

Whisker IDE Quick Start Guide

Standalone Edition

D6 Labs

May 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

1.6 Core concepts

1.6.1 I/O acronyms used throughout this guide

1.6.2 Arena memory

1.7 A tour of the IDE workspace

1.8 What's next

2 The Walkthrough scenario

2.1 What you're building

2.2 The hardware

2.2.1 Wiring map

2.2.2 RS-485 settings

2.3 Tags — where they come from

2.4 The control logic — what it does

2.5 The alarm

2.6 The HMI screen

2.7 Watching the live data — Tag Monitor

2.8 What's next

3 Your First PLC Project — Ladder

3.1 Before you start

3.2 Step 1 — Create the Application and Project

3.3 Step 2 — Add the mSmart Universal IO module

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

3.5 Step 4 — Add the memory tags

3.6 Step 5 — Write the ladder logic

3.6.1 Network 1 — Decode the operator mode (CMP)

3.6.2 Network 2 — Latch the alarm

3.6.3 Network 3 — Clear the alarm latch

[3.6.4 Network 4 — Auto-release the Clear Alarms button \(one-shot\)](#)

[3.6.5 Network 5 — call_motor SET \(start float closes the tank's request\)](#)

[3.6.6 Network 6 — call_motor RESET \(stop float clears the request\)](#)

[3.6.7 Network 7 — Drive `motor_run`](#)

[3.7 Step 6 — Define the HighLevelAlarm alert](#)

[3.8 Step 7 — Build the HMI screen](#)

[3.9 Step 8 — Connect and Build](#)

[3.10 Step 9 — Test it](#)

[3.11 What's next](#)

[4 Your First PLC Project — Python](#)

[4.1 Before you start](#)

[4.2 Step 1 — Create the Application and Project](#)

[4.3 Step 2 — Add the mSmart Universal IO module](#)

[4.4 Step 3 — Rename the I/O tags](#)

[4.5 Step 4 — Add the memory tags](#)

[4.6 Step 5 — Write the Python application](#)

[4.7 Step 6 — Define the HighLevelAlarm alert](#)

[4.8 Step 7 — Build the HMI screen](#)

[4.9 Step 8 — Connect and Build](#)

[4.10 Step 9 — Test it](#)

[4.11 What Python gives you that ladder doesn't](#)

[4.12 What's next](#)

[5 Your First WhiskerHMI Project](#)

[5.1 Before you start](#)

[5.2 Step 1 — Reopen the Walkthrough-Ladder Application](#)

[5.3 Step 2 — Add a WhiskerHMI project to the Application](#)

[5.4 Step 3 — Add an ArenaTCP device pointing at the controller](#)

[5.5 Step 4 — Import tags from the Motor project](#)

[5.6 Step 5 — Copy the HMI screen from Motor](#)

[5.7 Step 6 — Build the Panel runtime](#)

[5.8 Step 7 — Generate the Windows installer](#)

[5.9 Step 8 — Install and run on a PC](#)

[5.10 Step 9 — Test it](#)

[5.11 What's next](#)

6 Where to go next

6.1 The full user manual

6.2 Exposing tags to Modbus

6.3 The Design Spec

6.4 Connecting to a device

6.5 Sample projects

6.6 When you get stuck

6.7 Support

6.8 What we'd love to hear about

1 Getting Started

Welcome to the Whisker IDE — the design environment for the D6 Labs Nexus.io family of industrial automation controllers. This chapter walks you through downloading and installing the IDE, gives you a tour of the workspace, and ends with a five-minute walkthrough that gets you to a saved project ready to deploy.

1.1 About this manual

This manual covers the **Standalone edition** of the Whisker IDE on Windows. The Standalone edition is fully offline: it does not require an account, an internet connection, or any cloud subscription. Everything you do — designing programs, configuring I/O, deploying to a controller — happens between your PC and the controller on your local network.

A separate manual covers the Cloud edition of the Whisker IDE for users who want fleet management, cloud-staged deployments, and role-based access control through Whisker.io.

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	200 MB free	1 GB free (for projects and HMI runtime bundles)
Display	1280 × 720	1920 × 1080 or larger
Network	Ethernet or Wi-Fi on the same LAN as your controller	Same

The Whisker IDE itself does not need an internet connection. You only need network connectivity to reach your Nexus.io controllers during deployment.

1.3 Downloading the IDE

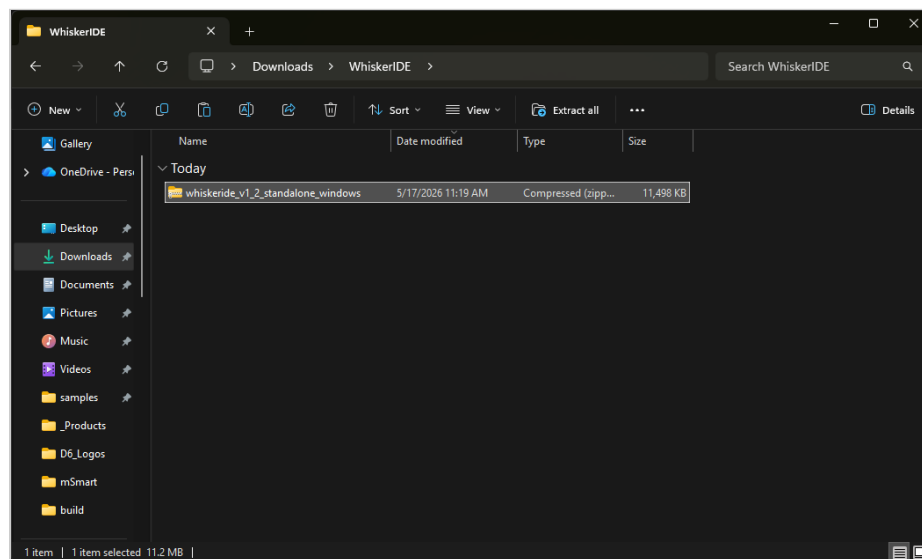
1. Open your web browser and go to the Whisker IDE download page:

```
www.d6labs.com/downloads
```

Pick the **Standalone (Offline) edition for Windows** download. You'll get a single zip file containing the installer plus four short text files (README, LICENSE, CHANGELOG, THIRD-PARTY-NOTICES) and this Quick Start PDF.

Note on versioning. The download page always serves the current release of the v1.2.x series. The installer inside the zip carries the full three-component version (e.g., 1.2.0).

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



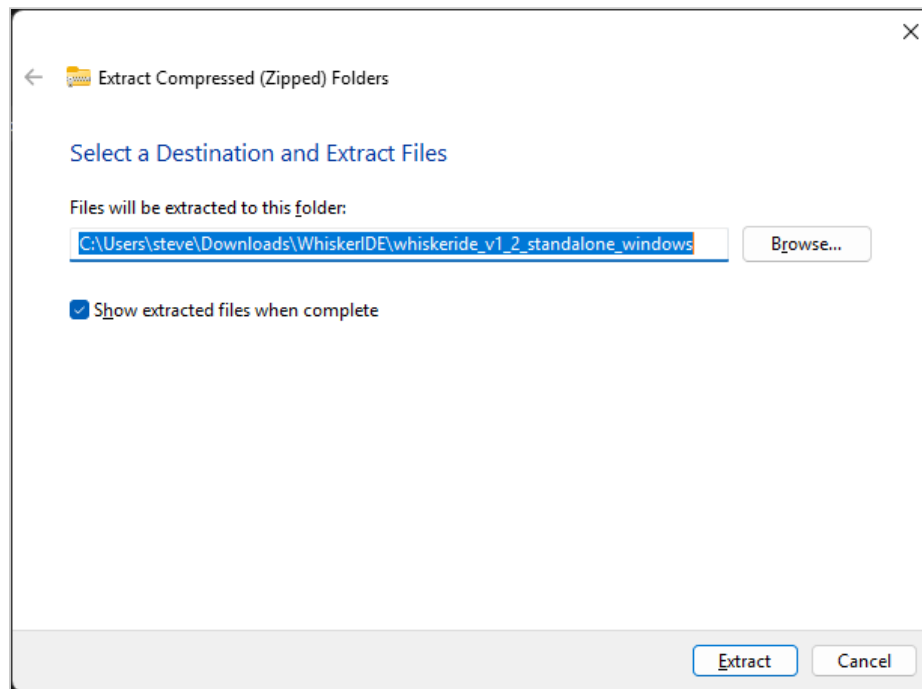
Note. If your browser warns “this file might be dangerous,” choose **Keep** or **Keep anyway**. Until D6 Labs’ installer is signed by a commercial code-signing certificate, Windows SmartScreen may flag any new download from us. This is normal and the file is safe.

1.4 Installing the IDE

The download is a zip file, not the installer itself. You need to extract it first.

1.4.1 Step 1 — Extract the zip

1. Right-click the downloaded `whiskeride_v1_2_standalone_windows.zip` file.
2. Choose **Extract All...**
3. Pick a destination (the default — a folder with the same name in Downloads — is fine).
4. Click **Extract**.



When extraction finishes, you'll have a folder containing:

File	What it is
<code>WhiskerIDE-Offline-1.2.0-Setup.exe</code>	The installer you'll run next
<code>README.txt</code>	Quick reference matching this chapter
<code>LICENSE.txt</code>	End-user license agreement
<code>CHANGELOG.txt</code>	Notable changes in this release
<code>THIRD-PARTY-NOTICES.txt</code>	Open-source component attributions
<code>WhiskerIDE-QuickStart-v1.2.0.pdf</code>	This Quick Start Guide

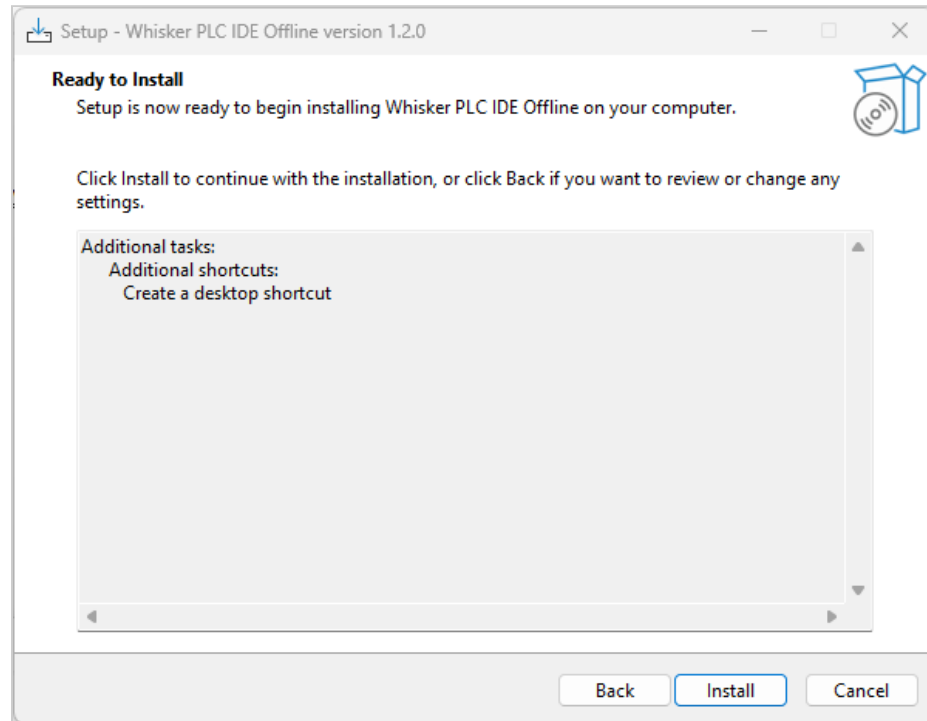
1.4.2 Step 2 — Run the installer

1. Double-click `WhiskerIDE-Offline-1.2.0-Setup.exe`.

Note. Windows may first show a “Windows protected your PC” or “The publisher of this file could not be verified” warning because the installer was downloaded from the internet and is not yet signed by a commercial code-signing certificate. Click **More info** → **Run anyway** (or **Run** on older Windows). This warning is separate from — and appears *before* — the User Account Control prompt in the next step.

2. Windows shows a **User Account Control** prompt asking permission to make changes. Click **Yes**.
3. The setup wizard appears. The default values are correct for almost everyone — you can click **Next** through each screen.

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



4. Installation takes 10–20 seconds. When it finishes, the IDE launches automatically.

1.4.3 Step 3 — Confirm the install

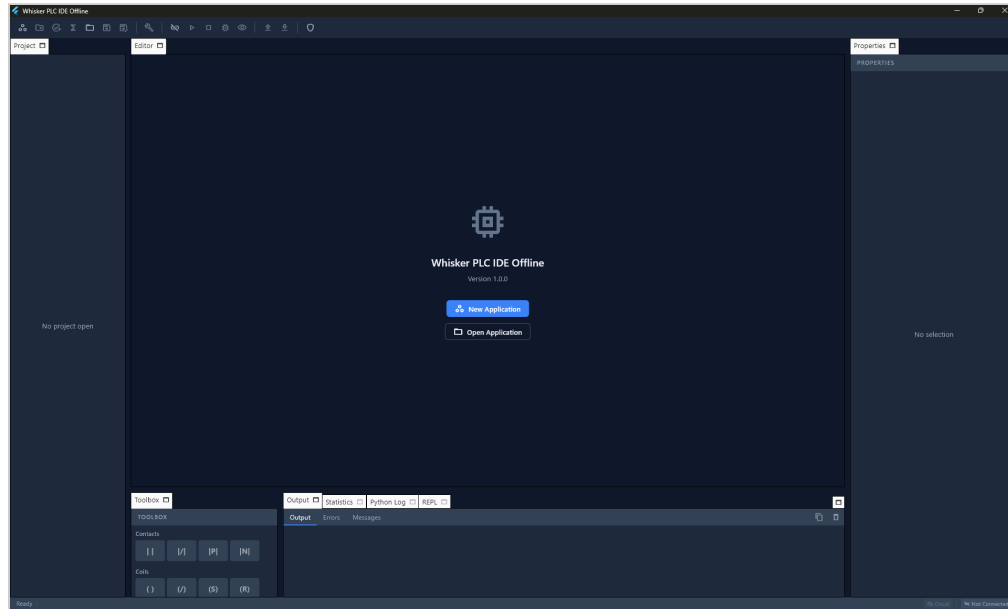
After the installer finishes, you should see:

- A **Whisker PLC IDE Offline** entry in your Start menu.
- A desktop shortcut (if you chose that option).
- A new program registered under **Settings** → **Apps & features**.

You can safely delete the extracted folder and the original zip — the IDE is now installed in `C:\Program Files\Whisker PLC IDE Offline\`.

1.5 First launch

When the IDE opens for the first time, you see an empty workspace. No login, no account setup, no first-run wizard — the Standalone edition is ready to use immediately.



Take a moment to find the window title in the title bar: it should read **Whisker PLC IDE Offline**. That's how you tell at a glance which edition you're running.

1.6 Core concepts

Before you build anything, it helps to understand the core concepts the IDE revolves around.

Application — A container that groups one or more related Projects. For a single-controller installation, your Application will contain one Project. For a system with several controllers and an operator panel, your Application might contain three or four Projects that share tags and design specs.

Project — A single program for a single controller. A Project knows which kind of controller it targets (its Target). Projects are saved as `.widez` files (a single-file zip archive containing ladder logic, tags, I/O configuration, HMI screens, and alerts).

Target — The type of controller the Project runs on. The Standalone edition supports three Targets:

- **Nexus.io AC** — D6 Labs' flagship Automation Controller. A full PLC in a panel-mount enclosure with an **integrated touchscreen** on the front. There is no separate "HMI box" sold alongside it — the touchscreen is part of the controller hardware, and the HMI screens you build in the IDE's HMI Designer render directly onto that touchscreen. So when this manual says "the controller" and "the touchscreen" in the same sentence, it's referring to two faces of the same physical device.

- **SmartControllerClassic** — D6 Labs' earlier MicroPython-based controller. Headless (no display). Uses the Python editor instead of ladder logic.
- **WhiskerHMI** — A **Windows desktop application**, not a piece of hardware. The WhiskerHMI runtime is a Flutter app that the IDE packages into a `.exe` installer; you run that installer on whatever Windows PC you want to use as an operator panel — a laptop, a desktop, or a hardened industrial panel-PC. It contains no ladder logic and runs no I/O of its own; it reads tag values from a Nexus.io AC over the network and renders the same HMI screens the controller's built-in touchscreen renders. Each WhiskerHMI installation is one additional view of an existing Nexus.io AC; you can deploy as many of them as you have PCs.

What about “the touchscreen” and “the HMI”? Both terms appear in this manual and they're not always the same thing:

- In a Nexus.io AC Project, “the touchscreen” and “the HMI” refer to the controller's built-in display.
- In a WhiskerHMI Project, “the HMI” is the Windows app and “the touchscreen” (if any) is the touch-capable display of the PC the app runs on — that's a property of the PC, not the IDE.
- The HMI Designer in the IDE is the same editor in both cases; only the deployment target changes.

Tag — A named variable in your program. Tags have a data type (BOOL, INT, DINT, REAL) and an address in controller memory (`%M0` , `%MW10` , `%MD4` , `%MF12`). You reference tags from ladder logic, HMI bindings, alerts, and the Modbus map.

Task — A scheduling unit on the controller that runs your logic at a defined cadence. A Project can contain three kinds of task:

- **Periodic task** — runs once every N milliseconds (e.g., 10 ms scan). The `Main` task that every new Project starts with is a periodic task. Used for normal control logic.
- **Event-driven task** — runs once when a triggering condition becomes true (a tag goes high, an alarm latches, a digital input edge fires). Used for one-shot reactions that don't need to be polled every scan.
- **Freewheel task** — runs continuously as fast as the CPU allows, yielding only between iterations. Used for the Python `run(ctx)` function and for anything that does its own internal pacing with `ctx.wait()` .

You'll see these in the project tree under **Tasks** in every PLC project.

1.6.1 I/O acronyms used throughout this guide

When discussing physical I/O points, the manual uses standard industry acronyms:

Acronym	Meaning	Example
DI	Digital Input — a binary on/off input signal	float switch, push button, limit switch
DO (or DQ)	Digital Output — a binary on/off output signal	contactor coil, relay, indicator lamp
AI	Analog Input — a continuously variable input signal	4-20 mA pressure transmitter, 0-10 V level sensor
AO	Analog Output — a continuously variable output signal	4-20 mA valve position command, 0-10 V VFD speed reference

Tag addresses follow the IEC-61131 convention: `%I` for digital inputs, `%Q` for digital outputs, `%IW` / `%QW` for analog channels. The IDE assigns these automatically when you add an I/O module — you never type them in by hand.

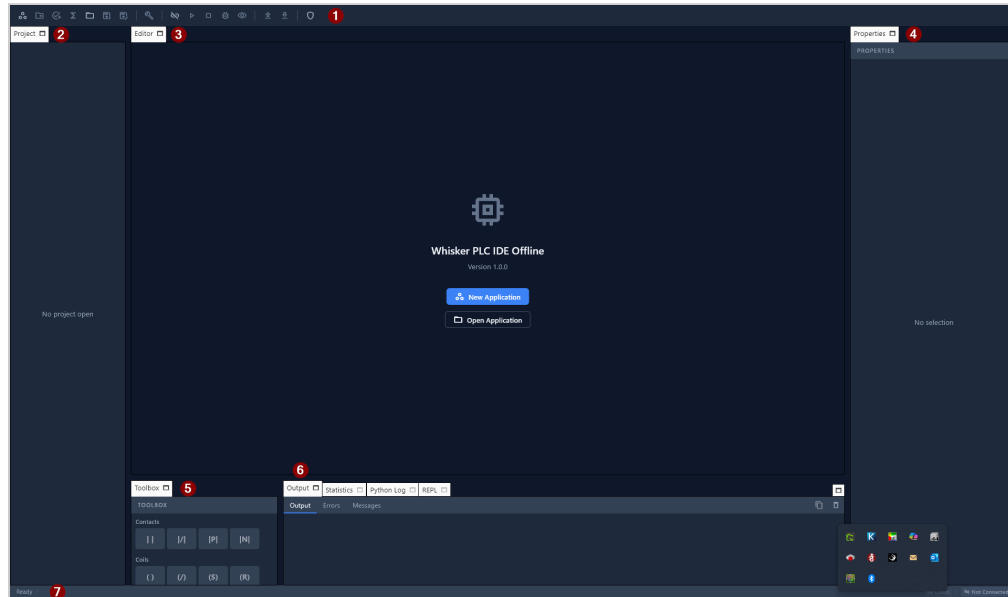
1.6.2 Arena memory

You'll encounter the term **arena memory** throughout the manual. The “arena” is the controller's shared, in-memory tag store — a block of RAM that holds the current value of every tag, every scan. The PLC runtime writes new values into the arena on each scan, and the HMI, the Modbus server, the Python runtime, and the cloud agent all read from (and write to) the same arena. That single shared store is what lets every component of the controller see consistent tag values without having to query each other.

For day-to-day work you don't have to think about it — “the arena” is just where tag values live. It becomes relevant when you read about Tag Monitor, ArenaTCP, or `ctx.read_tag()` later in the manual.

1.7 A tour of the IDE workspace

The Whisker IDE workspace is divided into seven main areas. The arrangement is shown below, and each area is described in detail in the chapter *The Workspace*.



1. **Toolbar** — file operations, build, connect, run, deploy. The set of visible buttons depends on the active project's Target type.
2. **Project Tree** — the structure of the open Application and all its Projects.
3. **Editor area** — open editors for tags, ladder programs, HMI screens, etc. Multiple editors stack as tabs.
4. **Properties panel** — when you click an element in the project tree or the active editor, its editable properties appear here.
5. **Toolbox** — context-sensitive palette for the active editor (ladder elements, HMI widgets, etc.).
6. **Output panel** — build results, deploy progress, errors and warnings.
7. **Status bar** — connection state, current target, dirty/clean indicator.

1.8 What's next

You now know enough about the IDE to build something real. The next three chapters walk you through three end-to-end projects against the same physical scenario — a single motor controlled by float switches, with an Auto/Hand/Off operator panel and a latched high-level alarm. Each walkthrough uses the same controller, the same I/O module, and the same HMI screen, so you can see how Ladder, Python, and a standalone WhiskerHMI panel each express the same control intent:

- **Chapter 2 — The Walkthrough scenario** sets up the shared hardware, I/O wiring, and tag conventions that the three walkthroughs reuse.
- **Chapter 3 — Your First PLC Project (Ladder)** builds the motor control logic as ladder rungs on a Nexus.io AC.

- **Chapter 4 — Your First PLC Project (Python)** builds the same logic as Python code on the same controller.
- **Chapter 5 — Your First WhiskerHMI Project** builds a standalone Windows HMI panel that connects to the Nexus.io AC over Modbus TCP.

If you already have a controller on your LAN, start with Chapter 2. If you're working without hardware for now, the Ladder and Python walkthroughs (chapters 3 and 4) cover the design steps you can do offline; the Build step requires a connected controller because Build compiles **and** uploads in a single operation.

2 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 standalone HMI panel — 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.

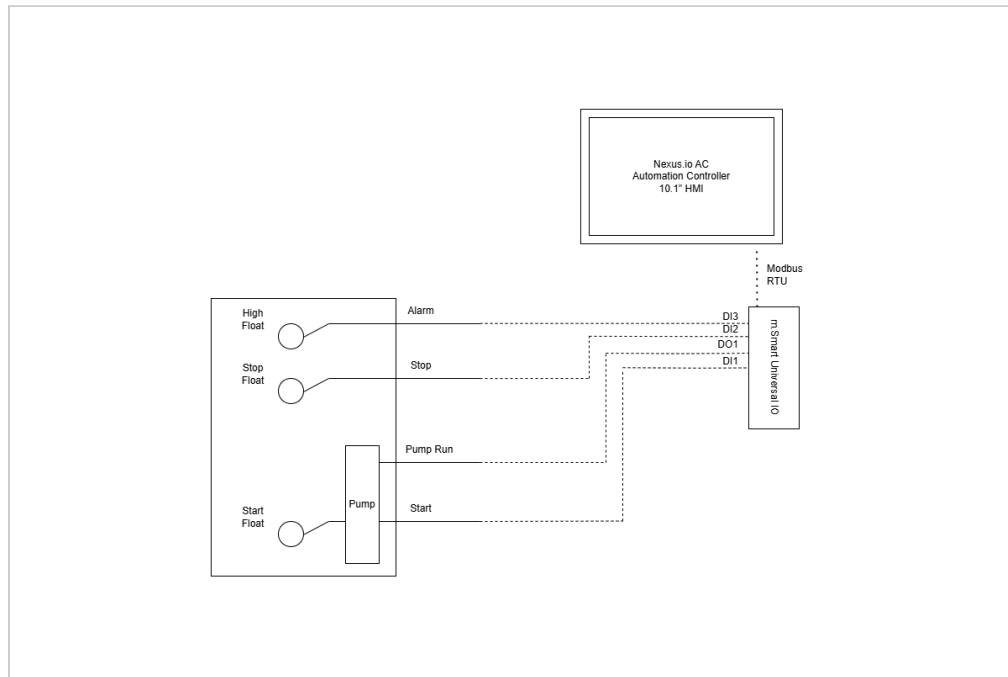
This chapter sets up everything the walkthroughs assume — the hardware, the field wiring, the tag names, the control intent, and the HMI layout. Read it once. Each walkthrough then refers back here for setup details rather than repeating them.

2.1 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 AC

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 AC 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.

2.2 The hardware

You need three things:

- **A Nexus.io AC** on your LAN, powered on, reachable from your PC by IP or mDNS hostname.
- **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.

2.2.1 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.

2.2.2 RS-485 settings

The mSmart module talks Modbus RTU at the following defaults. The IDE will let you change all of these from the Add Device dialog if your hardware is configured differently.

Setting	Value
Baud rate	19200
Data bits	8
Parity	None
Stop bits	1
Slave ID	1
Controller serial port	rs485 (UART 1)

2.3 Tags — where they come from

The walkthroughs use seven tags. 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 three 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
<code>aho_mode</code>	INT	<code>%MW0</code>	added in Tag Database	Operator mode: 0=Off, 1=Hand, 2=Auto

Name	Type	Address	Origin	What it is
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)
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 Appendix AB of the main 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.

2.4 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 equals.** The Whisker IDE doesn't have a standalone EQ instruction — it has a single CMP block that emits LT / EQ / GT outputs from one comparison. That fits this problem perfectly: one CMP rung 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.

2.5 The alarm

A single alert definition in each project:

Field	Value
Name	HighLevelAlarm
Condition	<code>float_alarm = TRUE</code>
Latched	Yes
Output tag	<code>alarm_latched</code>
Severity	Warning
Message	High-level float is wet

When `float_alarm` goes TRUE the alert system sets `alarm_latched` and records an entry in the alert history. The latch stays set until `clear_alarms` clears it through the control logic above.

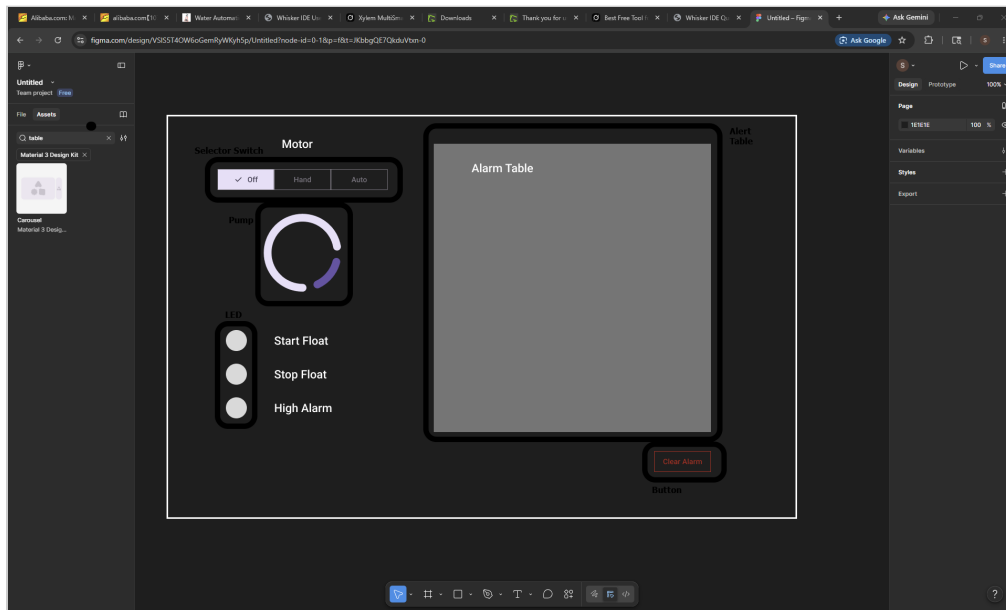
2.6 The HMI screen

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.



2.7 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.

2.8 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.

3 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 AC. 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.

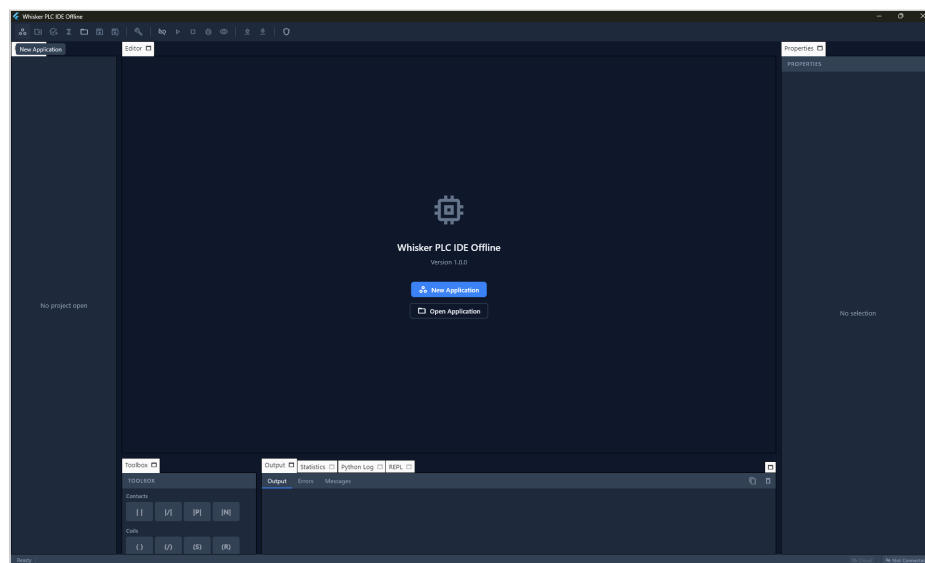
3.1 Before you start

You need:

- The Whisker IDE installed (Chapter 1).
- A Nexus.io AC powered on, on the same LAN as your PC, with the mSmart Universal IO module wired per [the wiring map](#).
- The Walkthrough scenario chapter read (or at least skimmed) — this walkthrough doesn't re-explain the tags, wiring, or control intent.

3.2 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:

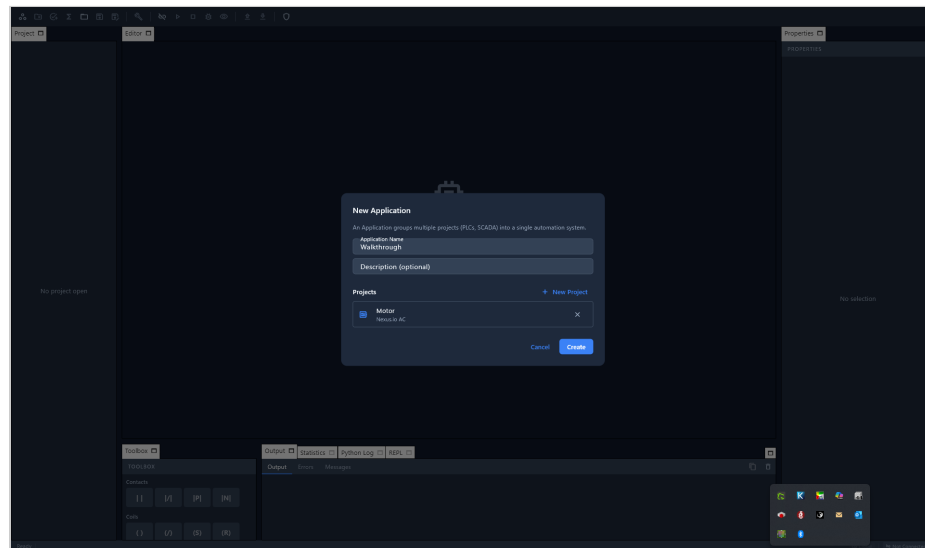
- **Application name:** Walkthrough-Ladder
- **Description:** leave blank (or type something brief like “Motor control demo”)

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.

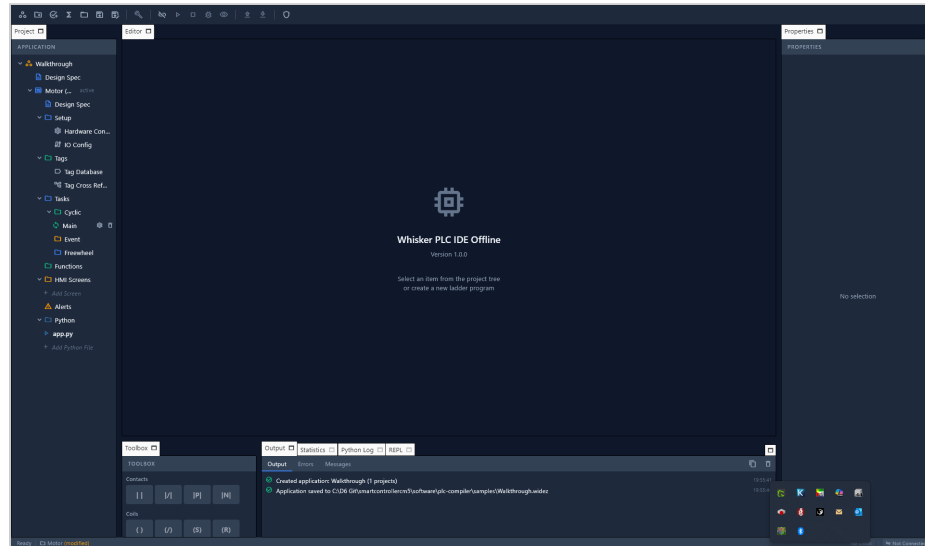
5. Back in the outer Application dialog, click **Create**.



6. 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:

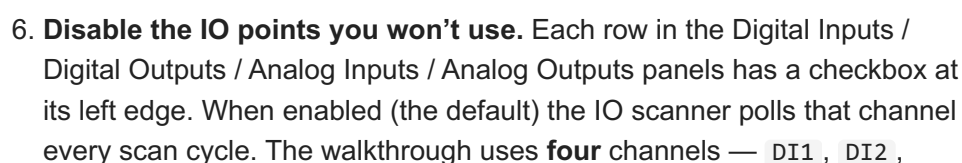
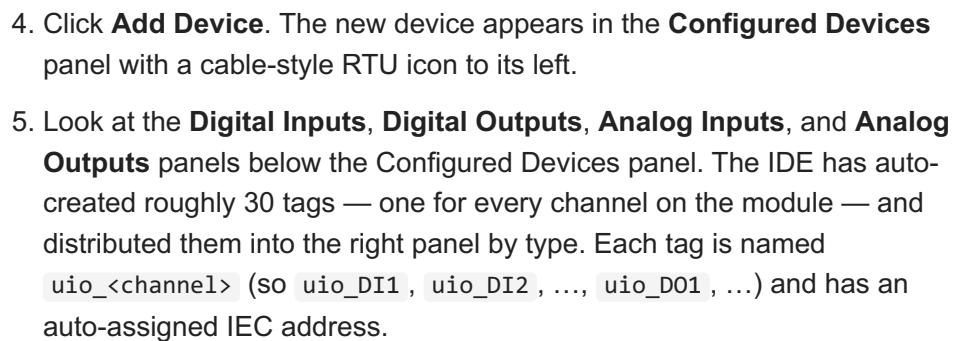
- A new Application named **Walkthrough-Ladder** in the title bar.
- A single project named **Motor** in the Project Tree on the left.
- An empty editor area in the middle.



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.

3.3 Step 2 — Add the mSmart Universal IO module

1. In the Project Tree, click **Motor** → **IO Config**. The IO Configuration editor opens.
2. Click **Add Device** in the toolbar. The Add Device dialog opens.
3. Configure the device:
 - **IODF**: 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 IODF:
 - **Port**: `rs485` (the only RS-485 port on the Nexus.io AC)
 - **Slave ID**: `1`



DI3 , and DO1 — so uncheck everything else (DI4, DI5, DI6, all *Total Count and *Events Per Min entries in Digital Inputs; AI1-5 and the *Filter Window entries in Analog Inputs/Outputs; Solar Voltage). 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.

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

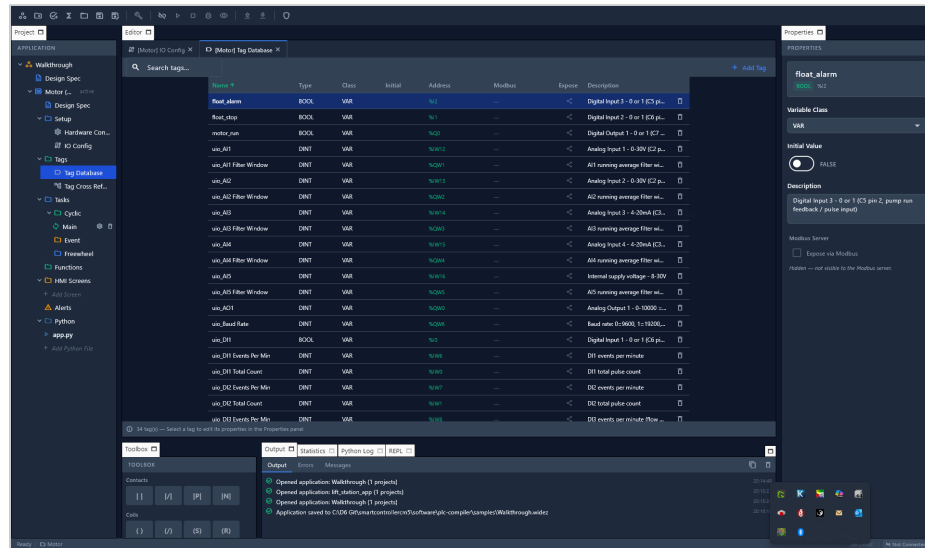
The 30 auto-created tags include analog readings, pulse counters, event rates, and config registers. The walkthrough only uses four of them. Renaming those four to friendlier names makes the ladder and HMI easier to read.

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

From	To	Why
uio_DI3	float_alarm	High-alarm float input
uio_DO1	motor_run	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**.

3.5 Step 4 — Add the memory tags

Seven 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.

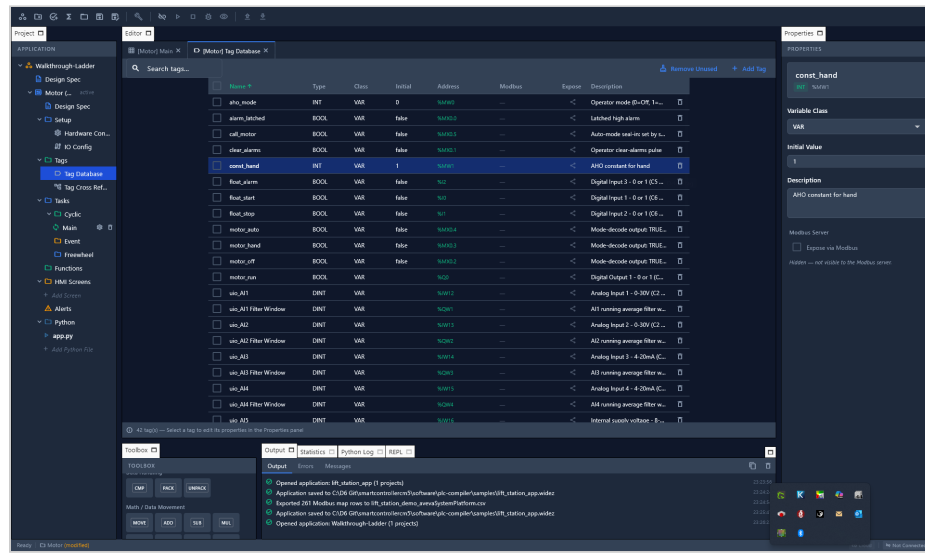
1. Still in the Tag Database editor, click **Add Tag**.
2. First tag — the operator mode:
 - **Name:** `aho_mode`
 - **Type:** `INT`
 - **Class:** `VAR`
 - **Memory Area:** `Memory Word (16-bit)` → address auto-fills `%MW0`
 - **Description:** `Operator mode (0=Off, 1=Hand, 2=Auto)`

Click **Add**.

3. Repeat for the remaining seven:

Name	Type	Memory Area	Address (auto)	Initial value	Description
const_hand	INT	Memory Word (16-bit)	%MW1	1	Constant used as IN2 of the CMP block — the AHO “Hand” value
motor_off	BOOL	Memory Bit	%MX0		Mode-decode output: TRUE when aho_mode = 0
motor_hand	BOOL	Memory Bit	%MX1		Mode-decode output: TRUE when aho_mode = 1
motor_auto	BOOL	Memory Bit	%MX2		Mode-decode output: TRUE when aho_mode = 2
call_motor	BOOL	Memory Bit	%MX3		Auto-mode seal-in: set by start float, reset by stop float
alarm_latched	BOOL	Memory Bit	%MX4		Latched high-alarm
clear_alarms	BOOL	Memory Bit	%MX5		Operator clear-alarms pulse

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 an initial value of `1`, and use it as the comparison operand. The tag never changes at runtime; it's just how you spell “the integer 1” to the CMP block.



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.

3.6 Step 5 — Write the ladder logic

In the Project Tree, expand **Motor** → **Tasks** and double-click **Main**. (Main is a task, not a folder, so it doesn't expand — a double-click on the task opens its ladder program directly.) 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

3.6.1 Network 1 — Decode the operator mode (CMP)

The IDE doesn’t ship a standalone EQ instruction. It 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.

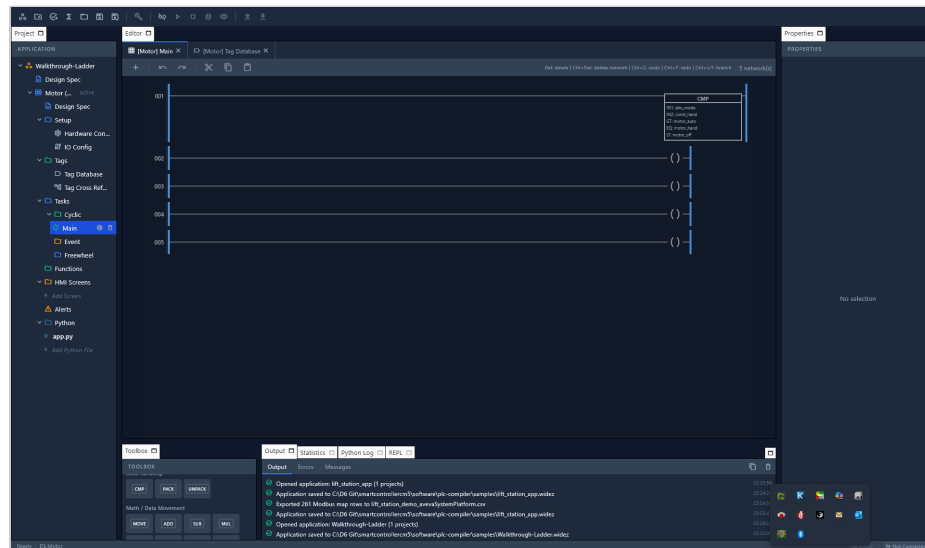
CMP is a **sink instruction** (like a coil). It has no right-hand output pin; instead its three outputs are set as properties in the property dialog and written to tag values directly. You drop one CMP in the terminal slot of a network and the network is complete.

1. Click into network 1 to select it.

2. From the **Toolbox**, drag a **CMP** block (under Compare) onto the terminal slot at the right end of the network. A dialog will appear; set the following properties:

- **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)
- **LT output:** `motor_off` (Rationale: `aho_mode < 1` means `aho_mode = 0`, which is Off.)
- **EQ output:** `motor_hand` (Rationale: `aho_mode = 1` is Hand.)
- **GT output:** `motor_auto` (Rationale: `aho_mode > 1` means `aho_mode = 2`, which is Auto.)

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.

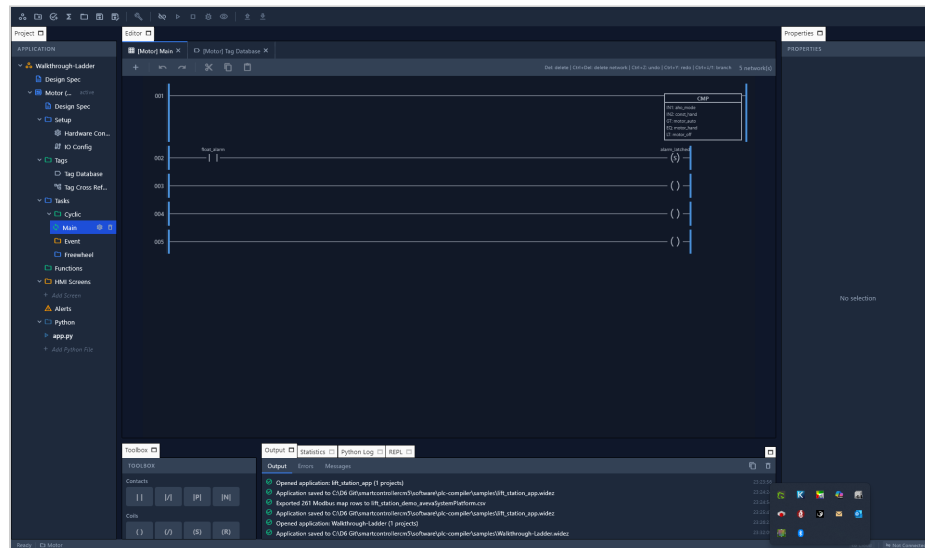


What's nice about this pattern. Three BOOL flags fall out of one CMP network. The rest of the program reads them like any other contact; nobody downstream needs to know `aho_mode` is an integer or what its values mean. That separation makes it easy to add a fourth mode later (e.g. a Maintenance mode 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.

3.6.2 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.

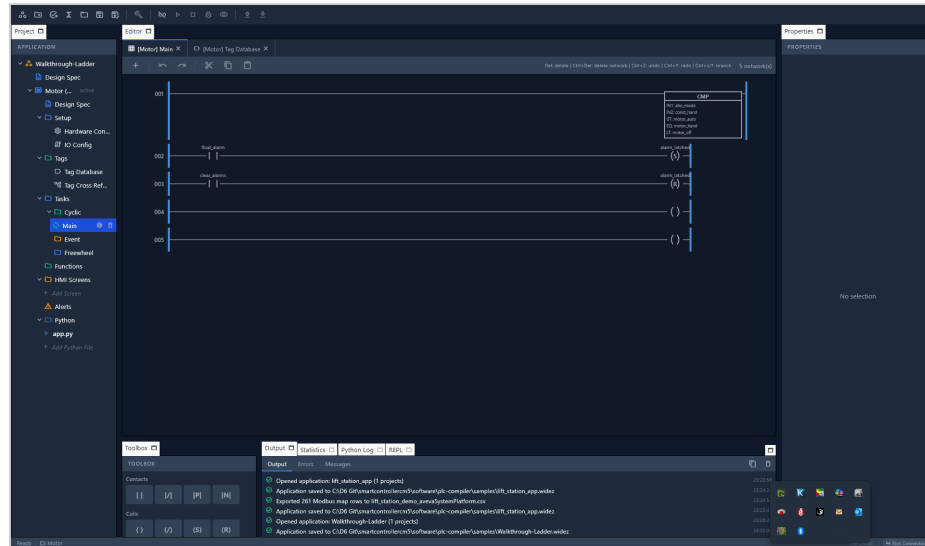
1. Click into network 2.
2. Drag a **Normally Open Contact** onto the first segment. Tag: `float_alarm`.
3. Drag a **Coil** to the terminal slot. Set:
 - **Tag:** `alarm_latched`
 - **Mode:** Set/Latch



3.6.3 Network 3 — Clear the alarm latch

When the operator presses Clear Alarms, reset `alarm_latched`.

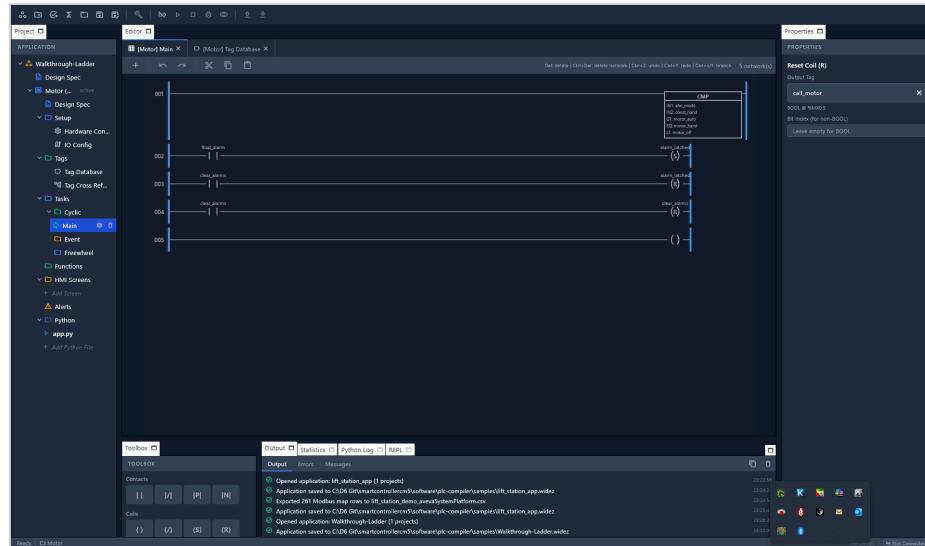
1. Click into network 3.
2. Drag a **Normally Open Contact** onto the first segment. Tag: `clear_alarms`.
3. Drag a **Reset Coil (R)** to the terminal slot. Tag: `alarm_latched`.



3.6.4 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`.

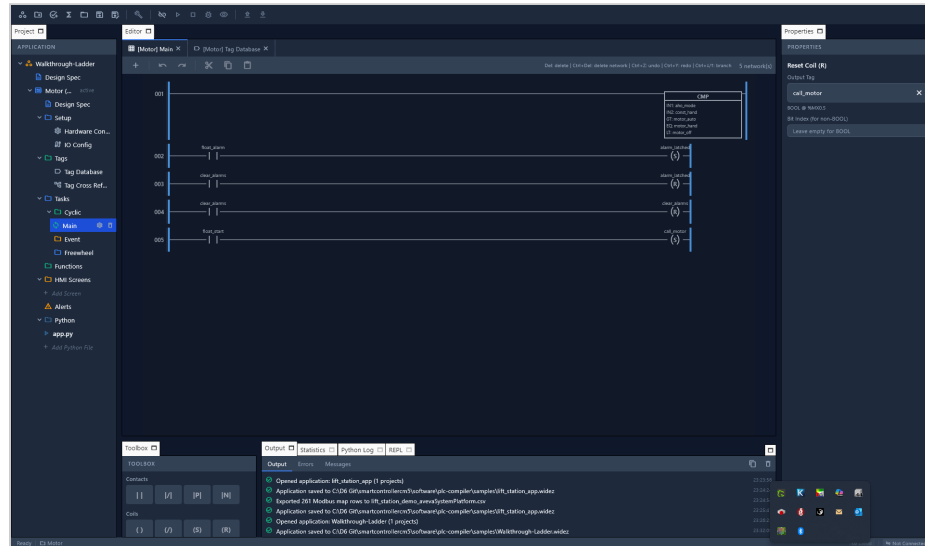


Why two networks instead of one? Networks 3 and 4 fire from the same condition (`clear_alarms = TRUE`) but drive different coils, and the IDE allows only one coil per network. So we spell the “reset both” intent as two networks with identical contact patterns. They execute in the same scan, so the behavior is equivalent to “reset both at once”.

3.6.5 Network 5 — `call_motor` SET (start float closes the tank’s request)

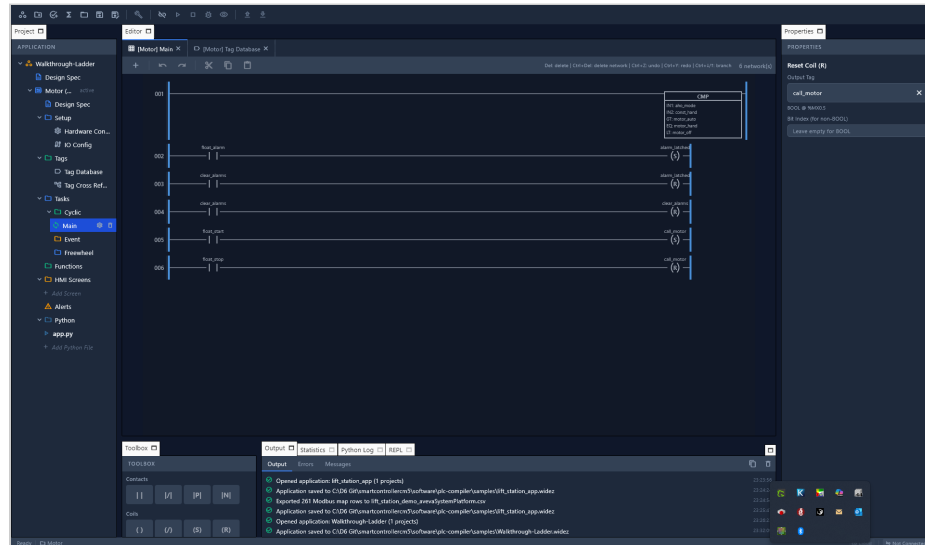
Pulling the Auto-mode seal-in out into its own pair of networks keeps the motor output readable. `call_motor` represents “the tank is asking for the motor to run” — it goes TRUE when the start float closes and stays TRUE (latched) until the stop float clears it in the next network.

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`.



3.6.6 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`.



3.6.7 Network 7 — Drive motor_run

The payoff network. With all the state already decoded above, this one reads like a one-line sentence:

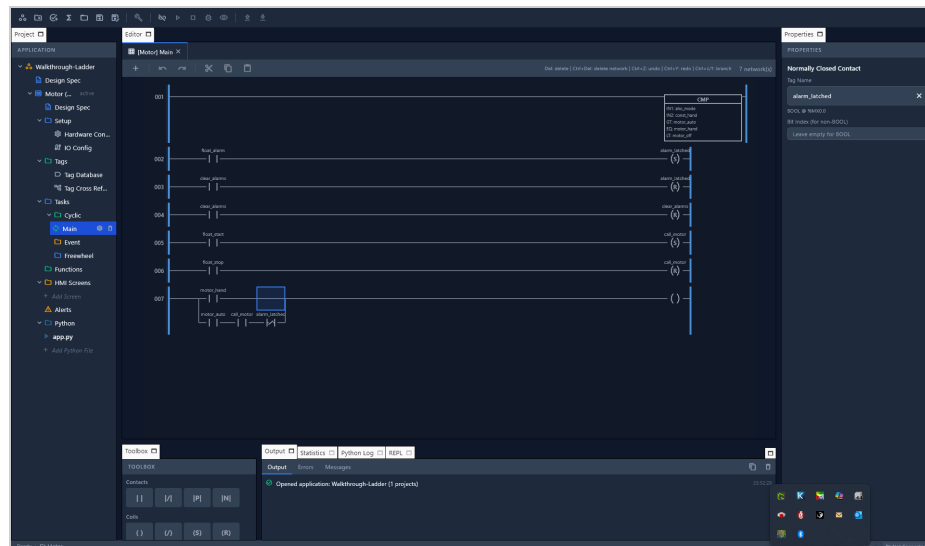
“Motor runs in Hand, or in Auto when the tank is calling and the alarm isn’t latched.”

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, then click into it.
2. Drag a **Normally Open Contact** onto the first segment. Tag: `motor_hand` .
3. Drag a **Coil** (mode `Simple`) to the terminal slot. 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.)



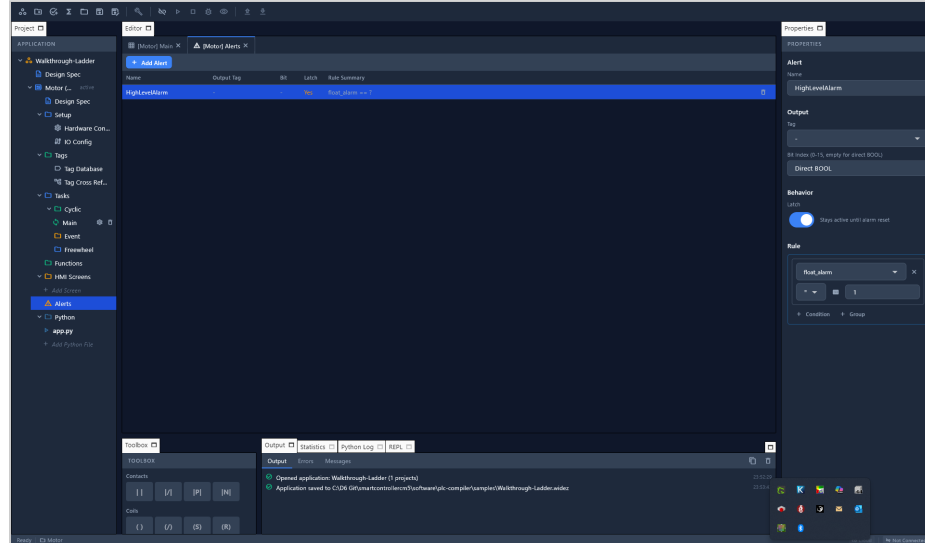
6. Press **Ctrl+S**.

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.

3.7 Step 6 — Define the HighLevelAlarm alert

1. In the Project Tree, click **Motor** → **Alerts**. The Alerts editor opens.
2. Click **Add Alert**. Fill it in:
 - **Name:** `HighLevelAlarm`
 - **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 `FIRED` each time `float_alarm` goes true and a `CLEARED` 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.

3.8 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:**

- Bind `position` → `aho_mode`
- Positions: 3
- Labels: `OFF`, `HAND`, `AUTO`

- **pump** (motor run):

- Bind `state` → `motor_run`
- Label: `Motor running`
- **Active color:** green (shown while the pump is running).
- **Fault color:** red (shown when the bound *fault* tag is true).
This walkthrough doesn't have a motor fault tag, so leave the fault binding blank — the fault color never shows. It's still worth setting because in a real project a motor overload bit would bind here 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):

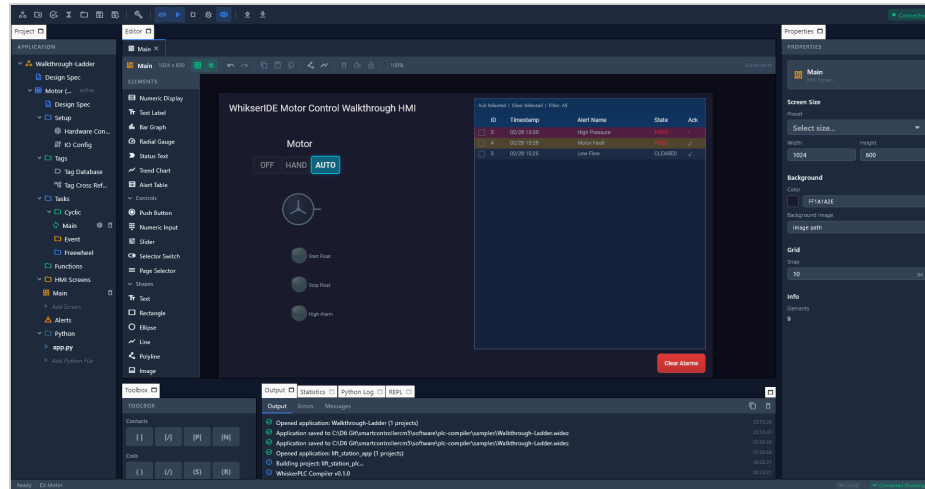
- Label: `Motor`

- **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.



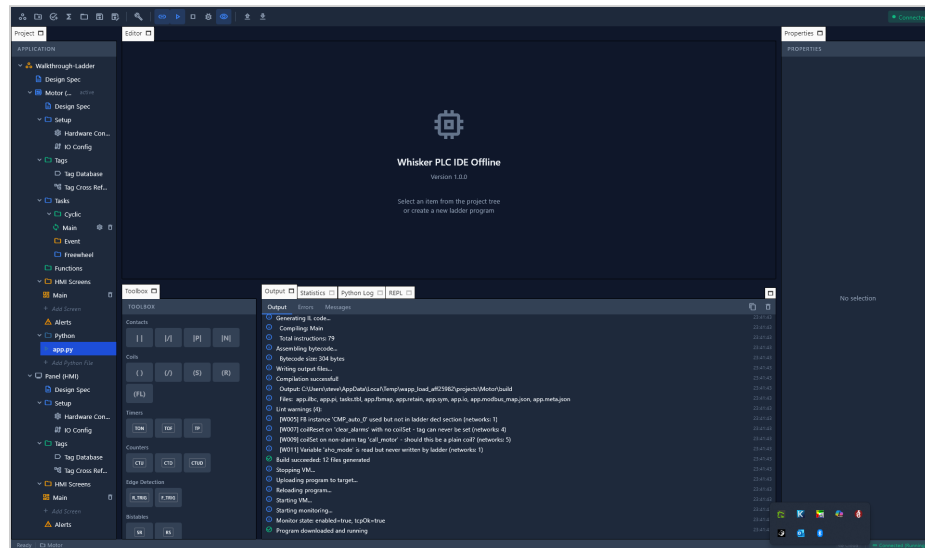
4. Press **Ctrl+S**.

3.9 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.
2. When your controller appears in the list, click it and click **Connect**. If mDNS is blocked on your network, click **Enter address manually** and type the controller's IP address.
3. A confirmation notification appears at the bottom of the IDE. The status bar now shows the connection.
4. Press **F5** (or click the **Build** icon). The Output panel at the bottom shows the compile, then the upload to the controller:

```
[INFO] Building project: Motor...
[INFO] Compiling ladder programs (1)...
[INFO] Compiling alerts (1)...
[INFO] Generating app.io for I/O scanner...
[OK] Build succeeded: 5 files generated
[INFO] Uploading to 192.168.1.244...
[OK] Upload complete (32 KB)
```

5. The controller restarts the runtime services automatically and begins executing the new program.



3.10 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.

3.11 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 (Chapter 15 of the main 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.

4 Your First PLC Project — Python

This walkthrough builds the same scenario as the Ladder walkthrough, on the same Nexus.io AC hardware, 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.

4.1 Before you start

Same prerequisites as the Ladder walkthrough — a Nexus.io AC on your LAN, the mSmart Universal IO wired up, and the [Walkthrough scenario](#) chapter read.

4.2 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.)

4.3 Step 2 — Add the mSmart Universal IO module

Same as Walkthrough 1 Step 2 — open **Motor** → **IO Configuration**, click **Add Device**, pick the `mSmart-Universal-IO` IODF, 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.

4.4 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.

4.5 Step 4 — Add the memory tags

Same as Walkthrough 1 Step 4 — same seven memory tags. The Python code in Step 5 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>
<code>call_motor</code>	BOOL	Memory Bit	<code>%MX3</code>
<code>alarm_latched</code>	BOOL	Memory Bit	<code>%MX4</code>
<code>clear_alarms</code>	BOOL	Memory Bit	<code>%MX5</code>

4.6 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:


```

"""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)

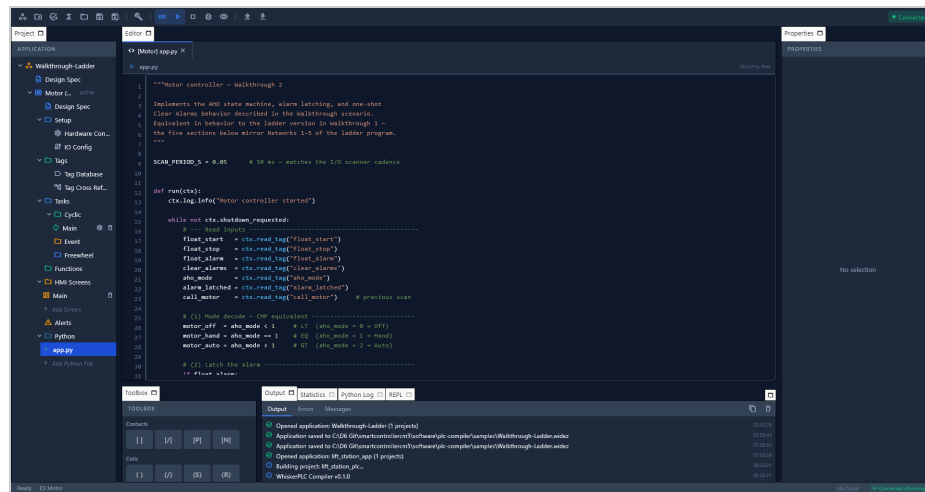
# --- Write outputs
-----

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 AC). 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.

4.7 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.

4.8 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, the fastest path is:

1. Open the Ladder walkthrough's `walkthrough-Ladder.widez` in a second IDE window (File → Open).
2. In its Project Tree, right-click **Motor** → **HMI** → **Screens** → **Main** and choose **Copy**.
3. Switch back to this walkthrough's window, right-click your project's **HMI** → **Screens** node, choose **Paste**.

Otherwise build it from scratch following Walkthrough 1's HMI step.

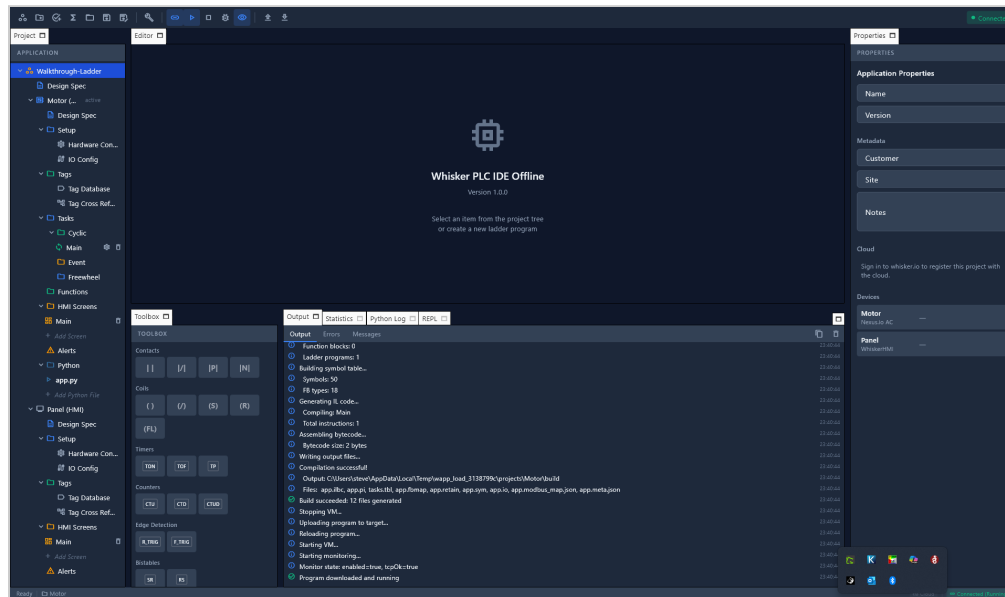
4.9 Step 8 — Connect and Build

Same as Walkthrough 1 Step 8. Connect to the controller, press **F5**.

This time the build output mentions Python:

```
[INFO] Building project: Motor...
[INFO] Compiling alerts (1)...
[INFO] Bundling Python application (app.py, 1.2 KB)...
[INFO] Generating app.io for I/O scanner...
[OK] Build succeeded: 4 files generated
[INFO] Uploading to 192.168.1.244...
[OK] Upload complete (28 KB)
```

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.



4.10 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.

4.11 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.

4.12 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 (Chapter 15 of the main 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, log it to a file the controller writes nightly. See Chapter 13 of the main manual for the full `ctx` API.

5 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 standalone 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).

5.1 Before you start

- Walkthrough 1 (or 2) is complete and the Motor PLC project is running on a Nexus.io AC.
- 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.)

5.2 Step 1 — Reopen the Walkthrough-Ladder Application

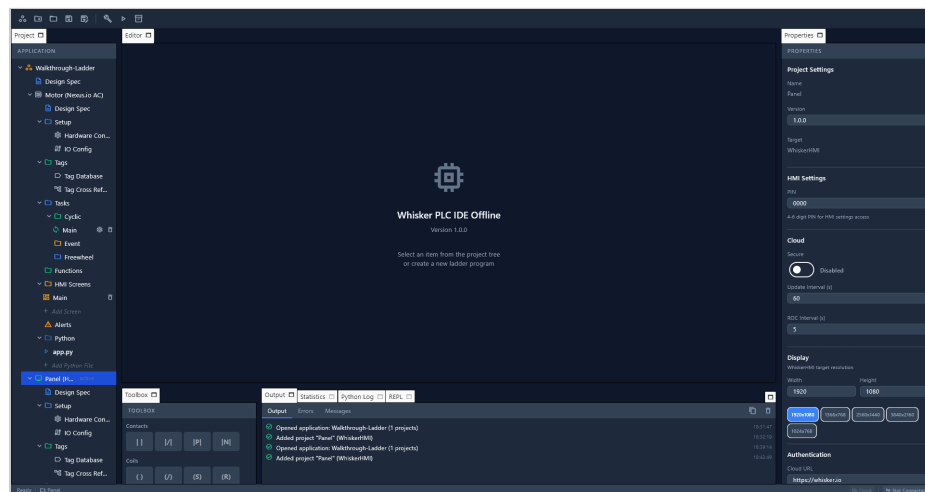
1. **File** → **Open Application**, pick `Walkthrough-Ladder.widez`. The IDE loads the Application; the Project Tree shows the **Motor** project from Walkthrough 1.

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.

5.3 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).



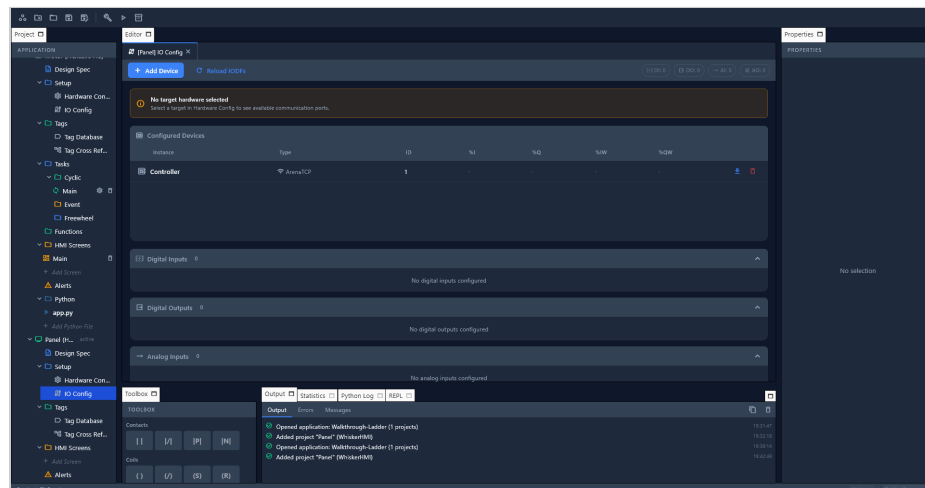
5. Press **Ctrl+S** to save the Application.

What WhiskerHMI looks like in the tree. A WhiskerHMI project has no Programs branch (no ladder), no Functions, no Alerts — only Tags, IO Configuration, and HMI. The IDE knows the target type doesn't support those features and hides the irrelevant branches.

5.4 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 → IO Configuration → 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 AC's IP from Walkthrough 1.
 - **TCP port:** `5000` (the arena agent default).
4. Click **Add**. 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.

5.5 Step 4 — Import tags from the Motor project

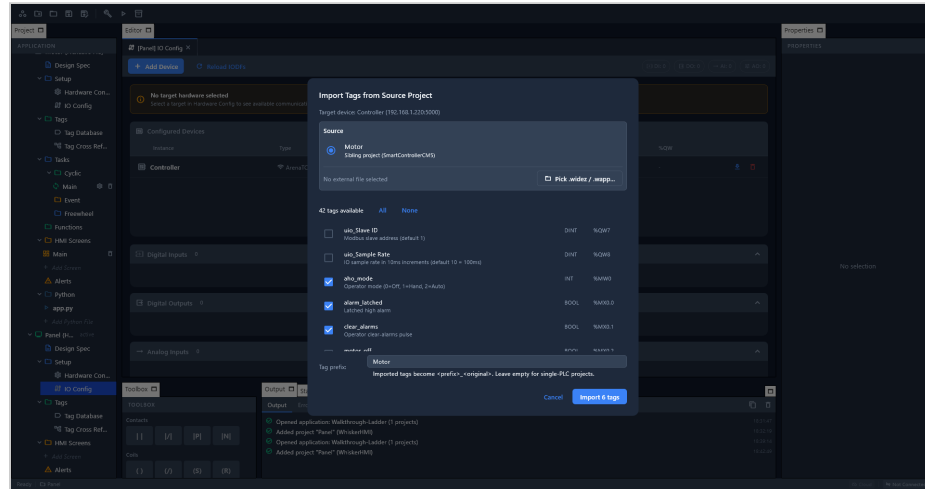
Now bring the seven HMI-facing tags from the Motor project into the Panel. Doing this through the import flow keeps the names, types, and controller addresses in sync — no retyping, no transcription errors.

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.)
3. The dialog lists every tag from the Motor project with its type and address. Tick the seven HMI-facing tags:

Tag	Type
<code>float_start</code>	BOOL
<code>float_stop</code>	BOOL
<code>float_alarm</code>	BOOL
<code>motor_run</code>	BOOL
<code>aho_mode</code>	INT
<code>alarm_latched</code>	BOOL
<code>clear_alarms</code>	BOOL

Leave the four internal scratch tags (`motor_off` , `motor_hand` , `motor_auto` , `call_motor`) unticked. The HMI doesn't need them — `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 7 tags imported and closes.
- Open **Panel** → **Tags** → **Tag Database** to verify. All seven 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.

5.6 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.

5.7 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:

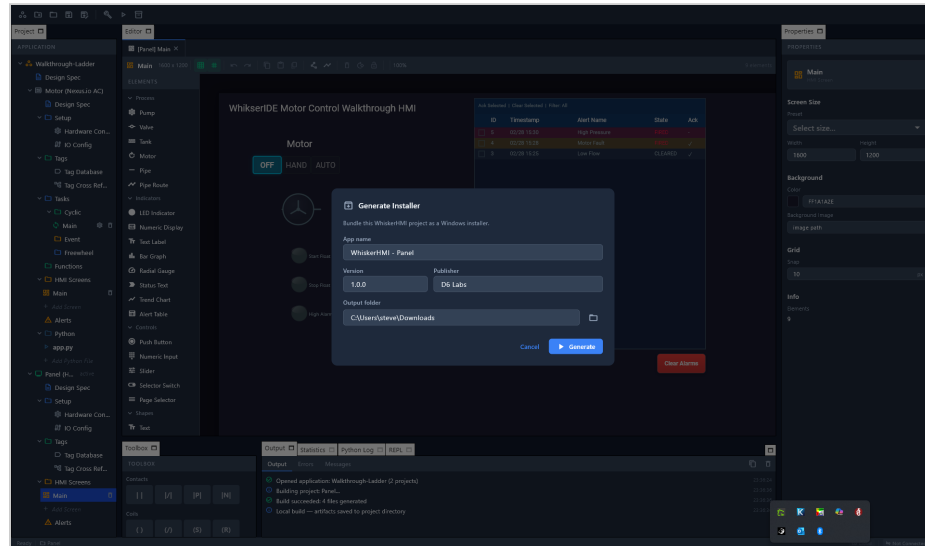
```
[INFO] Building project: Panel...  
[INFO] Generating screens.hmi (1 screen)  
[INFO] Generating symbols.json (7 tags)  
[INFO] Generating io_scanner_config.json (1 ArenaTCP source)  
[INFO] Generating app_config.json  
[OK] Build succeeded: 4 files generated
```

3. The IDE puts the build output in a `build/Panel/` folder under your Application directory. The Motor project's build is in `build/Motor/` — they don't interfere.

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.

5.8 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



- Click **Generate**. The IDE invokes Inno Setup; after a few seconds the Output panel reports:

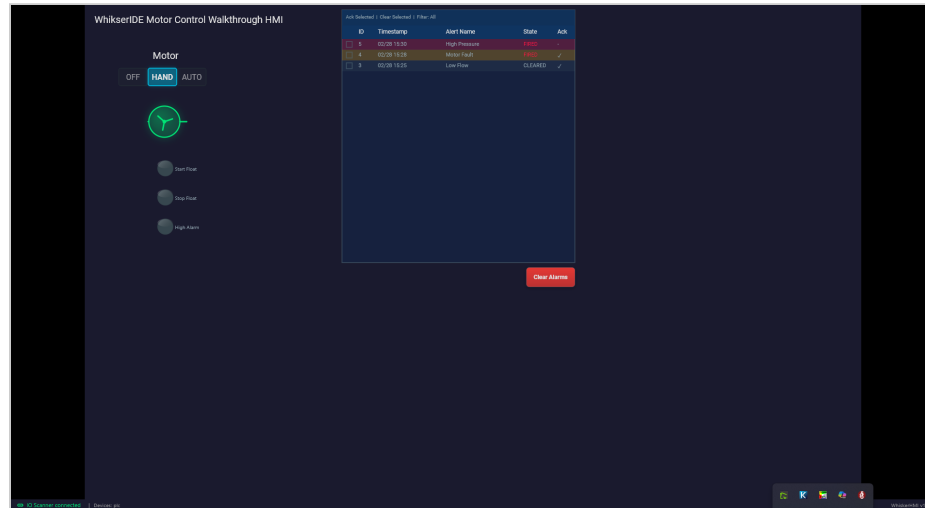
```
[INFO] Bundling WhiskerHMI runtime...
[INFO] Bundling project config (Panel)...
[INFO] Running Inno Setup...
[OK]  Installer created: WalkthroughMotorPanel-
Setup-1.0.0.exe (24.7 MB)
```

- The installer .exe lands in the output folder. That's the whole deliverable — a single file you can copy to any Windows PC.

5.9 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:

- User Account Control** prompts for permission — click **Yes**.
- 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.)
- 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 (arena agent) isn’t blocked by a firewall between them.

5.10 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.

5.11 What’s next

- Set the WhiskerHMI runtime to auto-launch on the panel PC’s boot (Chapter 17 of the main manual).
- 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 about WhiskerHMI's security model (PIN-gated settings, signed configuration bundles) in Chapter 16 of the main manual before shipping a panel into production.

6 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 PLC and HMI targets. You know enough now to build real things.

The rest of this section points at where to deepen each piece.

6.1 The full user manual

The Quick Start Guide you just finished is an extract from the full Whisker IDE user manual. The full manual covers:

Chapter	Topic
2	The Workspace — every panel, every toolbar button, every keyboard shortcut
3	Projects and Applications — multi-project layouts, sharing tags
4	Controller setup — first-time provisioning, network, time
5	Target hardware — every supported HDL, ports, expansion
6	Tag Database — full address map, classes, types, PVs
7	IO Configuration — every supported IODF, polling, error handling
8	Ladder editor — every element, function blocks, custom FBs
9	Functions and tasks — periodic, event-driven, freewheel
10	HMI Designer — every widget, themes, layout, multi-screen flow
11	Alerts editor — conditions, severity, output tags, latch policies
12	Modbus server — address allocation, range customization

Chapter	Topic
13	Python editor — the full <code>ctx</code> API, threading, dependencies
14	Connecting and deploying — security modes, discovery, OTA
15	Monitoring and debugging — Tag Monitor, Output & Python Log windows, AI Fix
16	Security — PKI, signed bundles, RBAC
17	Generating installers — WhiskerHMI panel deployment in detail

The full manual lives at d6labs.com/docs/whisker-ide-manual and ships as an HTML and PDF alongside this Quick Start Guide.

6.2 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 Chapter 12 (Modbus Server) for the full address conventions and read-only vs. read-write rules.

6.3 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.

6.4 Connecting to a device

Click **Connect** in the toolbar, then pick the target from the discovered list or enter its IP. You no longer need a project open to connect.

When you connect, the IDE reconciles the project you have open against the project stored on the target:

- **No project open, target has one** — the IDE offers to **download** the project from the target and open it. Decline, and the connection is cancelled (there'd be nothing to work with).
- **Your open project matches the target's** — it simply connects.
- **They differ** — the IDE asks whether to **download from the target** (this closes your open project, prompting you to save first) or keep the one you have. If you keep yours, live monitoring is **limited** when the target holds a different *version* of the same project, and **disabled** when it holds a *different* project — the tag addresses wouldn't line up with what's actually running.
- **Deploying** a project to a target that already holds a *different* project prompts you to confirm before it's replaced.

Because the target 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 at **Project Properties** → **Deployment** → **Store editable source on target**.

See Chapter 14 (Connecting and Deploying) for security modes, discovery details, and OTA updates.

6.5 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**.

6.6 When you get stuck

- **Output panel errors** usually have an *AI Fix* button next to them — clicking it sends the error context to the Whisker IDE's built-in AI helper, which suggests a one-rung, one-line, or one-config-field fix you can apply with one click.
- **The Output and Python Log windows** are your live view into the controller, right inside the IDE. The **Output** window (bottom of the workspace) shows build results and ladder/runtime messages; the **Python Log** window streams `ctx.log` output from your Python code as it runs. Both update live while you're connected, and every entry is selectable and copyable for pasting into a bug report.
- **Appendix AC of the full manual** is a troubleshooting guide organized by symptom (motor doesn't run, HMI shows blank, Build fails, etc.).

6.7 Support

- **Documentation site:** d6labs.com/support/documentation
- **Email support:** support@d6labs.com
- **Phone (US business hours):** 1-844-365-8647

6.8 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.