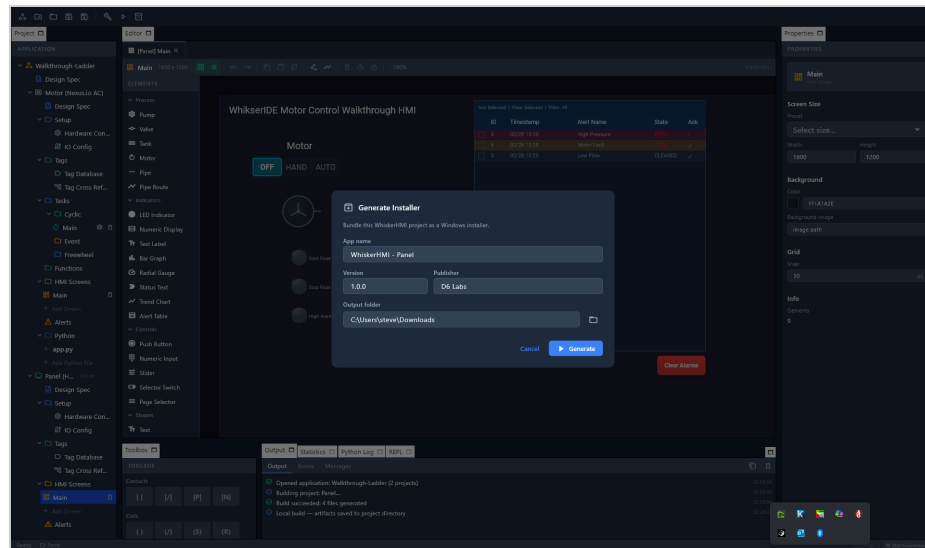
3. These four files are written to a temporary build folder, not inside your Application directory — you don't need to find or manage them. The next step (Generate Installer) bundles them into the installer automatically.

Build vs Generate Installer. Build produces the runtime configuration files. To run the panel on another PC, you generate a Windows installer that bundles the WhiskerHMI runtime executable together with the Panel build output — that's the next step.

27.8 Step 7 — Generate the Windows installer

1. Click the **Generate Installer** icon in the toolbar (a small package / box icon). Make sure the Panel project is selected; the button is only enabled for WhiskerHMI projects.
2. The Installer dialog opens. Fill in:
 - **App name:** `Walkthrough Motor Panel`
 - **App version:** `1.0.0`
 - **Publisher:** your name or company

- **Output folder:** somewhere convenient



3. Click **Generate**. The IDE invokes Inno Setup; after a few seconds the Output panel reports:

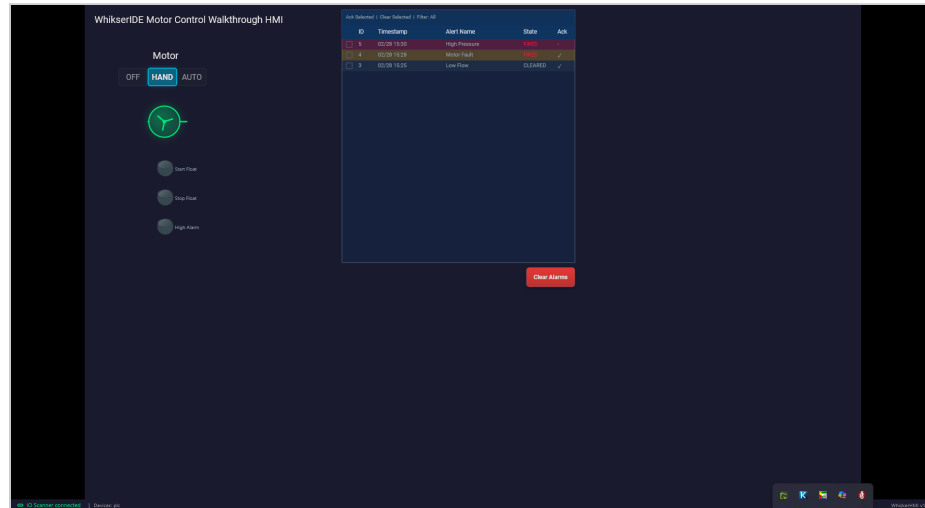
```
[INFO] Bundling WhiskerHMI runtime...
[INFO] Bundling project config (Panel)...
[INFO] Running Inno Setup...
[OK] Installer created: WalkthroughMotorPanel-
Setup-1.0.0.exe (24.7 MB)
```

4. The installer .exe lands in the output folder. That's the whole deliverable — a single file you can copy to any Windows PC.

27.9 Step 8 — Install and run on a PC

Copy the .exe to the PC where the panel will live (your laptop works for testing). Double-click it:

1. **User Account Control** prompts for permission — click **Yes**.
2. The installer wizard runs through License, Destination Folder, shortcuts, Install, Finish. The path shown on the Destination Folder page is informational only in the current installer layout — the app is always installed under the system Program Files area: `C:\Program Files\Walkthrough Motor Panel\`. (If you need to verify after install, that's where to look — not at whatever was shown on the wizard page.)
3. The HMI panel launches automatically. The window opens, fills with the Main screen layout, and starts polling the controller's arena agent.



If the screen comes up blank or shows “Connecting...” for more than a few seconds, the panel can’t reach the controller. Check:

- The PC and controller are on the same subnet.
- The IP address in Step 3 matches the controller’s current address (DHCP can move it).
- Port 5000 (the controller’s data port) isn’t blocked by a firewall between them.
- On Path A, the panel PC has a panel identity imported and the device has **Secure connection** ticked; a managed controller refuses a panel without a matching identity (see the box at the top of the chapter).

27.10 Step 9 — Test it

Tap the AHO selector switch from OFF → HAND on the Windows HMI panel. The motor LED in the Windows panel turns green — and at the same moment, the controller’s built-in touchscreen shows the same change because both panels are reading the same controller state.

Trigger the high-alarm float (physically or via tag forcing). The alarm latches on both panels. Press Clear Alarms on either panel and both clear.

That’s the point of WhiskerHMI: as many panels as you want, each showing the same live state, all writing back through the same arena agent. Run the panel on three workstations and each operator sees the same thing.

27.11 What's next

- Read *The WhiskerHMI PC Application* in the full manual for what the panel can do beyond this screen: several windows, the Trends dialog against the controller's historian, CSV export, and operator login with roles.
- Add a second screen to the Panel project (e.g., a trends screen showing motor run-time history) using the same Generate Installer flow — one installer, multiple screens.
- Build a *multi-controller* panel by adding a second ArenaTCP device pointing at a different controller and importing tags from that controller's project with a distinct prefix (e.g. PS2_). The same Panel can display state from any number of controllers side by side.
- Read *Security, Provisioning and Enrolment* and *The WhiskerHMI PC Application* in the full manual before shipping a panel into production — in particular what a secure panel connection needs.

28 Where to go next

You've built three projects that touch every major editor in the Whisker IDE — Tag Database, IO Configuration, Ladder, Python, Alerts, HMI Designer, and the Modbus server — plus the deploy pipeline for both the controller and the PC panel. You know enough now to build real things.

The rest of this section points at where to deepen each piece.

28.1 The full user manual

The Quick Start Guide you just finished is an extract from the full Whisker IDE User Manual, which ships as HTML and PDF alongside this guide. The full manual covers both editions of the IDE — Cloud and Offline — and says in each chapter what needs an account.

Chapter	What it adds to what you did here
The Workspace	every panel, toolbar button and bottom-panel tab (Output, Python Log, Tag Monitor)
Projects and Applications	multi-project Applications, what is stored where, projects stored on the controller
Setting Up Your Nexus.io Automation Controller	the touchscreen settings, network, serial ports, the controller's security postures
Target Hardware in the IDE	the Nexus.io Automation Controller, the SmartController, WhiskerHMI
Tag Database	the full address map, classes, initial values, process variables, retained values that survive a program update, system tags
I/O Configuration and Devices	every device kind — Modbus TCP/RTU, ArenaTCP (another controller's tags), EtherNet/IP PLCs
Ladder Logic Editor	the editor itself: networks, wiring, properties, tags from the Properties panel
Function Blocks, Faceplates and Tasks	reusable logic, faceplates, periodic / event / freewheel tasks

Chapter	What it adds to what you did here
HMI Designer	every widget, PIN login at the panel, trend charts, alert tables
Alerts Editor	conditions, latching, output tags, the alert table
Modbus Server and Map Viewer	exposing tags, renumbering, the four CSV export formats
Python on the Controller	the full <code>ctx</code> API, Python function blocks called from ladder, the SmartController's MicroPython
Connecting, Deploying and Emulating	discovery, the three controller postures, Build & Emulate with the IO Stimulus and HMI Preview tabs, what a signed deploy is
Monitoring and Debugging	Tag Monitor, Debug Mode, the Output panel, diagnosing a device that will not talk
Security, Provisioning and Enrolment	Path A and Path B in full, the Security dialog, certificates, revocation
Generating HMI Installers	packaging a WhiskerHMI panel for other PCs
Project Properties Reference	every project setting in one place
Ladder Instruction Reference	every instruction — timers, counters, compare, math, process blocks (PID, totalizer, filter...), motor and valve blocks, sequencer, scheduler
Historian and Trends	the controller's 60-day historian, trend widgets, CSV export
The WhiskerHMI PC Application	several windows, Trends, operator login and roles

28.2 Exposing tags to Modbus

The controller always runs a Modbus TCP **server** (slave) on port 502 — there's no switch to turn it on. Instead you choose *which* tags it exposes, one tag at a time, so a SCADA system, BMS, or any Modbus master can read and write them.

Expose a tag

1. Open the **Tag Database** and select the tag you want to expose.
2. In the **Properties** panel, check **Expose via Modbus**.
3. The IDE auto-assigns a Modbus address. To override it, type a different address in the Properties panel — a tag whose address you edit by hand is flagged as *manually assigned*, which the renumber tool (below) can leave untouched.

How tags map to registers

- **BOOL** tags are exposed *twice* — as a **Coil** and as a single-bit **Holding Register** — so every bool has two addresses.
- **INT** takes one Holding Register; **DINT** and **REAL** take two consecutive Holding Registers. The IDE allocates the registers for you.

Renumbering the map

To re-pack the addresses, select the **Project** node, and in its properties click **Renumber all auto-assigned tags**. In the dialog:

- Leave **Keep manually-assigned addresses** checked to renumber only the auto-assigned tags and leave your hand-edited addresses where they are; uncheck it to renumber everything.
- You can give each data type its own address range — for example, in the Holding Register space, `0-999` for bools, `1000-1999` for ints, and `2000-2999` for reals — so the map stays organized.

The Modbus Map

As soon as at least one tag is exposed, a **Modbus Map** node appears under the Project in the tree. It's generated automatically (you don't edit it directly) and shows two tables — **Holding Registers** and **Coils** — listing every mapped tag with its address, type, and access.

The **Export** button writes the map out as **Generic CSV**, **Ignition CSV**, **KEPServerEX CSV**, or **AVEVA System Platform CSV** — a big time-saver when you're pulling these tags into an existing SCADA system.

See *Modbus Server and Map Viewer* for the full address conventions and read-only vs. read-write rules.

28.3 The Design Spec

Every project has a **Design Spec** — a living document under the Project in the tree. It starts from a template with sections for Overview, IO Configuration, Control Logic, HMI Screens, Alarms, and Notes that you fill in as you design the system.

Two parts are **generated for you** and kept up to date automatically:

- **Tag Database** — tables of your tags, split into **IO Tags** (anything with a physical address), **Setpoints** (`sp_...`), **Internal Variables** (`state_...` , `var_...`), **Process Variables** (`cmd_...` , `call_...`), and a **Misc Tags** catch-all for everything else.
- **Modbus Registers** — Holding Register and Coil tables of every tag you exposed via Modbus (above), refreshed whenever the Modbus map changes.

These tables live inside marked blocks in the document. The IDE refreshes only what's *inside* the markers, so everything you write around them — your design narrative, control strategy, commissioning notes — is left untouched. Edit your tags in the Tag Database and the tables follow along.

28.4 Connecting to a controller

Click **Connect** in the toolbar, then pick the controller from the discovered list or enter its address. You don't need a project open to connect.

When you connect, the IDE reconciles the project you have open against the project stored on the controller:

- **No project open, controller has one** — the IDE offers to **download** the project from the controller and open it. Decline, and the connection is cancelled (there'd be nothing to work with).
- **Your open project matches the controller's** — it simply connects.
- **They differ** — the IDE asks whether to **Download from target** (this closes your open project, prompting you to save first) or keep the one you have. If you keep yours, live monitoring stays off until you Build, because the tag addresses wouldn't line up with what's actually running.
- **Deploying** a project to a controller that already holds a *different* project prompts you to confirm before it's replaced.

Because the controller stores the editable project source, you can pull a project straight off a device even without a local copy. For IP-sensitive deployments, turn this off per project under **Project Properties** → **Deployment** → **Store editable source on target**.

See *Connecting, Deploying and Emulating* for discovery details, the three controller postures and Build & Emulate, and *Security, Provisioning and Enrolment* for Path A and Path B in full.

28.5 Sample projects

The IDE ships with the two completed walkthrough projects. On first launch it copies them into your default project folder — `Documents\Whisker Projects` — which is also where **File** → **Open** and **Save As** start:

- **walkthrough_ladder.widez** — the completed ladder version of this walkthrough’s pump-station scenario.
- **walkthrough_py.widez** — the same scenario built in Python instead of ladder.

Open either with **File** → **Open**; it opens straight to `Documents\Whisker Projects`.

28.6 When you get stuck

- **A build that fails** names the program and network of each error in the Output panel; *Troubleshooting* lists the common causes.
- **The Output, Tag Monitor and Python Log tabs** are your live view into the controller, right inside the IDE. **Output** shows build results and runtime messages; **Tag Monitor** shows every tag’s current value; **Python Log** streams `ctx.log` output from your Python code as it runs. All update live while you’re connected, and every entry is selectable and copyable for pasting into a bug report.
- **Troubleshooting** in the full manual’s appendices is organized by symptom (the controller refuses to connect, the deploy was refused, the Tag Monitor is empty, the motor doesn’t run...).

28.7 Support

- **Email support:** `support@d6labs.com`
- **Phone (US business hours):** 1-844-365-8647

28.8 What we'd love to hear about

D6 Labs prioritizes new IDE features based on what users actually build. If you've extended this walkthrough into something interesting — a real production lift station, a hybrid Python/ladder scheme, an unusual WhiskerHMI deployment — drop us a note. Screenshots and brief descriptions go a long way.

Happy building.

Appendix A — Keyboard Shortcuts

Every keyboard shortcut the IDE binds, grouped by where it works.

28.9 Global

These work anywhere in the main window.

Shortcut	Action
Ctrl+N	New Project (within the open Application)
Ctrl+Shift+N	New Task
Ctrl+Alt+N	New Function
Ctrl+O	Open Application
Ctrl+S	Save
Ctrl+Shift+S	Save As
F5	Build (for a Nexus.io project: compile and download to the connected controller)
F7	Run Mode (connected Nexus.io controller)
Shift+F7	Stop Mode (connected Nexus.io controller)
F9	Debug Mode (connected Nexus.io controller; monitoring subject to the gate in <i>Connecting and Deploying</i>)
Ctrl+Shift+L	Reset the panel layout to the default

28.10 Ladder editor

The ladder canvas must have focus (click a network first).

Shortcut	Action
Left / Right	Move the selection along the network

Shortcut	Action
Ctrl+Down	Start a branch below the selected segment
Ctrl+Up	Join the branch back to the level above
Ctrl+Shift+Up	Join the branch back to the main line
Ctrl+Right	Extend the branch to the right
Delete	Delete the selected element; on an empty segment, remove its branch connection; on the output, clear the coil or block
Ctrl+Delete	Delete the whole network
Ctrl+Z / Ctrl+Y	Undo / Redo
Ctrl+C / Ctrl+X / Ctrl+V	Copy / Cut / Paste the highlighted element, block or network (see <i>The Ladder Editor</i>)
Escape	Clear the selection

Adding a network, and cut, copy and paste, are buttons on the ladder editor's own toolbar (**Add Network**, **Cut**, **Copy**, **Paste**).

28.11 HMI Designer

Shortcut	Action
Arrow keys	Nudge the selection 1 pixel (ignores the grid)
Shift+Arrow	Nudge 10 pixels
Delete / Backspace	Delete the selection
Ctrl+A	Select all
Ctrl+C / Ctrl+V	Copy / Paste
Ctrl+D	Duplicate
Ctrl+G / Ctrl+Shift+G	Group / Ungroup
Ctrl+L	Lock or unlock the selection
R	Rotate the selection
] / [	Bring forward / Send backward

Shortcut	Action
Ctrl+] / Ctrl+[	Bring to front / Send to back
Ctrl+Z / Ctrl+Shift+Z or Ctrl+Y	Undo / Redo
Enter	Finish drawing a polyline or pipe
Escape	Cancel drawing, otherwise deselect all

28.12 Python editor

Standard text-editing keys apply (cut, copy, paste, select all, Home, End, Ctrl+Home, Ctrl+End). In addition:

Shortcut	Action
Ctrl+F	Find (Escape closes the find bar)
Tab / Shift+Tab	Indent / outdent the current line or selected block by four spaces
Ctrl+/	Toggle a <code>#</code> comment on the line or selection
Enter	New line keeping the current indent, one level deeper after a line ending in <code>:</code> , <code>(</code> , <code>[</code> or <code>{</code>
Ctrl+V	Paste, converting tabs to four spaces

28.13 Design Spec editor

Shortcut	Action
Tab	Insert two spaces

28.14 SmartController Classic Console

Shortcut	Action
Up / Down	Recall earlier commands

Shortcut	Action
Ctrl+C	Interrupt the running MicroPython program

28.15 Mouse

Action	Behaviour
Click a tree node	Select it and open its editor
Double-click a ladder element	Bring the Properties panel to the front
Drag a panel tab or header	Re-dock the panel or float it
Drag a splitter	Resize the panels either side
Right-click	Context menu (tree nodes, HMI screens, Output lines)

Editor tabs are switched by clicking and closed with their ✕; there are no keyboard bindings for tab navigation.

Appendix B — File Formats Reference

The files the IDE reads, writes, deploys and keeps on your PC. You never need to edit any of them by hand; this appendix is for recognising what a file is when you meet it in a folder, a backup or a support request.

28.16 Application and project archives

An **Application** is saved as a single `.widez` archive (a zip; rename a copy to `.zip` to look inside). It contains `application.yaml`, which names the application and lists its projects, and one folder per project under `projects/`. A project can also be saved on its own as a `.widez` whose top level is the project folder itself. Older applications used the extension `.wapp`; the Open dialog still accepts them and the next Save writes `.widez`.

Inside a project folder:

Path	Contents
<code>project.yaml</code>	Name, project GUID, target, deployment and cloud settings, PIN login settings, historian retention, <code>site_id</code>
<code>symbols.yaml</code>	The tag database (format <code>symbols/2</code>)
<code>tasks.yaml</code>	Task definitions (format <code>tasks/1</code>)
<code>functions.yaml</code>	Function block definitions (format <code>functions/1</code>)
<code>io_map.yaml</code>	Devices and point bindings
<code>programs/*.lad.json</code>	One ladder program per file
<code>hmi/</code>	HMI screens, one JSON file per screen
<code>alerts.yaml</code>	Alert definitions (format <code>alerts/1</code>)
<code>python/</code>	Python source (<code>app.py</code> and any modules)

Path	Contents
<code>design_spec.md</code>	The project's design specification (the Design Spec node), when one has been written

The IDE rewrites the whole archive on every Save. Edits made inside the archive with another tool are lost the next time the project is saved from the IDE.

28.16.1 `project.yaml` keys worth knowing

Key	Meaning
<code>project_guid</code>	Stable identity of the project; Save As creates a new one. The controller stores it with the deployed program so the IDE can tell at connect time whether you have the same project open.
<code>store_source_on_target</code>	The Store editable source on target switch. <code>true</code> deploys the editable source with the program so it can be pulled back.
<code>site_id</code>	The site identity this project connects and signs with (Project Properties, Security). Absent means Automatic .
<code>target_hdl</code>	The hardware definition the project targets.

28.17 `symbols.yaml`

Format `symbols/2`: 16-bit `INT` and 32-bit `DINT` are distinct types with their own memory areas. Files in the older `symbols/1` format are migrated on open. Each entry under `globals` carries only the keys that differ from their defaults:

```

format: symbols/2
globals:
  - name: motor_start
    type: BOOL
    class: VAR
    init: false
    address: "%MX0"
    description: Operator start button
  - name: tank_level_pct
    type: REAL
    class: VAR
    address: "%MF0"
    pv: 0
    units: "%"
    hist: true
    mse: true
    msh: 100
  
```

Key	Column in the Tag Database
name , type , class , init , address , description	Name, Type, Class, Initial, Address, Description
pv	PV# — the cloud process-variable slot
units , gmin , gmax , dp , dgain , dofs , disp , dispif , incr	Cloud widget settings for the PV
mse , msh , msc , mshm , mscm	Expose, and the Modbus server holding- register / coil address (manual flags when you typed the address)
hist , histdb	Hist, and the historian deadband
atcp , atcpsa , atcpsn	Source — the ArenaTCP device and remote tag the value comes from
system: true	A system tag the IDE maintains (Comm OK , Comm Age , Error Type , cloud and cellular tags)

28.18 Build output

Build (F5) writes these into a build folder and, in the same action, sends them to the connected controller. On a managed controller they travel inside the signed bundle; on a standalone controller they are sent one by one.

File	Purpose
<code>app.ilbc</code>	Compiled ladder bytecode
<code>app.pi</code>	Program image: task assignments, network counts
<code>tasks.tbl</code>	Task table
<code>app.fbmap</code>	Function block instance map
<code>app.retain</code>	Retained-variable layout and initial values. Values are migrated by name across deploys, so a renamed retained variable starts fresh.
<code>app.init</code>	Initial values for non-retained variables, applied at each program load. Both files share one header format, version 6, whose records carry the variable names the migration keys on.
<code>app.io</code>	I/O scanner configuration (devices, points, polling)
<code>app.pv</code>	Process-variable index for the cloud service
<code>app.sym</code>	Symbol table for monitoring
<code>app.cfg</code>	Project configuration: HMI settings, PIN login and the roles table
<code>app.meta.json</code>	Build metadata (timestamp, IDE version)
<code>app.modbus_map.json</code>	Modbus server tag-to-register map for tags marked Expose
<code>app.cip_map.json</code>	CIP tag server map: whether the server is on, and each published tag's name, type, location and writeable flag
<code>screens.hmi</code>	Compiled HMI screens
<code>sc_config.json</code>	Cloud service configuration
<code>historian.json</code>	The on-controller historian's tag list: version, <code>retention_days</code> from Project Properties, sample period, and one entry per tag with Hist ticked or used by a trend chart

File	Purpose
<code>python_app.json</code>	The Python application bundle (<code>app.py</code> and its modules), when the project has one
<code>hmi_auth.json</code>	The device credential for PIN login at the panel, minted fresh on every deploy when Cloud PIN Login is on. Never stored in the project.
<code>project.widez</code>	The editable source, sent when Store editable source on target is on
<code>project.meta.json</code>	The project's identity stamp: GUID, name, content hash, artifact hash, deploy time and who deployed

28.18.1 The signed bundle

For a managed controller the build folder is packed and signed before sending:

File	Purpose
<code>bundle.tar</code>	The build folder as one deterministic archive
<code>bundle.tar.sig</code>	Detached ECDSA P-256 / SHA-256 signature
<code>bundle.tar.sig.json</code>	Manifest: signer name and fingerprint, payload hash and size, timestamp
<code>signer.cert.pem</code>	Your certificate
<code>ca_chain.pem</code>	The authority's chain. Always sent; a controller uses it only on its first signed deploy, to bind itself to that authority

The controller verifies the manifest, signature, chain and revocation status before it extracts anything. A refused bundle changes nothing on the controller.

28.19 Provisioning and security files sent to a controller

File	Sent by	Contents
<code>bootstrap.json</code>	Provision New Device, Approve Re-Enrollment	<code>cloud_url</code> , <code>account_id</code> , <code>device_serial</code> , <code>bootstrap_nonce</code> — the one-time nonce the controller uses to enrol. Deleted by the controller once it has enrolled.
<code>identity.json</code>	Provision unit (site)	The device certificate and key issued by the site CA, the site chain, the current CRL, and the site id and name. The controller installs it only if the certificate's name matches its serial and the chain matches any anchor it already has.
<code>crl.pem</code>	Every connect with a site identity, and Push CRL to unit	The site's certificate revocation list, installed only when newer than the controller's and signed by its trusted authority

28.20 The keystore on your PC

The IDE keeps certificates in a keystore folder under your user profile (`.whisker-ide/security` ; the Security dialog shows the full path in its title bar). Private keys in it are yours: back the folder up and do not share it.

```

security/
  accounts/<account id>/  your cloud-issued IDE certificate and
                           key, the account's CA chain,
                           its CRL and a meta.json – written at
login, refreshed automatically
  sites/<site id>/        one folder per imported site identity:
site.json, ca_chain.pem,  site.json, ca_chain.pem,
                           crt.pem, your user certificate and key,
meta.json
  root/                  the local root CA certificate and key
(bench units, site CA admin)
  intermediate/          the local intermediate CA certificate and
key
  user/                  the local signing certificate and key
  devices/<serial>/        certificates issued to bench units or
site-provisioned units
  users/<e-mail>/          engineer certificates issued by this site
CA (admin PC only)
  issued/                the site CA's ledger, one record per
issued certificate (admin PC)
  ca_chain.pem, crt.pem   the local chain and CRL
  site.json              present when this keystore has been made
a site
  
```

On a site administrator's PC the `root/` and `intermediate/` keys are encrypted with the CA passphrase. On an engineer's PC the `sites/` folder holds only what the administrator packaged: no CA keys.

28.20.1 Site packages

File	Contents	Secret?
<code><name>.whisker-site</code>	<code>site.json</code> , <code>ca_chain.pem</code> , <code>crl.pem</code>	No — send it freely; it carries the current revocation list
<code><e-mail>.<site id>.whisker-identity</code>	The three above plus the engineer's certificate and private key, the key encrypted with the identity passphrase, and <code>identity.json</code>	Yes — the passphrase must travel separately

28.21 Export formats

- **Modbus map CSV** — the Modbus Map node exports the server's register map in Generic, Ignition, KEPServerEX and AVEVA layouts; see *Modbus Server and Map Viewer*.
- **Tag Monitor TSV** — the Tag Monitor's copy button puts the visible rows on the clipboard as tab-separated text with the columns Name, Type, Address, Value, Description.
- **Output panel** — Copy All produces one line per message with a time and a [INFO] , [WARN] , [ERROR] or [OK] prefix.
- **WhiskerHMI trend CSV** — see *Historian and Trends*.

Appendix C — Troubleshooting

Symptoms, causes and fixes, grouped by where in the workflow they appear. Every fix here is done from the IDE; none needs a terminal session on the controller.

28.22 Installing and starting

28.22.1 Windows warns that the installer might be dangerous

The installer is not code-signed. Choose to keep the file and run it.

28.22.2 The IDE starts, then closes at once

Something failed during start-up. Start it again from a command prompt so the console shows the error, and include that text when you ask for help (see *Getting Help*).

28.23 Connecting

28.23.1 The Connect to Target dialog finds nothing

- The controller is off, still booting or not on your network. Give it a minute after power-on.
- Your PC is on a VPN or on a different subnet from the controller. mDNS does not cross either, and the subnet probe scans your PC's subnet, not the controller's. Expand **Enter address manually** and type the controller's address.
- Multicast is blocked on the PC or the switch. Manual entry works regardless.
- A project is open whose target type does not match the controller; the dialog lists only matching controllers (*Looking for: ...*). Open the right project or connect with none open.

28.23.2 The row says “Awaiting enrolment”

The controller is configured to require management and has not been made managed yet: either it was ordered as a locked-down unit, or it was decommissioned. It accepts a connection but no program. Make it managed: with

a cloud account, connect and use **Provision New Device** in the Cloud panel (**Approve Re-Enrollment** for a decommissioned device); without one, the site administrator connects and uses **Provision unit (site)**. See *Security, Provisioning and Enrolment*. A new controller that was not ordered locked down shows **Standalone** instead and needs none of this.

28.23.3 I connected plainly and want the controller managed

The row says **Standalone**: the controller accepts your connection and your deploys, and would accept any other IDE's too. That is the shipping posture, and it is fine to leave it there. To make the controller managed instead: with a cloud account, connect and use **Provision New Device** in the Cloud panel; without one, set up the site CA on the administrator's PC and use **Provision unit (site)**. Either way the controller restarts managed, the plain connection drops, and you reconnect with your account or site identity. See *Security, Provisioning and Enrolment*.

28.23.4 “Connection failed” with a TLS or handshake error

The controller is managed and did not accept the identity your IDE presented. In order of likelihood:

- **Wrong account.** The controller is enrolled into a different Whisker.io account from the one you are logged in to. Log in to the account that enrolled it.
- **Not logged in, no site identity.** The IDE fell back to its local keystore, which a managed controller does not trust. Log in, or import the site identity your administrator issued.
- **Wrong site.** More than one site identity is imported and the project's **Site identity** setting names the wrong one (or **Automatic** fell back to the local keystore). Set it in Project Properties.
- **Revoked or expired.** Your certificate is on the revocation list or has expired. In the Security dialog use **Re-issue** (account) or ask the administrator for a new identity (site).
- **No certificate yet.** Open the Security dialog; if the Account section says no IDE certificate is on this PC, click **Re-issue**.

28.23.5 Connecting by address falls back to plain and then is refused

A managed controller reached by address: the IDE tried TLS with your identity, was refused, tried plain, and the controller refused that too. Fix the identity as above; the plain attempt succeeds only on a standalone controller.

28.23.6 “Connected” but the toolbar buttons stay grey

No project is open, or the controller’s target type does not match the project. Open the right project; the connection is dropped and re-made when you switch projects.

28.23.7 The connection drops as soon as the controller enrolls or is provisioned

Expected. The controller restarts its services in managed mode, which closes the plain connection. Click **Connect** again; the row now says **Managed**.

28.23.8 The connection drops repeatedly for no visible reason

Only one IDE can be connected to a controller at a time; check nobody else is. A PC switching between Wi-Fi and cable, or a VPN that resets idle sessions, produces the same symptom.

28.24 Enrolling and provisioning

28.24.1 Provision New Device showed a nonce dialog

The IDE was not connected to that controller when you clicked **Provision**, so it could not deliver the enrolment bootstrap. Click **I’ve recorded it** (the nonce itself is not needed), then connect to the controller and choose **Approve Re-Enrollment** on the device in the Cloud panel. The IDE mints a new nonce and delivers it over the connection.

28.24.2 Enrolment never completes

The status stays at *Waiting for enrolment...* and times out after two minutes.

- The controller cannot reach the cloud. It needs a route to the internet over Ethernet or its cellular modem; the bootstrap stays on it and it keeps trying, so fix the connectivity and wait, then check the device’s state in the Security dialog’s controllers table.
- The device is **Decommissioned** in the cloud; enrolment is refused. Use **Approve Re-Enrollment**.

- The serial in the cloud does not match the controller's. The dialog fills the serial from the connected controller for this reason; if it was typed, **Edit Device** and correct it, then **Approve Re-Enrollment**.

28.24.3 Provision unit (site) says “Connect to the unit first”

The action needs a plain connection to a controller that is standalone or awaiting enrolment. A managed controller cannot take a new identity: revoke its current one in the Units table and decommission it before provisioning again.

28.24.4 Provision unit (site) is refused with “bound to another CA”

The controller was provisioned by a different site (or enrolled into a cloud account) before. It will only accept identities from the authority it already trusts. It has to be decommissioned under that authority first.

28.24.5 The Site card warns the revocation list is N days old

Ask the administrator for a fresh `.whisker-site` package and import it with **Update trust package....** Until then a revoked engineer might still be accepted by controllers you have not visited.

28.25 Building and deploying

28.25.1 Build is disabled

For a Nexus.io project Build is enabled only while connected to a controller, because it deploys in the same action. Connect first, or use **Build & Emulate** to check that the project compiles.

28.25.2 “Deploy stopped: This project requires a PIN at the panel...”

The project has **Cloud PIN Login** on, and every deploy fetches a device credential from Whisker.io, which needs you logged in. Log in and deploy again. For a controller that runs without Whisker.io, switch **Cloud PIN Login** off in Project Properties (it can always be turned off) and deploy. See *Project Properties Reference*.

28.25.3 “N tags are used but not defined”

The ladder references names the tag database does not hold. **Create All** creates them with the suggested types (check the types — a wrong one compiles and gives wrong numbers) and the build continues. **Cancel** stops the build. The same sweep warns on Save without blocking it.

28.25.4 Build failed with compiler errors

Each error names the program and network. Fix it in the ladder editor.

28.25.5 The deploy is refused: signature, chain or “signed deploys only”

The controller is managed and would not install what was sent. Its previous program keeps running.

- *Managed controller accepts signed deploys only* — the IDE sent an unsigned program because it connected plain. Reconnect over TLS with the right identity.
- *no user signing certificate* — the IDE has nothing to sign with. Log in (account) or import your site identity.
- *signer cert not signed by trusted ... CA* — you signed under a different authority from the one the controller trusts (another account, another site, or the local keystore). Connect with the identity that matches the controller.
- *payload hash mismatch* — the bundle was altered or corrupted in transit. Build again.
- *certificate revoked* — your certificate is on the controller's revocation list. Re-issue or get a new identity.

28.25.6 Any IDE on the network can deploy to my controller

The controller is standalone, which is how it ships and is fine for many sites. If yours needs signed deploys from known engineers only, make it managed: see / *connected plainly and want the controller managed* under *Connecting* above.

28.25.7 “Replace project on target?”

The controller holds a different project from the one you are deploying. **Replace** overwrites it; **Cancel** keeps it. If you meant to work on the controller's project, connect again and choose **Download from target** when offered.

28.25.8 The deploy succeeds but the program does not change

Check the Output panel: the deploy ends with *Program downloaded and running* only when the reload and start succeeded. A *Reset timed out* line followed by success is normal. If the program still looks old, edit something trivial, save and **Build (F5)** again so the sweep, compile and deploy all run.

28.25.9 “The VM would not start”

The program loaded but refused to run. A project with no ladder logic (Python only) leaves the ladder runtime stopped on purpose and this is not an error. Otherwise the Output panel carries the runtime’s reason.

28.26 Monitoring

28.26.1 Monitoring shows nothing, the monitoring button is disabled

The gate is closed. Hover the button; its tooltip names the reason. Almost always it is *This project has not been downloaded to the target yet* — press **Build (F5)** — or you edited the project after the last deploy, which closes the gate until the next deploy.

28.26.2 The Tag Monitor is empty or every value is a dash

- *No project loaded* — open the project.
- *Not connected* — *no live values* — monitoring is off (see above).
- Rows are missing — the filter is set, or the eye toggle is hiding system tags. Only tags with an address are listed.
- Values show ? — the tag’s address is in a form the monitor cannot read; check the address in the Tag Database.

28.26.3 The controller holds a different project

The connect-time reconcile said so and you chose **Keep my project**, so monitoring is off. Either deploy your project with **Build (F5)** (you will be asked to confirm the replacement) or connect again and choose **Download from target** to work on the controller’s copy.

28.26.4 A value will not change when the ladder should write it

It is forced. Look for the amber **FORCES (N)** chip in the ladder editor header and clear the force from the Properties panel.

28.27 Emulation

28.27.1 The emulator will not start

- *No Python interpreter found on PATH* — install Python 3.9 or newer and restart the IDE.
- *Could not find ... run_emulator.py* — the emulator files are missing from the installation; reinstall the IDE.
- *Port 9000 ... is already in use* — an emulator from an earlier session is still running. Press **Stop Emulation**; if the button is not active, end the leftover Python process from Task Manager and try again.
- *Build failed — nothing to emulate* — fix the build errors first.

28.27.2 “Python application not running: ...” after Build & Emulate

The ladder is emulating but the project’s Python runtime did not start; the reason follows the colon and the [py] lines above it in the Output panel carry the runtime’s own message. - *Port 9001 (Python Log) is already in use* — a Python runtime from an earlier emulation is still running, or another program holds the port. Stop it and try again. - *Could not find ... run_python_app.py* or *Could not find software/smartcontroller/app_runner.py* — the runtime files are missing from the installation; reinstall the IDE. - *The Python runtime exited during startup* — read the [py] lines: usually a syntax error in a project file that the bundle check did not catch, or an import of a module that is not on the PC. The same file would fail on the controller.

28.28 Runtime behaviour

28.28.1 A Modbus client sees no registers

- The tag is not marked **Expose** in the Tag Database. Only exposed tags are in the server map.
- The project was not deployed after the Expose change. **Build (F5)**.
- The client is polling the wrong address or unit ID. Export the map from the Modbus Map node and compare; watch for 0-based versus 1-based addressing in the client.
- The controller is awaiting enrolment (a unit configured to require management): its data port is closed until it is enrolled or provisioned.

See *Modbus Server and Map Viewer*.

28.28.2 Modbus RTU modules on an RS-485 port never answer

- The bus is wired as the terminals are printed. The **A** and **B** labels on the controller's RS-485 connectors are reversed: bus A (D+) goes to the terminal printed B. Wired as printed the controller polls and nothing answers; every device on the port shows a timeout in its **Error Type** tag.
- The devices are on the other port. `rs485` is the connector labelled RS485A and `rs485_2` is RS485B; check the **Port** of each device in I/O Configuration.
- Baud rate, parity or stop bits on the port differ from the modules' own settings. Every module on one port must match the port's settings.
- Two modules share a slave ID.

See *I/O Configuration and Devices*.

28.28.3 InTouch changed dotted tag names to underscores

InTouch tag names cannot contain a dot. Keep the underscore in the tag name and put the dotted CIP Name in the tag's **Item**: that is the name ABCIP sends, so the controller sees it unchanged. See *CIP Tag Server*.

28.28.4 An EtherNet/IP driver cannot connect, or sees no tags

- The project's **CIP tag server** switch is off, or it was turned on after the last deploy. **Build (F5)**.
- Nothing is published. The Project Settings section counts the published tags; publish them in the Tag Database's **CIP** column or in tag Properties.
- The driver cannot reach TCP port 44818. Check any firewall between the SCADA PC and the controller.
- The controller is awaiting enrolment (a unit configured to require management): its data port is closed until it is enrolled or provisioned.

28.28.5 The EtherNet/IP driver connects but every read fails

The driver is in a Logix optimisation mode the controller does not provide. On an AVEVA ABCIP Logix5000 device set **Optimization Mode** to **No optimization** and restart the driver.

28.28.6 A value poked from the SCADA snaps back

The controller refused the write: the tag is published read-only, or it is physical I/O, which can never be written over CIP. The controller's log names the tag. Make a memory tag writeable in its Properties and let the ladder drive the output.

See *CIP Tag Server*.

28.28.7 The historian or a trend chart is empty

- The tag has no **Hist** tick (a tag on a trend chart is recorded anyway).
- The project was not deployed after ticking it.
- Not enough time has passed; samples are recorded once a second and the chart's time window may start before the deploy.
- **History Retention (days)** in Project Properties is shorter than the range you are looking at.

See *Historian and Trends*.

28.28.8 The panel's status bar shows “No cell”

- The modem is not attached to a network: check the antenna and the SIM.
- The controller runs without the cloud (standalone). The cell values come from the controller's cloud service, so a standalone controller shows **No cell** even with a working modem.
- **--** in place of a number means the modem has not reported that value yet; it fills in within about a minute.

28.28.9 An alert will not clear

It is latched, and stays active until acknowledged at the panel. Otherwise its condition is still true; watch the condition's tags in the Tag Monitor.

28.28.10 PIN login at the panel is locked out

This PIN is locked for N minutes. Other users can still log in. Five wrong PINs lock that user; the lock doubles on each further cycle up to thirty minutes. Wait it out, or log in as another user. An administrator can **Reset PIN** for the user in Whisker.io. If the panel says PIN login is not enabled, the organisation's PIN policy is off in Whisker.io, or the project was deployed with **Cloud PIN Login** switched on before the policy was enabled.

28.28.11 The panel shows a blank screen after a deploy

The HMI was not part of the build. Save the project and **Build (F5)** again; every deploy sends the compiled screens.

28.29 WhiskerHMI panel identity

A WhiskerHMI panel connecting to a managed controller (cloud-enrolled or site-provisioned) must present a panel identity issued by that controller's authority; see *The WhiskerHMI PC Application*. A panel against a standalone controller needs none of this — leave *Secure connection* unticked.

28.29.1 “No panel identity installed on this PC — use Import panel identity... in the panel's menu”

The I/O scanner's reason for the *not connected* banner: the panel was built for a managed controller (the message ends with the authority the build trusts) and no identity has been imported on this PC. Have the engineer issue one from the Security dialog's **Panels** sub-section and import it from the panel's **Panel ▸ Import panel identity...** menu. The scanner restarts and connects.

28.29.2 Import failed: “passphrase incorrect”

The passphrase does not decrypt the key in the package. It is the **Panel passphrase** the engineer typed when issuing, not the CA passphrase and not a login. Ask the engineer; if it has been lost, issue a new identity.

28.29.3 Import failed: “this panel was built for another authority”

The package was issued by a different certificate authority from the one the panel's configuration was built for — a site-issued identity on a panel built while logged in to a cloud account, another account, or another site. Either issue the identity from the authority the controller trusts and the panel was built for, or rebuild the panel for the right authority (log in to the account, or set the project's **Site identity**), regenerate the installer and install it again, then import.

28.29.4 Import failed: the certificate is expired or not yet valid

station certificate expired on ... or is not valid until ...: the identity is outside its validity dates. Issue a new one with a fresh **Validity (days)**; if the date looks wrong, check the panel PC's clock.

28.29.5 The panel was connected and is now refused

The banner returned and the scanner reports the TLS handshake refused. Either the panel's identity has been revoked in the Security dialog and the controller has received the updated revocation list (cloud: within fifteen minutes; site: with an engineer's next connect or **Push CRL to unit**), or the identity has expired. A

revoked identity cannot be reinstated: issue a new one and import it on the PC. If the controller was decommissioned and provisioned under a different authority, rebuild the panel for that authority as well.

28.29.6 The Panel menu is missing from the status bar

Only the main window has it, because that window owns the I/O scanner. Import from the main window; popup and auto-open windows follow it.

28.30 SmartController (Classic)

28.30.1 The controller restarts every 30 seconds with a blank screen

The project has no process variable: the controller's display task stops at once and the controller's own watchdog restarts it. Give a tag a PV slot in the Tag Database, then **Sync** and **Restart controller**. The IDE refuses to Sync or Build such a project, but a controller loaded from an older IDE may still be in this state.

28.30.2 The console shows nothing, or “Connected” but nothing answers

The controller restarted underneath the IDE (a power blip, its watchdog, an unplugged cable) and the connection was lost. The toolbar then shows the connection lost; click **Connect** again. If the port cannot be opened, another program (a serial terminal such as Thonny or PuTTY) has it — close that program first.

28.30.3 Sync failed: “could not read the device's /manifest.json”

The controller's record of what is installed could not be read reliably, so Sync stopped without changing anything rather than write a new record over it. Try again; if it persists, restart the controller and reconnect.

28.30.4 After a Sync the controller still runs the old program

Restart it: **Restart controller** in the console. A folder named **app (legacy)** in the Target tree also hides the installed program; the next Sync removes it.

28.31 When all else fails

1. Save the project and restart the IDE.
2. Copy the Output panel (**Copy All**) — it holds the build, deploy and connect history.
3. Copy the Tag Monitor (its copy button) if values are the problem.
4. Send both with the `.widez` and a description of what you expected; see *Getting Help*.

Appendix D — Glossary

Definitions of terms used throughout the manual.

Acknowledgement. Operator action on an active alert in the alert table. Whether it clears the alert depends on the alert's latch flag. See *Alerts Editor*.

Application. The file you open and save: a `.widez` archive holding one or more Projects and a design spec. See *Projects and Applications*.

Arena. The controller's shared in-memory tag store, divided into named areas (DI, DQ, AI, AQ, MI, MR) with fixed sizes. Every part of the controller reads and writes the same arena. See *Target Hardware in the IDE*.

ArenaTCP. The network protocol by which one Nexus.io controller, or a WhiskerHMI PC application, reads and writes another controller's tags. Added as a device in I/O Configuration. See *I/O Configuration and Devices*.

Awaiting enrolment. The posture of a Nexus.io controller configured to require management before it does anything: no identity, and an IDE port that accepts only connection, status and the enrolment bootstrap or a site identity until it has been made managed. A locked-down configuration some sites order, and the posture a decommissioned controller returns to; not how a new controller ships (see **Standalone**). Shown as a label in the Connect dialog. See *Security, Provisioning and Enrolment*.

Bootstrap. The one-time file the IDE writes to a controller during cloud provisioning, from which the controller enrolls and obtains its certificate. See *Security, Provisioning and Enrolment*.

Build. Compiles the open project. For a Nexus.io project the Build button is enabled only while connected and, on success, downloads the program to that controller in the same action. See *Connecting, Deploying and Emulating*.

Build & Emulate. Compiles the project and runs it on the PC in the emulator, with no controller. See *Connecting, Deploying and Emulating*.

Cloud Explorer. The **Cloud** tab beside the project tree (Cloud edition): your account's locations and devices, and the entry point for **Provision New Device** and the certificate actions. See *Security, Provisioning and Enrolment*.

Cloud edition / Offline edition. The two editions of the IDE. Cloud signs in to Whisker.io; Offline (installed as *Whisker PLC IDE Standalone*) has no account and no cloud features. See *Getting Started*.

CMP. The single comparison block in ladder, with LT, EQ and GT outputs. See *Ladder Instruction Reference*.

Coil. A ladder element that writes a BOOL tag from the state of its network. One coil per network. See *Ladder Logic Editor*.

Contact. A ladder element that reads a BOOL tag: normally open conducts when the tag is true, normally closed when it is false.

CRL. Certificate revocation list. Lists the identities an authority has revoked — engineer, unit and panel identities of a site CA, IDE, device and panel certificates of a cloud account. Enrolled controllers fetch the account's list every fifteen minutes; every IDE pushes a newer site list to each controller it connects to. See *Security, Provisioning and Enrolment*.

Cyclic task. A task that runs every N milliseconds. The default `Main` task is cyclic at 10 ms. See *Function Blocks, Faceplates and Tasks*.

Debug Mode. Controller mode like Run Mode but with values streamed to the IDE. See *Monitoring and Debugging*.

Deploy. Getting a program onto a controller: for a Nexus.io project, press Build while connected — the IDE compiles and downloads the program (the Output panel calls this “Build + Download”). A managed controller accepts only a bundle signed by an authorised IDE.

Design Spec. The Markdown design document attached to every Application and Project, with tag and Modbus tables the IDE maintains. See *Projects and Applications*.

Device. External I/O the controller communicates with: a Modbus RTU or TCP module, another controller over ArenaTCP, or an EtherNet/IP PLC. Adding a device creates its tags. See *I/O Configuration and Devices*.

DHCP. Dynamic Host Configuration Protocol; a controller in DHCP mode takes its address from the network. See *Setting Up Your Nexus.io Automation Controller*.

DI / DO (DQ) / AI / AO (AQ). Digital input, digital output, analog input, analog output.

DINT. 32-bit signed integer tag type, addressed `%MD`.

Discovery. The Connect dialog's LAN scan for controllers, using mDNS. Does not cross a VPN; use **Enter address manually**.

Download from target. The offer the IDE makes on connect when the controller holds a project different from, or newer than, the one open: pull the editable source off the controller. Available only if the project was deployed with **Store editable source on target**. See *Connecting, Deploying and Emulating*.

Emulation. Running a compiled project on the PC instead of a controller. See **Build & Emulate**.

Enrolment. The process by which a controller obtains its certificate from an account or site certificate authority and restarts managed. A cloud-connected controller is always enrolled. See *Security, Provisioning and Enrolment*.

EtherNet/IP. Industrial Ethernet protocol used by many PLCs. A Nexus.io project can add an EtherNet/IP PLC as a device and read and write its tags. See *I/O Configuration and Devices*.

Event task. A task that runs once when a chosen memory bit changes on a rising, falling or either edge. See *Function Blocks, Faceplates and Tasks*.

Faceplate. A ready-made HMI widget bound to an industrial function block instance (motor, valve), showing its state and commands. See *Function Blocks, Faceplates and Tasks*.

Freewheel task. A task that runs continuously, as fast as the controller allows; used for the Python `run(ctx)` program.

Function block (FB). A reusable piece of ladder logic with typed input and output pins, called from a network.

Historian. On-controller recording of tags marked **Hist** in the Tag Database, kept for the project's History Retention and read by the trend chart widget and WhiskerHMI Trends. See *Historian and Trends*.

HMI. Human-machine interface: the controller's touchscreen, or a WhiskerHMI PC application.

HMI PIN login. Operator login at the touchscreen: pick your user, enter your Whisker.io PIN. Enabled per project by **Cloud PIN Login**. See *Setting Up Your Nexus.io Automation Controller*.

IDE certificate. The credential a managed controller checks before accepting a connection or a deploy. The Cloud edition fetches yours when you sign in; a site CA issues one in an engineer identity package.

INT. 16-bit signed integer tag type, addressed `%MW`.

IODF. I/O Device File: the description of a Modbus device's registers from which the IDE creates its tags. See *I/O Configuration and Devices*.

Ladder logic. Graphical PLC language: networks of contacts, blocks and a coil between two rails. See *Ladder Logic Editor*.

Latch (alert). An alert that stays active after its condition clears, until acknowledged. See *Alerts Editor*.

Managed. A controller enrolled into a cloud account or provisioned into a site: mutual TLS on the IDE and data ports and signed deploys only. Every cloud-connected controller is managed; without cloud it is the engineer's choice. Shown as a label in the Connect dialog. Compare **Standalone**.

mDNS. Multicast DNS; how controllers announce themselves for discovery.

Modbus. Industrial protocol; Modbus TCP over Ethernet, Modbus RTU over RS-485. The controller is a Modbus master towards its devices and a Modbus TCP server (port 502) towards SCADA. See *Modbus Server and Map Viewer*.

Monitoring. Live values shown in the editors and Tag Monitor. Stays off until this session has built and downloaded the open project to the connected controller. See *Monitoring and Debugging*.

Network (ladder). One line of ladder logic between the rails, with one coil. Added with **Add Network**.

NexusSec. The secure (mutual TLS) option on an ArenaTCP connection to a managed controller, from another controller or from a WhiskerHMI panel.

Panel identity. The certificate a WhiskerHMI panel PC presents to a managed controller, issued to the station name `HMI-<PC name>` by the controller's authority (the cloud account or the site CA). The engineer issues one per panel PC from the Security dialog's **Panels** sub-section as a passphrase-protected `.whisker-panel` package; the operator imports it from the panel's **Panel ▶ Import panel identity...** menu. Revocable individually; not part of the installer; not needed against a standalone controller. See *The WhiskerHMI PC Application*.

Nexus.io Automation Controller. D6 Labs' panel PLC with an integrated 10-inch touchscreen; `Nexus.io AC` in the IDE. See *Target Hardware in the IDE*.

Part number. The product code by which the cloud and the factory identify a controller model; printed in *Target Hardware in the IDE*.

Project. One program for one target, held inside an Application.

PYFB. Python function block: a ladder block that calls a Python function. See *Python on the Controller*.

REAL. 32-bit floating-point tag type, addressed `%MF`.

REPL. Read-eval-print loop: the interactive MicroPython console of a SmartController (Classic), in the **SmartController Classic Console** tab.

Retained variable. A tag whose value survives a program update: values are carried over by tag name, so a setpoint entered at the panel is still there after a re-deploy. See *Tag Database*.

RS-485. Multi-drop serial bus used for Modbus RTU.

Run Mode. Controller mode in which tasks scan and outputs are driven.

Scan. One pass through a task's programs.

SCADA. Supervisory software that reads controllers plant-wide, typically over Modbus TCP.

Site CA / site identity. A certificate authority a customer runs on one PC: the optional way to manage controllers without a cloud account. It issues engineer identity packages; a project's **Site identity** setting selects which imported identity it connects and signs with. See *Security, Provisioning and Enrolment*.

SmartController (Classic). D6 Labs' earlier MicroPython controller, programmed over USB serial; `SmartController Classic` in the IDE.

Standalone. The posture a Nexus.io controller ships in, and stays in unless you enrol or provision it: no identity, plain connections and unsigned deploys from any IDE that can reach it; a WhiskerHMI panel connects plainly. Shown as a label in the Connect dialog. Compare **Managed**.

Stop Mode. Controller mode in which scanning halts.

System tag. A tag the IDE creates automatically: per device *Comm OK*, *Comm Age* and *Error Type*, plus cloud and cellular status on a cloud project. Hidden in Tag Monitor by default.

Tag. A named variable with a type and an address. See *Tag Database*.

Tag Monitor. Bottom-panel tab listing every addressed tag with its live value once monitoring is on.

Target. The kind of hardware a Project runs on: `Nexus.io AC`, `SmartController Classic` or `WhiskerHMI`.

Task. The schedule a ladder program runs on: cyclic, event or freewheel.

Toolbox. The panel listing what can be added to the active editor.

Undefined-tag sweep. The check, at Save and before Build, Emulate and Deploy, that every tag name the logic refers to exists in the Tag Database; it offers to create the missing ones. See *Tag Database*.

WhiskerHMI. The Windows operator application the IDE generates; `WhiskerHMI` as a target. See *The WhiskerHMI PC Application*.

Whisker.io. D6 Labs' cloud service: accounts and users, device provisioning and certificates, published projects, process data and audit. Used by the Cloud edition.

.widez file. An Application archive. See *Projects and Applications*.

Appendix E — Getting Help

28.32 Documentation

This manual covers both editions of the Whisker IDE, Cloud and Offline. It ships as a PDF in the download package alongside the installer, and the Quick Start — the first chapter plus the four walkthroughs — is also available as a separate, shorter PDF. Ask your D6 Labs contact for the current version.

Other D6 Labs documents you may need:

- **Nexus.io Automation Controller hardware guide** — mounting, wiring, power and enclosure dimensions. Ships with the controller.
- **WhiskerHMI runtime guide** — for integrators who install the PC application at customer sites. Complements *Generating HMI Installers* and *The WhiskerHMI PC Application*.

28.33 Support

support@d6labs.com

When you write, include:

- Your IDE version and edition (Cloud or Offline). The edition is in the window title — *Whisker PLC IDE* or *Whisker PLC IDE Standalone* — and the version in Windows **Settings** → **Apps**.
- The Application archive (`.widez`) you were working with, if the problem is project-specific.
- What you did, what you expected and what happened.
- A screenshot if the problem is visible in the IDE.
- The Output panel's contents: click **Copy All** in its header and paste into the e-mail. The IDE does not write a log file.
- For a controller problem: the controller's serial number (shown in the Cloud Explorer, or on the unit's label) and whether the Connect dialog lists it as *Managed*, *Standalone* or *Awaiting enrolment*.

1-844-365-8647

Phone support is available during US business hours. Use it for urgent production issues; e-mail is faster for everything else.

28.34 Reporting bugs

Report IDE crashes, incorrect behaviour and display glitches to the same address, with:

- IDE version and edition
- Steps to reproduce
- Expected and actual behaviour
- The Output panel contents (Copy All)
- If the IDE crashed: the entry from Windows **Event Viewer** → **Windows Logs** → **Application** for `whisker_ide`

28.35 Feature requests

Send them to the support address. Every request is read; popular ones are rolled into future releases.

28.36 Release notes

`CHANGELOG.txt` in the download package lists the notable changes in each release.